AI-based Test Automation: A Grey Literature Analysis

Filippo Ricca*, Alessandro Marchetto†, Andrea Stocco‡

*Università degli Studi di Genova, Genoa, Italy, filippo.ricca@unige.it
†Independent Researcher, alex.marchetto@gmail.com
‡Software Institute - USI, Lugano, Switzerland, andrea.stocco@usi.ch

***Abstract*—This paper provides the results of a survey of the grey literature concerning the use of artificial intelligence to improve test automation practices. We surveyed more than 1,200 sources of grey literature (e.g., blogs, white-papers, user manuals, StackOverflow posts) looking for highlights by professionals on how AI is adopted to aid the development and evolution of test code. Ultimately, we filtered 136 relevant documents from which we extracted a taxonomy of problems that AI aims to tackle, along with a taxonomy of AI-enabled solutions to such problems. Manual code development and automated test generation are the most cited problem and solution, respectively. The paper concludes by distilling the six most prevalent tools on the market, along with think-aloud reflections about the current and future status of artificial intelligence for test automation.**

***Index Terms*—artificial intelligence, test automation, grey literature**

I. Introduction

The advent of Test Automation (TA) techniques has been a major advancement in empowering software engineers in their QA processes. TA is used to enable a large variety of testing tasks—from automated code analysis, unit testing, acceptance testing, or performance testing—across many different software products, such as web or mobile apps [1].

However, the limitations of TA tools such as Selenium became apparent to developers who sought to use them to develop complex test suites. First, nontrivial testing knowledge and programming skills are still required by these tools, as they offer limited support in the creation of high-quality test code. For instance, the creation of robust locators, or deterministic test scripts, is still largely performed manually by testers. Second, in case of ever-changing requirements and software evolution, the technical limitations affecting such automated tests—such as flaky or fragile tests [2]—make test scripts almost unusable. Test maintenance is still a laborious and difficult task because the level of automation of existing tools in this respect is either very limited or totally absent.

Artificial Intelligence (AI) brings the promise of changing the way TA is performed, impacting the whole testing phase by facilitating or automating testing activities such as test planning, authoring, development, and maintenance. In principle, *can we really ask a machine to test software systems?* Our conjecture is that AI can facilitate identify, design, build, execute, and maintain automated test suites by employing statistical techniques to improve the tasks of testers. A plethora of testing tools that exploit AI-based capability for delivering semi- or fully automated testing have been presented, e.g., in this work, we surveyed 48 tools. Unfortunately, to date, AI testing is a hackneyed term. A clear picture of what AI testing really is, the problems it aims to alleviate, and the solutions being utilized is missing.

In this paper, we aim to investigate the synergy between AI and software testing, and how AI is reshaping test automation. We analyzed sources from the grey literature—e.g., white-papers, magazines, online blog-posts, question-answers sites, survey results, and technical reports—because they are the default fora in which practitioners often share their experiences matured on the field, and propose novel practices, guidelines, and tips. Synthesizing knowledge from the grey literature is a contemporary issue in empirical software engineering research [3]. For instance, works have mined the knowledge by practitioners about selecting the right test automation tool [4], the factors behind the choice of what and when to automate [5], or the TA best practices for developing high-quality test code [2].

From our experience, the grey literature is an unexplored gold mine of AI practices for TA. Potential interesting insights are still hidden as practitioners lack both the time and the scientific background to distill the most relevant solutions rigorously. Our task is to try to surface the source of truth usually held in sparse documentation, or in the minds of professionals, stakeholders, developers, and end-users.

To this aim, we surveyed the grey literature to collect, unify, and organize such existing literature in AI practices for TA, to understand what tools exist, what problems they target, and what solutions do they offer.

The main contribution of our work is two taxonomies of problems and solutions in AI for TA, composed of a rich set of guidelines about different technical aspects of the testing process. We also distilled the set of six most cited AI testing tools that, according to developers, can improve the quality of the TA process using AI. Our taxonomies can be useful to both practitioners and researchers, who can, respectively, use the most quoted tools to deliver better test code and foster future research in this field.

II. Background

A. Test Automation

Test automation encompasses the entire testing process within a company or an organization. The basic elements of TA are *test scripts*, that are test programs executed against a portion of the software under test. A test script executes a sequence of predefined *actions*, implemented as commands and inputs, against the software under test and determines its correct behavior by means of assertions. Assertions are (often) manually-defined oracles that determine whether the implemented functionalities are correct. Automated test scripts can be developed to test any level of the software, such as a unit, an API, or even the system as a whole in an end-to-end user-centric fashion. The last option is the most common in the web and mobile environment.

B. Artificial Intelligence

Artificial intelligence refers to the "emulation" of human intelligence by computers that are programmed to think like humans and mimic their actions and reasoning. The term is also applied to any machine that exhibits traits associated with a human mind such as learning and problem-solving. The goals of artificial intelligence include learning, reasoning, and perception. With no claims of completeness, we introduce the AI-related terms that are necessary to understand the remainder of the paper.

Machine learning (ML) is related to pattern recognition and learning from data in order to solve classification or regression problems. The performance of ML algorithms depends heavily on the representation of the data they are given. Indeed, ML algorithms "learn" to perform some tasks, based on a training phase on representative sample data referred to as training datasets. Machine learning can deal with **supervised learning** problems (e.g., classification, regression), in which training sets are annotated (or labeled) with the ground truth values, or **unsupervised learning** problems (e.g., clustering, or dimensionality reduction), in which no ground truth is given. The last category is **reinforcement learning** (RL). RL algorithms are used to constantly adapt to the environment in which they operate, based on a feedback-directed mechanism. The algorithm chooses an action, observes the consequences of the decision, and adapts its strategy to maximize an expected cumulative reward function.

Many AI tasks can be solved by designing and extracting the right set of features. However, for many tasks, it is difficult to know apriori what features are more relevant to be extracted. To this aim, *artificial neural networks*, and in particular *deep neural networks*, solve the problem of finding the right representation by introducing hierarchies of representations [6]. The input data is sent through layers activated by nonlinear functions. Each layer transforms the raw input (first level) into progressively more abstract representations (inner, or hidden, levels). As such, a deep learning algorithm is able to discover underlying hidden patterns in high-dimensional data, which allows the algorithm to correctly perform a task, even on previously unseen data [6].

Computer Vision (CV) provides techniques for analyzing and understanding images, similar to the way humans perceive them. Popular applications in CV include pattern recognition, image analysis, and optical character recognition.

Finally, *Natural Language Processing* (NLP) enables computers to analyze and understand human language.

III. Related Work

Secondary studies. Trudova et al. [7] performed a systematic literature review (SLR) to highlight the role of AI in TA. Based on the analysis of 34 research papers, they highlight that scholars adopted mostly ML and CV techniques for manual effort reduction, improving test suites effectiveness, and their reusability. Lima et al. [8] present another SLR in which they highlight that ML techniques (supervised, unsupervised, and reinforcement learning) are applied mostly to black-box testing, whereas Artificial Neural Networks and Genetic Algorithms are applied to fuzz and regression testing.

There are works that gathered the practitioners' opinions to improve the overall quality of the testing process [2], [4], [5], [9], [10]. For instance, Ricca and Stocco [2] highlight the TA best practices for test code development and maintenance based on an analysis of several hundred documents from the grey literature. Raulamo-Jurvanen et al. [4] identify different criteria for choosing the right testing tool driven by the practitioners' perspective. A study by Garousi [5] aims to characterize what industry wants from academia in software testing, by soliciting testers' challenges during their activities.

Similarly to these works, we consider the grey literature as a primary source of information. Different from these works, this paper distills and summarizes the AI-based test automation tools, what problems they target, and what solutions they provide. To the best of our knowledge, our work represents a unique and novel contribution to the state of the art.

AI for TA. Zhu et al. [11] propose an automated tool for datamorphic testing that classifies test artifacts into test entities and test morphisms, and synthesizes test strategies to generate test sets that adequately cover mutant test cases. Yalla and Sunil [12] describe an NLP-driven sentence generator that can be used as test input for conversational AI bots by popular automated testing tools/frameworks such as Selenium.

In the context of web development, AI has been utilized for the design and development of graphical user interfaces from image mockups [13], [14], [15]. Recently, also the software testing community has witnessed an increasing adoption of CV techniques for assisting or solving common software engineering tasks [16], such as detecting cross-browser incompatibilities (XBIs) [17], web test migration [18], [19], test repair [20], automated test generation [21], or test prioritization techniques for mobile applications [22].

Different from these works, our work focuses on tools available in the state of the practice, often developed, supported, and maintained by companies with a large user base. The results of our research are intended to complement the state-of-the-art in academia and highlight the developers' desiderata thereby fostering future research in the field.

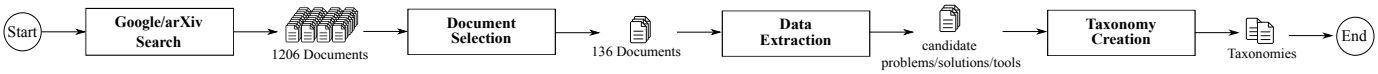

Fig. 1: Overview of the selection procedure

IV. Experimental Study

Our study focuses on the grey literature describing *artificial intelligence (and machine learning) to support test automation*. We consider the following research questions:

RQ_1 (Problems). *What problems in test automation does artificial intelligence help to alleviate?*

RQ_2 (Solutions). *What solutions does artificial intelligence offer to improve test automation?*

RQ_3 (Tools). *What are the most popular AI-based testing tools?*

This section describes the selection procedure we carried out to obtain the relevant documents to answer our research questions, which has been designed according to the guidelines by Garousi et al. [23].

A. Procedure

Figure 1 graphically illustrates our four-phase procedure: (1) Google/arXiv search, (2) document selection, (3) data extraction, and (4) taxonomy creation. In the rest of the section, we provide additional details on each phase.

1) Google/arXiv Search: For the Google search, the authors crafted a search string based on the goal of this study. The output of this step is a set of candidate relevant studies. Additionally, we considered the arXiv database to retrieve relevant papers that are not yet published. This database offers an advanced search feature[1] and has been used in similar studies [3] as a source of grey literature.

To formulate the search string, the authors identified an initial set of candidate keywords starting from the goal of the study. Each tentative search string was then validated against a list of relevant documents, as suggested in the guidelines by Kitchenham and Charters [24]. The final search string is:

> (("artificial intelligence" OR "AI" OR "machine learning" OR "ML")
> AND
> ("test automation" OR "automated testing"))

The first group of terms characterizes words that relate to artificial intelligence and machine learning, whereas the second group of terms narrows the search to automated testing techniques. All relevant documents contained an instance of each keyword from each group (AND operator), whereas keywords within the same group were ORed.

The search was performed from 10 September 2020 to 14 September 2020. For each search query, the first 15 pages of results were scraped, each having 10 documents. We conducted eight queries, which accounted for 1200 documents that were analyzed overall (150 documents for each query). No more significant documents were found after the 15th page. For arXiv, we retrieved only six documents.

[1]https://arxiv.org/search/advanced

2) Document Selection: The Google search is, by construction, very inclusive. This allowed us to collect as many documents as possible in our pool, at the price of having documents that are not directly related to the scope of this study. Accordingly, we defined a set of specific inclusion and exclusion criteria to remove documents not meeting the criteria and ensure that each collected document is in line with the scope of the study.

Inclusion Criteria. First, the document should propose artificial intelligence or machine learning tools or algorithms to support test automation practices, i.e., such as GUI testing, acceptance testing, or functional testing. Second, the document should apply to either capture-replay (C&R), programmable (or script-based), visual, or combinations of these testing approaches. Last, tools' websites and presentations are included as long as they specify useful information. Presentations and slide decks are allowed if they have dedicated and detailed sections discussing tools or algorithms able to support test automation practices.

Exclusion Criteria. We excluded peer-reviewed papers (very few recovered through Google search) in agreement with our goal of conducting grey literature. We also excluded documents not written in the English language, or that provided guidelines for using artificial intelligence within manual testing. Furthermore, we did not consider videos or books, which are quite difficult to extract information from or to retrieve, respectively. We also discarded websites that required registration for consulting the resource. We excluded sources that provided either generic information, e.g., documents only explaining AI or TA.

Results of Document Selection. The studies obtained from the database search were assessed manually by the authors and only those studies that provide direct evidence about the objective of the study were retained. The final number of selected primary documents was 136.

3) Data Extraction: In the data extraction step, the authors read and analyzed in detail the candidate documents, filling out an extraction form with the information gathered from each source [24]. A tabular data extraction form was used to keep track of the extracted information. In particular, each row of such form reports a document and an individual problem or solution. If multiple problem/solution tuples were found within a document, they were collected individually within separate rows. It is important to highlight that no predefined set of problems/solutions was provided; for each newly retrieved hit found during the data extraction phase, a new row was added to the form incrementally.

First, the authors performed a pilot study, in which they labeled a sample of 10 documents selected randomly. The consensus on the procedure and the labels were high, therefore

for the subsequent documents, the authors proceeded with the analysis independently on separate sets of documents. During the mapping, the authors could reuse existing labels previously created, should an existing label apply to the document under analysis. This choice was meant to limit introducing nearly-similar labels for the same problem/solution, and help authors to use consistent naming conventions.

4) Taxonomy Creation: After enumerating all extracted data, the authors began the process of creating a taxonomy for the first two RQs, following a systematic process [25]. For each pair problem/solution, candidate equivalence classes were identified and assigned to descriptive labels. By following a bottom-up approach, the first clustered tags that correspond to similar notions into categories. Then, they created parent categories, in which categories and their subcategories follow specialization relationships.

V. Results

A. RQ_1 (Problems)

Table I presents the list of problems in test automation that, according to our analysis, are addressed with artificial intelligence. Overall, we grouped 339 individual occurrences into six main categories, namely test planning (6%), test design (2%), test authoring (32%), test execution (21%), test closure (14%), and test maintenance (24%). Around 83 occurrences were not assigned to any category, being left either unspecified (18%) or too generic (7%).

The most represented subcategory is *Manual code development* (15%). It is well known that the development of test scripts is a cumbersome activity. When the programmable approach is adopted, this task requires good domain knowledge and sufficient testing and programming expertise because test scripts must be designed and implemented using programming languages such as Java, Python, or Ruby. Researchers have estimated that developing a medium-length Java web test script (about 10/15 commands) takes an average of 7 minutes [26] using, e.g., Selenium WebDriver and JUnit frameworks. Industrial projects have test suites in the size of hundreds or thousands of test cases. Thus, a 7-min development time would quickly add up to a substantial cost for any company.

The second most mentioned subcategory pertains to the maintenance of test scripts (12%). This is known to be a daunting problem, especially in the web and mobile scenarios, which are subject to rapid and continuous evolution. Thus, test scripts must undergo periodic maintenance to stay aligned with the application. Some estimates suggest that this task is almost as time-consuming as development [26], or even the most expensive TA activity [10].

Another relevant subcategory is related to *Untested code* (7%). Testing is a resource optimization problem that testers need to face daily. Since it is impossible to test everything, testers can only automate those test cases that they deem crucial for finding faults. As a consequence, automated test suites can only cover a portion of the application's functionalities. This can have a detrimental effect on the quality of the delivered application because a test suite with low coverage has a lower chance of spotting bugs [27].

TABLE I: Taxonomy of main TA problems by practitioners.

Problem	#
Test Planning (22)	
Critical paths identification	13
Planning what to test	7
Planning long release cycles	2
Test Design (8)	
Programming skills required	5
Domain knowledge required	3
Test Authoring (109)	
Manual code development	52
Manual API test development	7
Manual data creation	19
Test object identification	13
Cross-platform testing	10
Costly exploratory testing	5
Locators for highly dynamic elements	1
Test code modularity	1
Accessibility testing	1
Test Execution (71)	
Untested code	29
Flakiness	18
Slow execution time	14
Useless test re-execution	4
Scalability	2
Parallelization	2
Low user responsiveness	1
Platform independence	1
Test Closure (47)	
Manual debugging overhead	18
Costly result inspection	10
Visual analysis	19
Test Maintenance (82)	
Manual test code migration	3
Bug prediction	11
Fragile test script	10
Regression faults	2
Costly visual GUI regression	8
Maintenance overhead	48
Unspecified	60
Generic	23

Other less numerous, yet representative, categories pertain to *Manual data creation* (5%), *Visual analysis* (5%), *Flakiness* (5%) and *Manual debugging overhead* (5%). Producing high-quality data is a critical part of testing [2] as they have a higher chance of detecting bugs. Unfortunately, realistic test data are often unavailable, and testers need to come up with their own, which requires careful input selection and domain knowledge.

Validating the visual correctness of a GUI is another very challenging task. When performed manually, testers have to check by eye-balling that all visible parts of the application are displayed as intended, often on different devices and platforms. Typically, screenshots of the application under test are compared to a previously stored baseline (i.e., a golden master), and any visually significant difference is reported.

Another problem that plagues test automation is test flakiness. A test script is flaky when its execution on the same application yields different outcomes due to environmental factors such as the screen size, the version of the browser, or the network traffic [28]. This problem poses a threat to the very use of test automation because non-deterministic test cases are more likely to miss faults (false negatives) or report erroneous bugs (false positives).

Manual debugging is another prevalent problem experienced by testers, which requires them to trace the root-cause for test script failures in the application under test. At last, we mention test results inspection activities, i.e., analyzing, interpreting, and validating the results obtained from a test suite execution, which is also a time-consuming task.

B. RQ_2 (Solutions)

Solutions. Table II presents the list of solutions provided by the use of artificial intelligence. Overall, we grouped 307 individual occurrences into four main categories, namely test generation (41%), test oracle (12%), debugging (20%), and test maintenance (26%). Around 116 occurrences were not assigned to any category, being left either unspecified (30%) or too generic (8%).

Among the solutions, the most represented category is *automated test generation* (20% overall). Ideally, the dream would be to automatically produce a test code with human-level quality. This is an ambitious goal even for AI. Indeed, in 47% of the cases, the documents we analyzed do not specify how automated test code generation is implemented. However, interesting solutions are proposed. For instance, TestSigma uses NLP to drive the creation of tests. The approach is as follows. A tester manually initiates a test in the tool, which analyzes the sentences and divides them into segments. These segments can be organized into blocks that specify actions, targets, and input data. As a tester uses limited and precise language, this is far easier to analyze than everyday speech. Other AI-based testing tools can automatically produce test scripts mimicking the behavior of real users (3%) by analyzing the operations and inputs by real users on the app, or by inferring test data automatically (6%).

The second most mention category pertains to the *maintenance* of test scripts. Most effort has been devoted to the development of self-healing mechanisms that use AI to proactively detect and fix threats before their occurrence impacts the production or the test code. This form of anticipatory testing has been implemented mainly through self-healing test scripts (8%) or smart locators (6%). Both solutions aim to keep test scripts and applications aligned during the evolution, but in practice, this is achieved quite differently.

Self-healing test scripts offer a *corrective* action when a test script breaks, through an automated repair procedure that fixes the test script automatically. For instance, *Mabl*, one of the testing frameworks found in the analyzed documents, aims to identify robustly web elements having similar counterparts in their neighborhood, like a table, instead of relying on complex XPath selectors. Suppose the user wants to generate a locator for an element based on its position in a list or table. Mabl will look at similar elements on the page that can function as potential anchors for the target element, along with a confidence rating. The user can edit the identifying list of attributes based on the use case. Upon breakage, the broken locator is shown to the tester, along with several potential fixes that the user has to manually validate. The selected fix is then automatically propagated to all tests using that locator.

TABLE II: Taxonomy of AI Solutions to TA problems.

SOLUTION	#
Test Generation (125)	
Automated test generation	29
Automated test generation using machine translation	11
Automated test generation from user behaviour	11
Automated test generation from API calls	6
Automated test generation from mockups	3
Automated test generation using crawling	2
Automated data generation	22
Robust element localization	13
Dynamic properties recognition based on user behaviour	8
Automated exploratory testing	7
Object recognition engine	6
Mock generation	3
Self-learning	2
Automated API generation	1
Page object recognition	1
Oracle (38)	
Visual testing	38
Debugging (62)	
Intelligent test analytics	17
Automated coverage report	14
Noticeable code changes identification	12
Runtime monitoring	10
Flaky test identification	7
Bad smell identification	1
Decoupling test framework from host	1
Maintenance (81)	
Self-healing mechanisms	43
Self-healing test scripts	24
Smart locators	19
Intelligent fault prediction	12
Test selection intelligent test case re-execution	12
Intelligent waiting sync	5
Intelligent test case prioritization	4
Automated identification environment configurations	3
Pattern recognition	1
Remove unnecessary test cases	1
Unspecified	91
Generic	25

Smart locators, on the other hand, operate a *preventive* action that hinders test scripts from breaking. Such locators are deemed smart because of their resilience to prevent test breakages. Smart locators are equipped with multiple constructs that are updated dynamically as the app evolves. This gives them the capability to keep test scripts up-to-date with little to no human effort. In practice, smart locators are implemented in different ways. An interesting case is the Testim testing framework, which uses a redundant set of locators for each test script's element [29] to train a predictive model that evaluates the reliability (presumed robustness) of each construct. The weights of the model are updated continuously based on the developers' changes to the web app. For instance, if a developer changes the text value of an element (e.g., "Login" $\rightarrow$ "Login Now!"), the corresponding `text` attribute is given a negative weight on the overall prediction because it is regarded as an unstable attribute. In practice, by maintaining multiple attributes per element as well as an updated predictive model, the robustness of test scripts is highly improved. Indeed, based on such statistical analysis, Testim is always able to select effective candidate fixes (i.e., alternative correct locators). The level of automation is arguably superior to the previous Mabl case, even though the techniques can be combined, e.g., smart locators can be used within self-healing test scripts.

TABLE III: Most mentioned AI-based TA tools from the grey literature.

Tool	Reference	Platform	Category	# Hits
Functionize	https://www.functionize.com	Web and Mobile	Test generation and maintenance	19
Applitools	https://applitools.com	All	Test maintenance, Cross-platform testing	18
Mabl	https://www.mabl.com	Web	Test generation and maintenance	18
Testim	https://www.testim.io	Web	Test maintenance	15
Test.ai	https://www.test.ai	Mobile	Test generation and maintenance	14
Appvance.ai	https://www.appvance.ai	Web	Test generation	11

Among other notable examples of *maintenance* reduction techniques, we mention two interesting industry-level cases. At Netflix, the company uses Reinforcement Learning agents for test case prioritization in a CI environment. Their algorithm is inspired by the work by Spieker et al. [30]. On a similar line, Facebook has implemented a predictive test selection technique to estimate the probability of each test failing for a newly proposed code change. Unlike typical regression test selection (RTS) tools, the system automatically develops a test selection strategy by learning from a large data set of historical code changes and test outcomes [31].

Debugging is the third subcategory, with intelligent test analytics being the most represented (5%). Our survey of the literature highlighted two main use cases. First, intelligent test analytics is used to make predictions based on historical data and spot the portions of the application under test with the highest probability of containing bugs. Second, it can be also used to analyze test reports and identify the reason for the failure of test scripts (root cause analysis).

The last subcategory of our taxonomy pertains to *oracles*, specifically those that check the *visual correctness* of the application's GUI (12%). Different CV solutions are used to quickly find functional and visual problems using OCR or image-recognition techniques. We have identified two main approaches. The first approach uses a comparative approach using the golden master to replace assertions. *Applitools* is an instance of such an approach: upon test execution of the test script, CV is used to compare the currently visualized state of the page against the correspondent ground truth represented by the golden master, and reporting human-perceptible differences. This approach is particularly effective for regression and multi-browser testing, in which analyzing all GUIs of an app across different platforms is virtually impossible to perform manually. In the second approach, used for GUI testing at eBay, a set of correct and flawed images are used to train a deep neural network classifier to detect visual imperfections, such as images partially occluded by other images or text.

C. RQ_3 (Tools)

Our analysis revealed a proliferation of AI-based TA tools or TA tools that increasingly adopt some AI. Particularly, we found 48 different tools that were mentioned in the analyzed documents. For the sake of space, Table III lists only the tools that were mentioned at least 10 times, ordered by ascending number of hits. Overall, we observed a trend towards GUI-based system testing, as often in the context of test automation. Next, we briefly describe the main characteristics of the tools in Table III.

Functionize is the most mentioned tool, with 19 hits. The tool performs NLP-based test script creation and AI-assisted maintenance to dynamically update test scripts. Particularly, Functionize proposes self-healing maintenance strategies based on intelligent element selection. Moreover, it includes smart suggestions capable to automatically identify the root cause of broken tests as well as intelligent visual differences recognition.

Applitools is the second most mentioned tool, and it is focused on visual GUI testing. Applitools is mostly used for automated visual assertions employing numerous CV algorithms that can filter human imperceptible differences. The tool integrates smoothly with other testing frameworks (e.g., Selenium), to which it offers a programmatic API for enabling visual analysis. Moreover, Applitools facilitates cross-browser testing of web applications.

Our third-ranked tool is *Mabl*. Mabl is a full-fledged test automation platform built for CI/CD. It is built as a SaaS solution that is used for creating, executing, and maintaining tests. For example, Mabl features a link-crawler that autonomously explores a web application and generates test scripts that cover all reachable paths. Mabl offers also a self-healing test script solution: during test script creation, the tool records several dozens of attributes for each web element, which are used to locate them during software evolution.

Other promising tools are *Testim*, *Test.ai* and *Appvance.ai*. In brief, Testim adopts an intelligent capture-replay approach to the generation of test scripts and guarantees change-resistant smart locators. Differently, Test.ai allows controlling a running AI-Bot capable of autonomously navigating the application under test, collecting and labeling data, and creating test scripts. Finally, Appvance IQ supports two methods of test creation. First, a visual script writer enables test engineers to create test scripts without coding. Second, an AI autonomous scripting generator uses ML and cognitive generation to produce test scripts based on the actual user's activity.

D. Threats to Validity

The main threat to the *internal validity* of this work is the possibility of introducing bias and errors during the selection of documents and classification of the considered items (i.e., problems, solutions, and AI-based test automation tools). Moreover, we may have missed relevant documents that are not captured by our search queries. Thus, we do not claim

that our survey captures all relevant grey literature but we are confident that the included documents cover the most important tools. To minimize classification errors, the authors followed a systematic and structured procedure with multiple interactions. Each doubt concerning creating a new category or classifying an item was discussed among the authors.

Concerning the *external validity*, we considered only Google/arXiv documents in a specific time frame, and our taxonomy may not generalize to other documents or other search engines and repositories. Finally, other relevant classes of items might be unrepresented or underrepresented within our taxonomy. We tried to mitigate this threat by selecting a diverse range of documents using different search queries.

VI. Findings and Remarks

How good are AI-based Test Automation testing tools. The growing importance of AI-based testing tools is testified by their adoption within large companies such as eBay, Netflix, and Facebook. In order to fully understand the potential and benefits of such tools, comparative experiments should be carried out (evidence-based software engineering), which are however challenging given that companies' policies prevent transparent access to the algorithms underlying such tools.

Our survey of the grey literature witnessed a proliferation of AI-based TA testing tools. We distinguish two main cases. First, there are companies (mainly start-ups) that emerged with the sole purpose of bridging AI and QA together. A second category pertains to companies already established on the market that have decided to introduce AI/ML to provide a better user experience and increase testers' accuracy.

Unfortunately, being commercial frameworks, additional details on the algorithms and solutions offered by such tools are often not disclosed. Our preliminary analysis revealed Functionize as the tool able to tackle the greatest number of problems listed in Table I. However, to date, it is not possible to comprehensively understand how the existing problems in TA were addressed. Indeed, the main limitation of the documents we analyzed consist of their low technical depth when describing the solutions listed in Table II.

What AI-based Test Automation testing tools are good for. The list of solutions offered by the introduction of AI/ML is very broad and ranges from the automatic generation of test cases to their automatic repair. Automated oracles were limited to the visual correctness of the presentation layer, whereas the generation of more functional oracles was not considered in the list of documents we surveyed. A current trend in research is to consider metamorphic oracles for testing AI-based software [32]. Despite promising results, we found no applications of such oracles in the grey literature.

From the considered articles, it appears that AI can generate test cases and test code using several different approaches, which is utilitarian for automating more manual tests. However, the testing activity concerns the identification of faults, which consists of finding corner cases. From our preliminary analysis, AI/ML techniques still have to advance to reach this level of effectiveness, since, at the moment, it can only generate quite simple test scripts.

Finally, our analysis shows that AI is mainly applied for unit, regression, and functional end-to-end testing. We hypothesize that applying AI/ML to other testing levels will be more challenging. For example, functional and unit tests are easier to generate using AI (e.g., the test requirements are easier to specify), whereas integration tests are much harder, as they require a more advanced setup and complex flow.

The role of the tester in AI-based TA tools. Despite the high claims, our analysis highlighted that we are far from having fully-automated AI-based testing frameworks able to autonomously generate robust test scripts. Our analysis confirmed our initial hypothesis that AI solutions are used as a toolset to facilitate the use of existing TA tools and practices. In the near future, we do not envision AI to replace but to support human testers, whose role will increasingly be that of a matter expert who knows about the application and how to verify its correct behavior (or lack thereof). More human effort and focus will be on providing better training data, annotations, and analytics.

We recognize two main criticalities of current solutions. Being probabilistic techniques, it is still hard to understand the behavior of AI algorithms as transparency of decisions is sometimes missing. This is quite critical to testing, as test cases must be as deterministic as possible. Second, most solutions have evolved to self-healing paradigms, in which test suites evolve based on the codebase changes. This might be detrimental to the overall fault-finding capability as the risk to miss faults is high (what if tests adapt to faulty code?). For this reason, a safe update and deployment of AI solutions that are meant to test software need to be backed up by strong oracles of fail-safe mechanisms.

VII. Conclusions and Future Work

Artificial intelligence proposes to revolutionize the way we develop and test software systems. Novel tools and testing frameworks are being proposed every year, however, to date, little is still known about AI-based test automation, what problems it addresses, what solutions it offers, and what tools are available, and for what scope.

Towards filling this gap, in this paper we present a study of the grey literature concerning AI solutions for test automation. We manually analyzed several dozens of documents from which we retrieved many problems about different aspects of the automated testing process. Moreover, our taxonomy includes the solutions that are used to mitigate such problems and the list of most popular tools available.

As part of our ongoing and future work, we plan to improve the sources of grey literature with more documents, to assess the generalizability of our taxonomies. Triangulating our results through surveys and semi-structured interviews with developers is also part of the plan to validate our findings. Finally, we intend to run comparative experiments between traditional and AI-based testing tools, to better assess the benefits brought by AI to TA.

REFERENCES

[1] F. Ricca, M. Leotta, and A. Stocco, "Three open problems in the context of e2e web testing and a vision: Neonate," *Advances in Computers*, 01 2018.

[2] F. Ricca and A. Stocco, "Web test automation: Insights from the grey literature," in *Proceedings of 47th International Conference on Current Trends in Theory and Practice of Computer Science*, ser. SOFSEM 2021. Springer, 2021.

[3] V. Garousi, M. Felderer, M. V. Mäntylä, and A. Rainer, "Benefitting from the grey literature in software engineering research," 2019.

[4] P. Raulamo-Jurvanen, M. Mäntylä, and V. Garousi, "Choosing the right test automation tool: A grey literature review of practitioner sources," in *Proc. of the 21st International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '17. ACM, 2017, p. 21–30.

[5] V. Garousi and M. V. Mäntylä, "When and what to automate in software testing? a multi-vocal literature review," *IST*, vol. 76, pp. 92–117, 2016.

[6] A. Stocco, "How artificial intelligence can improve web development and testing," in *Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*, ser. Programming '19. New York, NY, USA: ACM, 2019, pp. 13:1–13:4. [Online]. Available: http://doi.acm.org/10.1145/3328433.3328447

[7] A. Trudova., M. Dolezel., and A. Buchalcevova., "Artificial intelligence in software test automation: A systematic literature review," in *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE,*, INSTICC. SciTePress, 2020, pp. 181–192.

[8] R. Lima, A. M. R. da Cruz, and J. Ribeiro, "Artificial intelligence applied to software testing: A literature review," in *2020 15th Iberian Conference on Information Systems and Technologies (CISTI)*, 2020, pp. 1–6.

[9] H. Gamido and M. Gamido, "Comparative review of the features of automated software testing tools," *International Journal of Electrical and Computer Engineering*, vol. 9, pp. 4473–4478, 10 2019.

[10] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, K. Petersen, and M. V. Mäntylä, "Benefits and limitations of automated software testing: Systematic literature review and practitioner survey," in *2012 7th International Workshop on Automation of Software Test (AST)*, 2012, pp. 36–42.

[11] H. Zhu, I. Bayley, D. Liu, and X. Zheng, "Automation of datamorphic testing," in *2020 IEEE International Conference On Artificial Intelligence Testing (AITest)*, 2020, pp. 64–72.

[12] M. Yalla and A. Sunil, "Ai-driven conversational bot test automation using industry specific data cartridges," in *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*, ser. AST '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 105–107. [Online]. Available: https://doi.org/10.1145/3387903.3389306

[13] T. A. Nguyen and C. Csallner, "Reverse engineering mobile application user interfaces with remaui (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2015, pp. 248–259.

[14] T. Beltramelli, "pix2code: Generating code from a graphical user interface screenshot," *CoRR*, vol. abs/1705.07962, 2017. [Online]. Available: http://arxiv.org/abs/1705.07962

[15] K. P. Moran, C. Bernal-Cárdenas, M. Curcio, R. Bonett, and D. Poshyvanyk, "Machine learning-based prototyping of graphical user interfaces for mobile apps," *IEEE Transactions on Software Engineering*, 2018.

[16] M. Bajammal, A. Stocco, D. Mazinanian, and A. Mesbah, "A Survey on the Use of Computer Vision to Improve Software Engineering Tasks," *IEEE Transactions on Software Engineering*, 2020.

[17] S. Mahajan and W. G. J. Halfond, "Detection and localization of HTML presentation failures using computer vision-based techniques," in *Proceedings of 8th IEEE International Conference on Software Testing, Verification and Validation*, ser. ICST '15, 2015, pp. 1–10.

[18] M. Leotta, A. Stocco, F. Ricca, and P. Tonella, "Automated migration of DOM-based to visual web tests," in *Proceedings of 30th Symposium on Applied Computing*, ser. SAC 2015. ACM, 2015, pp. 775–782.

[19] ——, "PESTO: Automated migration of DOM-based web tests towards the visual approach," *Software Testing, Verification And Reliability*, vol. 28, no. 4, 2018.

[20] A. Stocco, R. Yandrapally, and A. Mesbah, "Visual web test repair," in *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018, p. 12 pages.

[21] C. Zhang, H. Cheng, E. Tang, X. Chen, L. Bu, and X. Li, "Sketch-guided GUI Test Generation for Mobile Applications," in *Proc. of ASE '17*, 2017, pp. 38–43.

[22] Y. Feng, J. A. Jones, Z. Chen, and C. Fang, "Multi-objective Test Report Prioritization Using Image Understanding," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '16, 2016, pp. 202–213.

[23] V. Garousi, M. Felderer, and M. V. Mäntylä, "Guidelines for including grey literature and conducting multivocal literature reviews in software engineering," *IST*, vol. 106, pp. 101–121, 2019.

[24] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," 2007.

[25] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, Árpád Beszédes, R. Ferenc, and A. Mesbah, "BugJS: A benchmark and taxonomy of javascript bugs," *Software Testing, Verification And Reliability*, 2020.

[26] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella, "Capture-replay vs. programmable web testing: An empirical assessment during test case evolution," in *Proceedings of 20th Working Conference on Reverse Engineering*, ser. WCRE 2013. IEEE Computer Society, 2013, pp. 272–281.

[27] L. Brader, *Testing for Continuous Delivery with Visual Studio 2012 (Microsoft patterns & practices)*. Microsoft patterns & practices, 2013.

[28] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 643–653. [Online]. Available: https://doi.org/10.1145/2635868.2635920

[29] M. Leotta, A. Stocco, F. Ricca, and P. Tonella, "Using multi-locators to increase the robustness of web test cases," in *Proceedings of 8th IEEE International Conference on Software Testing, Verification and Validation*, ser. ICST '15. IEEE, 2015, pp. 1–10.

[30] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 12–22. [Online]. Available: https://doi.org/10.1145/3092703.3092709

[31] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive test selection," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2019, pp. 91–100.

[32] C. Murphy, G. E. Kaiser, L. Hu, and L. Wu, "Properties of machine learning applications for use in metamorphic testing," in *Proceedings of the Twentieth International Conference on Software Engineering & Knowledge Engineering (SEKE'2008), San Francisco, CA, USA, July 1-3, 2008*. Knowledge Systems Institute Graduate School, 2008, pp. 867–872.