A Multi-Year Grey Literature Review on AI-assisted Test Automation

Filippo Ricca[a,*], Alessandro Marchetto[b] and Andrea Stocco[c,d]

[a] University of Genoa, Via Balbi 5, Genova, 16126, Italy
[b] University of Trento, Via Sommarive 9, Trento, 38123, Italy
[c] Technical University of Munich, Boltzmannstraße 3, Munich, 85748, Germany
[d] fortiss GmbH, Guerickestraße 25, Munich, 80805, Germany

ARTICLE INFO

Keywords:
Test Automation
Artificial Intelligence
AI-assisted Test Automation
Grey Literature
Automated Test Generation
Self-Healing Test Scripts

ABSTRACT

Context: Test Automation (TA) techniques are crucial for quality assurance in software engineering but face limitations such as high test suite maintenance costs and the need for extensive programming skills. Artificial Intelligence (AI) offers new opportunities to address these issues through automation and improved practices.
Objectives: Given the prevalent usage of AI in industry, sources of truth are held in grey literature as well as the minds of professionals, stakeholders, developers, and end-users. To this aim, our study surveys grey literature to explore how AI is adopted in TA, focusing on the problems it solves, its solutions, and the available tools. Additionally, the study is complemented by expert insights.
Methods: Over five years, we reviewed over 3,600 grey literature sources, including blogs, white papers, and user manuals, and finally filtered 342 documents to develop taxonomies of TA problems and AI solutions. We also cataloged 100 AI-driven TA tools and interviewed five expert software testers to gain insights into AI's current and future role in TA.
Results: The study found that manual test code development and maintenance are the main challenges in TA. In contrast, automated test generation and self-healing test scripts are the most common AI solutions. We identified 100 AI-based TA tools, with Applitools, Testim, Functionize, AccelQ, and Mabl being the most adopted in practice.
Conclusion: This paper offers a detailed overview of AI's impact on TA through grey literature analysis and expert interviews. It presents new taxonomies of TA problems and AI solutions, provides a catalog of AI-driven tools, and relates solutions to problems and tools to solutions. Interview insights further revealed the state and future potential of AI in TA. Our findings support practitioners in selecting TA tools and guide future research directions.

1. Introduction

The development of Test Automation (TA) techniques [49] is meant to advance the quality assurance (QA) processes for software engineers [32, 54]. TA supports a wide range of testing tasks, including automated code analysis, unit testing, integration testing, acceptance testing, and performance testing, applicable to various software products such as web and mobile applications. However, the limitations of TA frameworks like, e.g., Selenium WebDriver [62] and Selenium IDE [54] have become evident when creating complex test suites. These tools still demand substantial testing knowledge and programming skills, providing limited assistance in producing high-quality test scripts. Activities like developing robust locators [36] and deterministic test scripts are still predominantly manual processes [32]. Additionally, maintaining automated test scripts becomes laborious and challenging due to constantly changing requirements and software evolution, resulting in issues like flaky or fragile tests that make test scripts costly to maintain [6, 57].

The integration of Artificial Intelligence (AI) and Machine Learning (ML) into TA is getting significant attention from researchers and practitioners, recognizing AI's potential to bridge the gap between human and machine-assisted testing activities [20, 58]. In this paper, we refer to these techniques as Artificial Intelligence assisted Test Automation (AIaTA). AI holds the promise of transforming TA by simplifying or automating various testing activities, including test planning, authoring, development, and maintenance. Despite the growing adoption of AIaTA by companies [55], there remains a limited understanding of the challenges it addresses, the solutions it offers, and the existing tools and their integration with the software development process.

Several secondary studies have reviewed existing work on AIaTA [26, 29, 38, 47, 67]. This paper is a secondary study that focuses on the grey literature [16] to capture practitioners' perspectives on the adoption of AIaTA. The main goal of our work is to understand the available AIaTA tools, the problems they address, and the innovative solutions they offer, from the developers' perspective.

In our previous work [56, 55], of which this article is an extension, we surveyed the grey literature to collect, consolidate, and organize existing AI practices for TA. This paper builds upon the grey literature reviews by Ricca et al. [55, 56] by incorporating more recent sources, thus presenting an extended, multi-year grey literature review. We provide details on the differences between the prior papers and this article. First, this article contains an expanded empirical evaluation (Section 3) as we extended the study to sources from the year 2024. This comprehensive review spans several years,

*Corresponding author
filippo.ricca@unige.it (F. Ricca)
https://person.dibris.unige.it/ricca-filippo/ (F. Ricca)
ORCID(s): 0000-0002-3928-5408 (F. Ricca); 0000-0002-6833-896X (A. Marchetto); 0000-0001-8956-3894 (A. Stocco)

demonstrating the generalizability of the results and examining the underlying relationships among TA problems, AI-based solutions, and existing tools. Second, to gather further insights on the application of AIaTA, we conducted interviews with five researchers and practitioners to collect their experience. These interviews were recorded, transcribed, and analyzed to identify the key problems, solutions, and tools discussed by the interviewees. Thus, the main contribution of this extended paper is a set of taxonomies of problems, solutions, and tools regarding AIaTA, which are corroborated by experts. To our knowledge, this is the first work that includes interviews with developers on the usage of AI for TA.

Our experience shows that grey literature is a valuable yet underutilized resource for AI practices in TA, containing insights that practitioners may not have the time or scientific expertise to rigorously extract. Our goal is to surface these valuable insights, often found in scattered documentation or within the knowledge of professionals, stakeholders, developers, and end-users. We believe our work can help practitioners understand the current state and practices in AIaTA, aiding in the selection of appropriate tools for their testing needs. Additionally, our findings can guide researchers in identifying issues that need further investigation and new research directions.

This paper is organized as follows: Section 2 provides essential background information to help comprehend the rest of the paper. This includes an overview of the current limitations of TA and an introduction to an AI-based mechanism, self-healing test scripts, which addresses the well-known issue of fragile tests. Section 3 reports research questions, adopted procedure, document selection phase, and data analysis of our multi-year grey literature review. The results are detailed in Section 4, and Section 8 explores these findings in relation to the threats to validity and empirical evidence collected. Section 6, on the other hand, briefly presents the design and results of the interviews conducted with five expert software testers. Finally, Section 9 summarizes related works, and Section 10 concludes the paper.

2. Background

In this section, background information is provided to understand the content of the paper. Specifically, a brief explanation is given of what test automation is, along with the known practical benefits and drawbacks as described in both practice and literature. Subsequently, the rationale behind the utilization of AI and ML in this context is explained, along with the associated benefits. Finally, to provide a more concrete understanding, the process of self-healing test scripts is detailed. This process utilizes ML to automatically adjust failing test scripts due to the evolution of the application under test.

2.1. Test Automation

Test automation is the practice of using specialized software tools or frameworks to control the execution of tests and compare actual outcomes with predicted outcomes [15]. It encompasses the entire testing process within an organization, aiming to improve efficiency, accuracy, and coverage in software testing [18]. The primary component driving test automation is the use of test scripts—i.e., programs that run specific portions of the software being tested. These test scripts perform a sequence of predefined actions against the Application Under Test (AUT), consisting of commands and inputs. The expected results of test scripts are typically described by assertions, which are specific statements provided by a testing framework, e.g., JUnit [41]. Assertions check values, e.g., the result of the call under test, or the final status of some part of the system, against given conditions and raise an exception in case of failure. The testing framework detects these exceptions and marks the tests as failed.

Automated test scripts can target various levels of software [45], addressing different aspects of the system to ensure comprehensive testing coverage. At the most granular level, unit tests focus on individual components or functions within the AUT, verifying that each unit behaves as expected in isolation. This level of testing is crucial for identifying and fixing bugs early in the development process.

Moving up the hierarchy, automated tests can target APIs (Application Programming Interfaces). API testing ensures that the interactions between different software components, as well as with external services, are functioning correctly [27]. This involves sending requests to the API endpoints and validating the responses, including data formats, status codes, and content. API testing is essential for verifying the integrity and reliability of the communication paths within the AUT.

At the highest level, automated test scripts can perform system-wide testing in an end-to-end, user-focused manner. End-to-end (E2E) testing simulates real user scenarios and interactions with the software, covering the entire application flow from start to finish [32]. This approach ensures that all components and subsystems work together seamlessly to deliver the intended user experience. E2E tests are particularly prevalent in web and mobile environments, where user interactions span multiple layers of the application, including the user, back-end services, and databases. This approach is especially important in these kind of environments due to the complexity and variability of these platforms. For example, web applications must be tested across different browsers [44] and devices to ensure consistent performance and user experience. Similarly, mobile applications must be tested on various operating systems, screen sizes, and hardware configurations. Automated end-to-end tests help identify issues that may arise from these variations, ensuring that the application remains working and user-friendly across different environments.

By targeting different levels of the software, from individual units to entire systems, automated test scripts provide a comprehensive and scalable testing solution. This multi-level approach [9] helps ensure that all aspects of the software are thoroughly tested, contributing to the overall quality and reliability of the final product.

Test automation offers numerous benefits [51], including increased testing efficiency, higher test coverage, and improved accuracy by reducing human error. It enables repetitive and regression testing [74], where tests are run frequently to ensure that new changes do not negatively impact existing functionality. This is crucial for maintaining software quality over time.

However, test automation also presents well-known challenges, in particular in the context of web and mobile applications [46, 33]. Traditional automation testing faces issues such as slow test execution and the persistent problem of maintaining test scripts. Slow test execution is a primary reason for delays in testing, often caused by a focus on GUI automation, poorly designed test scripts, and insufficient test case sequencing.

Excessive test maintenance is another major issue, as test scripts are highly sensitive to the application's UI and structure. Any minor change in the UI necessitates corresponding changes in the test script, leading to a significant portion of automation effort being dedicated to test script maintenance.

Additionally, the test script can break due to small changes in the code (test script fragility problem), such as renaming or relocating a GUI component on the screen [65]. Managing and maintaining test data adds another layer of complexity, requiring testers to create test data generation scripts and use version control effectively. Finally, traditional automation testing is complex and code-intensive, and a lack of skilled resources often leads to the failure of test automation projects due to inadequate planning.

2.2. AI-assisted Test Automation

Artificial Intelligence involves enabling computer programs to execute tasks that would typically necessitate human intelligence. Within this broad definition, we find machine learning (ML), which involves pattern recognition and learning from data to solve classification or regression problems [5]. Additionally, Computer Vision (CV) provides techniques for analyzing and understanding images, much like how humans perceive them, with popular applications including pattern recognition, image analysis, and optical character recognition. Finally, Natural Language Processing (NLP) enables computers to analyze and understand human language. In the last ten years, the availability of large benchmarks of labeled data and unprecedented computing power has enabled the application of Deep Learning techniques such as deep neural networks (e.g., convolutional neural networks and recurrent neural networks) to complex vision and spatiotemporal problems. More recently, transformer architectures have enabled the exploitation of language embedding that is at the basis of large language models (LLMs), i.e., novel types of AI that achieved unprecedented performance in NLP and coding tasks.

Traditionally, AI methods and models have been widely applied to various phases of the software development lifecycle, including software testing. The application of AI to support software testing is a well-established and increasingly popular research topic, as evidenced by several recent studies in the literature [1, 11, 21, 28].

In the context of Test Automation, AI is also employed for various purposes [47, 67]. For example, AI and ML algorithms are used to analyze application behavior and user interactions to automatically generate test scripts. This reduces the manual effort required for test script creation and improves test coverage. Another example is self-healing test scripts, in which test constructs are automatically maintained when the test code becomes obsolete or loses sync with the AUT during software evolution. This is meant to diminish the maintenance overhead associated with test automation. Additionally, AI techniques can generate additional test data by analyzing patterns observed in production data. This facilitates the creation of diverse test scenarios, for more comprehensive testing of the AUT. Lastly, ML algorithms can detect anomalies in test results, including unexpected behavior or performance issues. This allows for early detection of defects, mitigating their impact on the production environment. Additionally, AI can enhance test execution by intelligently scheduling tests, considering factors such as code changes, risk profiles, and resource availability.

3. Empirical Study

Our research centers on the grey literature that discusses the application of AIaTA. We consider the following research questions:

RQ$_1$ (Issues/problems). *How does AI contribute to mitigating challenges in TA?*

RQ$_2$ (Solutions/approaches). *In what ways does AI provide solutions to enhance TA?*

RQ$_3$ (Tools and Platforms). *Which AI tools/platforms are widely used in the context of software testing?*

RQ$_4$ (Problems vs. Solutions). *What is the relationship between TA problems and solutions in AIaTA?*

RQ$_5$ (Tools vs. Solutions). *How are AIaTA tools and solutions interconnected?*

RQ$_1$ identifies the main TA problems and issues that are supported by AI-based solutions, i.e., AI-enhanced testing techniques and tools. To answer RQ$_1$, we manually built a taxonomy of TA issues and problems addressed by AI-based solutions, considering the sources of grey literature.

RQ$_2$ analyzes the solutions proposed for TA problems and issues that are based on AI algorithms and tools. To answer RQ$_2$, we built a taxonomy of AI-based solutions used to address TA issues and problems identified in RQ$_1$. In this case, as well, the information was inferred using grey literature sources.

RQ$_3$ identifies the existing tools and platforms that assist in the testing phase with AI-based solutions presented in the literature. To answer RQ$_3$, we extracted from grey literature existing tools that implement the AI-based solutions identified in RQ$_2$.

RQ$_4$ maps TA problems and issues with proposed AI-based solutions. To answer RQ$_5$, we mapped the problems

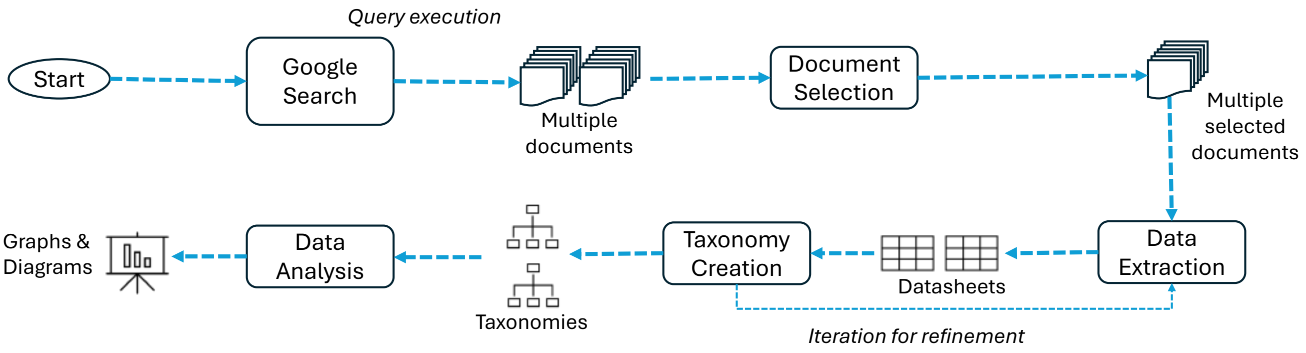


Figure 1: Overview of the procedure adopted in each iteration of the grey literature.

and issues identified in RQ_1 with their proposed AI-based solution identified in RQ_2.

Finally, the last research question RQ_5 maps such proposed AI-based solutions towards existing tools that implement them. To answer RQ_6, we mapped the proposed AI-based solution identified in RQ_2 with the implemented tools supporting them, as listed in RQ_3.

3.1. Procedure

To review the grey literature relevant to our research questions, we carried out a selection procedure designed according to the guidelines by Garousi et al. [16]. Particularly, we iterated the five-step procedure defined in our initial publication [55] three times, with an approximately one-year interval between each iteration, and further refined it in our extended study [56]. Figure 1 shows these five steps:

1. Google search: by using a string query, we searched for potentially interesting documents.
2. Document selection: by applying inclusion/exclusion criteria, we checked the initial set of potentially interesting documents, thus identifying a reduced number of documents of interest, to be further analyzed.
3. Data extraction: we read and analyzed each document of interest to extract information concerning TA problems, AI-based solutions, and tools.
4. Taxonomies creation: we created taxonomies of problems, solutions, and available tools.
5. Data analysis: we further analyzed the collected information using Sankey diagrams [60] and manual in-depth analysis for discovering relationships.

In the rest of the section, we present each step in detail.

3.1.1. Google Search

The Google search was intentionally broad to gather a large pool of documents. This approach was intended to maximize the retrieval of all relevant documents, at the cost of including documents that were not directly relevant to our study. Thus, we established specific inclusion and exclusion criteria to filter out the irrelevant documents and ensure that the remaining ones align with the study's scope. We used Google in incognito mode to conduct our searches, to ensure that our search results were not influenced by previous search histories or personalized algorithms. For our Google search, we used the following query string:

((“artificial intelligence” OR “AI” OR “machine learning” OR “ML”)
AND (“test automation” OR “automated testing”))

The first part of the query string is characterized by words related to artificial intelligence and machine learning, whereas the second part is related to automated testing. All relevant documents contained an instance of each keyword from each part of the string (AND operator), whereas keywords within the same part were ORed. Hence, we have applied eight queries overall in each of the three repetitions of the search. For each query, the first 15 pages of results were scraped, each having 10 documents. No relevant documents were found after the 15^{th} page. The eight applied queries led us to collect 1,200 documents for each repetition (150 documents for each query), thus resulting in a total of 3,600 documents collected and then analyzed in the three repetitions. The output of this step is a list of candidate web documents potentially regarding TA conducted with AI-based solutions. Table 1, among other aspects, reports the number of documents collected in this step for each iteration (column “# Docs”) and the date on which the search was conducted (column “# Date”).

3.1.2. Document Selection

The list of candidate web documents has been analyzed according to a set of inclusion and exclusion criteria, for filtering out irrelevant documents. We mainly considered three inclusion criteria: (i) the document needs to investigate AI or ML tools or methods that can support TA; (ii) the document should apply to either capture-replay, programmable (or script-based), visual, or combinations of these testing approaches; (iii) tools' websites and presentations are included as long as they specify useful information, and (iv) presentations and slide decks are included only if in scope. We mainly excluded documents (exclusion criteria) if: (i) they are scientific peer-reviewed papers; (ii) they are not written

Table 1
Summary of data related to the document search and taxonomies construction.

Iteration	Date	#Docs	#Selected	#Non-Con.	#Problems	#Solutions	#Tools
First [55]	Sept. 2020	1,200	156	14	35	35	50
Second [56]	Febr. 2023	1,200	95 (251)	4	5 (40)	9 (44)	17 (67)
Third	Apr. 2024	1,200	91 (342)	6	7 (47)	15 (59)	33 (100)

in the English language; (iii) they provide guidelines for using AI within manual testing; (iv) they are videos or books; (v) they are part of the website that requires registration; and (vi) they contain only generic information on AI or software testing, without relating the two concepts. These criteria have been applied by analyzing each document obtained in the initial search with Google, thus only those documents that provide direct evidence about the study's objective were retained. The output of this step is the list of relevant selected documents regarding TA conducted with AI-based solutions. Table 1 (column "# Selected") reports the number of documents selected for the analysis in this step for each iteration, among the ones collected in the Google search step. In parentheses, we have reported the cumulative total value across the various iterations. We can see that, in each repetition, we selected around 10% of the collected web documents, for an overall selection of about 9.5% (i.e., 342 out of 3,600 documents) of documents identified in the Google Search phase.

3.1.3. Data Extraction

In this step, each relevant candidate document has been read and analyzed in detail and the following information has been collected: (1) web link to the document, (2) name and surname, if any, about the author(s) of the document, (3) the publication date of the document (if available), (4) the type of the document (e.g., blog post, interview transcript), (5) the test automation tool(s) investigated or described in the document, (6) the testing level discussed (e.g., unit, integration, system, acceptance), (7) the test automation problems and issues addressed, and (8) the solution offered. Table 2 reports a summary of the most frequent types of documents among the selected ones, e.g., blog posts, and interview transcripts (column "Document Type" in the top part of the table). We can see that most of the documents were web articles or blogs. Overall, we were able to classify most of the document, i.e., 76.9%. Table 2 reports also the number of documents in which the author's name was not specified (row "Without author(s)" in the bottom part of the table). We can see that it was impossible to identify the name of the authors for a non trivial number of documents, overall for 37.1% of documents, with an increased trend in the last repetition.

For each iteration, the authors conducted a pilot study by labeling a randomly selected sample of 10 documents. Each author labeled independently the sample of documents (e.g., type of TA problems and TA solutions), then during a meeting each document that was labeled differently by at least two authors was re-analyzed by all the authors together

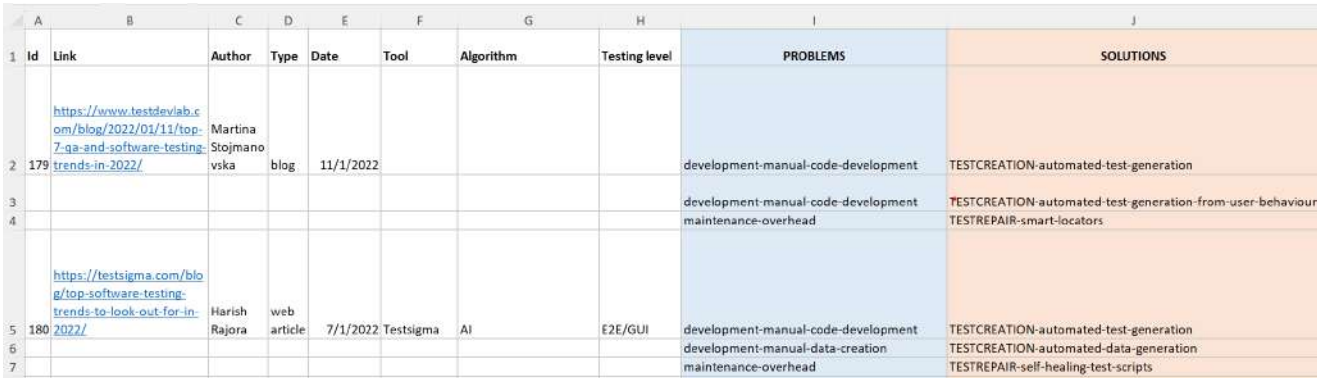
Table 2
Summary of types of documents.

Document Type	# First [55]	# Second [56]	# Third
Blog	38	24	35
Interview transcript	0	2	0
Manual	0	2	0
Post	2	7	8
Slide	8	0	0
StackOverflow	2	0	0
Technical feed	0	0	3
Tool	6	3	1
Web article	46	40	30
White-paper	5	0	0
Total	108 (69.2%)	78 (82.1%)	77 (84.6%)
		263 (76.9%)	
Without author(s)	53 (33.9%)	22 (23.1%)	52 (57.1%)
		127 (37.1%)	

and the labels were discussed for reaching an agreement. These discussions were also used to refine the descriptions of the different categories of the taxonomies, thus reducing ambiguous definitions. Only in a few of the compared cases (4.5%), we started the discussion for reaching an agreement since we observed different labels. Instead in the other cases (95.5%), we observed the same labels for at least two classifiers, out of three, and for all three classifiers in the majority of the cases (59.1%). Generally, hence, we observed that the consensus on the classification of the documents was high and the discussions quite limited, allowing the authors to proceed with the analysis independently on separate sets of documents for the subsequent phases.

In detail, we build a tabular representation of the data extracted from the documents implemented as an online spreadsheet on Google Docs. Figure 2 shows an image of the online spreadsheet on Google Docs we used to gather data, which details the contribution aspects. Each row of the table reports the information collected for a document and an individual problem or solution. Hence, more rows can be used to detail several TA problem-solution tuples found in a single document. During the mapping process, the authors reused existing labels whenever applicable to avoid introducing nearly identical labels for the same TA problem/solution and to maintain consistent naming conventions. We recall that the set of TA problems and TA solutions considered has been defined iteratively by starting from a small set of documents (as documented in [55]). Each newly identified problem and/or solution was incrementally considered, and the document was re-analyzed if needed.

In the three iterations, not all documents contributed to the identification of relevant TA problems, solutions, and tools supporting them. Table 1 (column "# Non-Con.") reports the number of documents that did not contribute to the taxonomies in each iteration. We can notice that the reported numbers are rather limited and, in particular, they decrease between the first iteration and the subsequent ones, meaning that the selected documents tend to discuss more and more interesting content.



	A	B	C	D	E	F	G	H	I	J
1	Id	Link	Author	Type	Date	Tool	Algorithm	Testing level	PROBLEMS	SOLUTIONS
2	179	https://www.testdevlab.com/blog/2022/01/11/top-7-qa-and-software-testing-trends-in-2022/	Martina Stojmanovska	blog	11/1/2022				development-manual-code-development	TESTCREATION-automated-test-generation
3									development-manual-code-development	TESTCREATION-automated-test-generation-from-user-behaviour
4									maintenance-overhead	TESTREPAIR-smart-locators
5	180	https://testsigma.com/blog/top-software-testing-trends-to-look-out-for-in-2022/	Harish Rajora	web article	7/1/2022	Testsigma	AI	E2E/GUI	development-manual-code-development	TESTCREATION-automated-test-generation
6									development-manual-data-creation	TESTCREATION-automated-data-generation
7									maintenance-overhead	TESTREPAIR-self-healing-test-scripts

Figure 2: A screenshot of the publicly available Google Docs spreadsheet featuring the data extracted from the documents.

Table 3
Summary of Credibility and Content Quality.

	First [55]	Second [56]	Third	Total
Credibility				
Low	39 (42.8%)	39 (57.3%)	36 (42.8%)	12 (4.5%)
Medium	33 (36.2%)	22 (32.2%)	33 (39.2%)	104 (39.8%)
High	19 (20.8%)	7 (10.2%)	15 (17.8.7%)	145 (55.5%)
Content Quality				
Low	8 (8.1%)	2 (4.1%)	1 (1.1%)	114 (46.9%)
Medium	32 (32.6%)	35 (47.9%)	37 (41.1%)	88 (36.2%)
High	58 (59.1%)	35 (47.9%)	52 (57.7%)	41 (16.8%)

It is worth noting that the quality of documents typically found in grey literature, such as those analyzed in this study, tends to be more diverse and often more challenging to evaluate compared to traditional systematic literature reviews. It is also more complex to conduct a sort of quality assessment of the used sources. By inspired the quality assessment checklist reported by [16], we focused on: (i) the presence of authors; (ii) the quality and understandability of the content of the document; and (iii) a subjective assessment of the credibility of the document as perceived by the classifiers, i.e., the authors in our case. For each iteration, we collected this information for each selected document. In particular, the content of the document and the document's credibility are scored in terms of Low/Medium/High according to the judgment of the classifiers. We recall that Table 2 (row "Without author(s)" in the bottom part of the table) reports the number of documents for which an author was not identified.

Table 3 summarizes the results in terms of content quality and document credibility as assessed by the authors. We can notice that overall only a limited amount of documents have been classified as Low in terms of credibility (i.e., 4.5%). However, the classifiers did not provide any judgment for approximately 23% of the documents (i.e., 342 selected documents minus 12+104+145), suggesting uncertainty or low credibility for these sources. Concerning the document content quality, we can notice that overall a large set of documents have been classified as of Low quality (i.e., 46.9%). Moreover, the classifiers did not express any judgment for about 28% (i.e., 342 selected documents minus 114+88+41) of the documents, for which we can assume a doubt to low content quality. These numbers suggest that, on one hand, the overall quality of the considered sources is sufficient for analysis; on the other hand, they confirm that grey literature is not always reliable [16].

3.1.4. *Taxonomy Creation*

By collecting the identified tuples of test automation problems and solutions, as well as the list of existing tools, two taxonomies have been constructed to answer RQ_1 and RQ_2, following a systematic process [19], while the list of tools has been used to answer RQ_3. To build the taxonomies, we clustered related TA problems and/or TA solutions, thus identifying categories of problems and solutions. Then, parent categories have been created, by following a specialization relationship between categories and subcategories. Especially, at the beginning of the process, some iterations between this step and the previous one, related to Data extraction, were needed to clearly identify the relevant information. The last columns of Table 1 report, for each iteration, the number of identified categories of problems (column "# Problems") and solutions ("# Solutions") in the two taxonomies and the number of identified tools ("# Tools"). In parentheses, we have reported the cumulative total value across the various iterations.

3.1.5. *Data Analysis*

We answered the first three research questions (RQ_1, RQ_2 and RQ_3) by analyzing the two taxonomies of problems and solutions and the list of tools. Further analysis has been conducted after the construction of the taxonomies by considering the information related to the tuples of TA problems, solutions, and tools. In particular, several Sankey diagrams [60] have been constructed aiming at visualizing the possible interconnections between AI-based solutions and TA problems (to answer RQ_4), and AI-based tools and AI-based solutions (to answer RQ_5). In a Sankey diagram, the relations between nodes are shown by links that connect input and output nodes (in our case, respectively, solutions–problems and tools–solutions), while the width of a link indicates the relevance or magnitude of the relationship. By using the Sankey diagrams, we identified trends and patterns

Table 4
Taxonomy of test automation problems identified in the first [55], second [56] (in green), and third iteration (in light blue).

Problem	[55]	[56]	#
Test Planning	22	32	52
Critical paths identification	13	13	19
Planning what to test	7	9	11
Planning long release cycles	2	2	2
Test process management	-	8	20
Test Design	8	22	55
Programming skills required	5	11	25
Domain knowledge required	3	9	12
Designing effective Testcases	-	1	17
Adherence to coding standards	-	1	1
Test Authoring	110	133	220
Manual code development	52	58	108
Manual API test development	7	11	13
Manual data creation	19	24	37
Test object identification	13	13	19
Cross-platform testing	10	15	26
Mimic geo-location testing	1	1	2
Costly exploratory testing	5	5	6
Locators for highly dynamic elements	1	1	1
Test code modularity	1	1	1
Accessibility testing	1	1	1
Adequacy-focus on faulty-areas	-	3	5
Optimize test strategies	-	-	2
Test Type	-	-	3
Conducting thorough security testing	-	-	2
Testing-AI-generated-code	-	-	1
Unspecified	60	77	133
Test Execution	71	103	118
Untested code	29	34	40
Flakiness	18	22	22
Slow execution time	14	16	23
Useless test re-execution	4	7	9
Scalability	2	3	13
Parallelization	2	3	5
Low user responsiveness	1	1	1
Platform independence	1	1	5
Test Closure	47	59	96
Manual debugging overhead	18	22	26
Costly result inspection	10	11	23
Visual analysis	19	25	39
Data Quality	-	1	1
Code Quality	-	-	7
Test Maintenance	82	102	184
Manual test code migration	3	3	4
Bug prediction	11	13	17
Fragile test script	10	12	27
Regression faults	2	7	7
Costly visual GUI regression	8	10	19
Maintenance overhead	48	57	100
Change Requirements	-	-	1
Communication developers-testers	-	-	5
Troubleshooting and root cause analysis	-	-	4
Generic	23	33	55
Total	423	545	917

in the relations among TA solutions, problems, and tools, thus being able to answer RQ_4 and RQ_5. More specifically, we employed `pandas` data-frames [42] to filter information related to the problems addressed and solutions provided (RQ_4), as well as the tools used (RQ_5). We excluded entries where the problem, solution, or tool was either unspecified or too generic. For the relevant entries, we counted each pairwise combination of `<problem, solution>` and `<tool, solution>`, treating each instance of a problem-solution pair or tool-solution pair as a unique connection. We utilized the `plotly` library [23] to generate the Sankey diagrams [60], though it is important to note that the diagrams presented in this paper focus only on the most prominent connections for the sake of space and readability, while the complete set of diagrams encompasses all collected sources.

3.2. Iterations

As previously mentioned, we conducted three iterations of the described procedure. The first search was conducted in September 2020 while the second search was conducted in February 2023. These iterations have been documented in our previous published studies [55, 56]. The third iteration search was conducted in April 2024 and it is presented in this paper. In each iteration, we started from the previously built taxonomies and refined them by adding new categories and values if needed.

4. Results

4.1. RQ_1 (Issues/problems)

Table 4 reports the existing TA problems and issues and the number of individual occurrences. The table reports the data collected in the first [55] and in the second [56] (in green) iteration, as well as the updated version related to the third iteration (in light blue).

Overall, in the three iterations, we identified 917 individual occurrences (545 after the second iteration, and 423 after the first one) into the seven main categories: (1) test planning, 5.6% of occurrences; (2) test design, 5.9% of occurrences; (3) test authoring, 24.1% of occurrences; (4) test type, 0.3% of occurrences; (5) test execution, 12.8% of occurrences; (6) test closure, 10.4% of occurrences; and (7) test maintenance, 20% of occurrences. Other two categories such as Unspecified and Generic collected respectively 14.5% and 5.9% of occurrences.

The most represented subcategory is *Manual code development* (11.6%), which was not surprising since it is well known [31, 34] that the development of test cases and scripts is a complex task that requires non-trivial domain knowledge and appropriate testing and programming skills.

The second most mentioned subcategory is related to *Maintenance overhead* (10.9%). Nowadays, software applications evolve continuously, in particular, in fields such as web and mobile. This continuous evolution of software applications requires continuous maintenance and evolution also of corresponding test suites, thus being aligned with the applications. Existing research [31, 34] shows that test maintenance is a highly expensive and time-consuming task,

Table 5
Taxonomy of test automation solutions identified in the first [55], second [56] (in green), and third iteration (in light blue).

Solution	[55]	[56]	#	Solution	[55]	[56]	#
Test Generation	125	192	302	**Debugging**	62	82	149
Aut. test generation	29	48	74	Intelligent test analytics	17	22	27
Aut. generation using machine translation	11	13	25	Automated coverage report	14	22	28
Aut. generation from user behaviour	11	22	29	Noticeable code changes identification	12	13	23
Aut. test generation from API calls	6	6	13	Runtime monitoring	10	11	12
Aut. test generation from mockups	3	3	3	Flaky test identification	7	8	9
Aut. test generation using crawling	2	11	16	Bad smell identification	1	1	1
Aut. test generation using GenAI	-	-	18	Intelligent log analysis	-	-	5
Codeless test generation	-	-	11	Intelligent test reporting	-	-	6
Declarative testing	-	2	3	Decoupling test framework from host	1	1	7
Predict faulty-areas	-	4	4	Root-cause-analysis	1	10	10
Aut. data generation	22	22	32	Prediction of failures	-	3	21
Robust element localization	13	13	14				
Dynamic user-behaviour properties recognition	8	8	8	**Maintenance**	81	141	201
Automated exploratory testing	7	10	12	Self-healing mechanisms	43	43	63
Object recognition engine	6	13	19	Self-healing test scripts	24	32	55
Mock generation	3	3	3	Smart locators	19	21	24
Self-learning	2	3	7	Intelligent fault prediction	12	13	13
Automated API generation	1	9	9	Intelligent selective test re-execution	12	12	12
Page object recognition	1	2	2	Intelligent waiting sync	5	5	5
				Intelligent test prioritization	4	6	11
Test Optimization	-	-	12	Aut. identification environment configurations	3	6	6
Improve test quality	-	-	8				
Static (AI) Code Analysis	-	-	2	Pattern recognition	1	1	3
Improve test scalability	-	-	1	Remove unnecessary test cases	1	1	6
Anonymous test data	-	-	1	Reduce UI testing	-	1	3
Test Execution	-	8	40	**Test Process**	-	-	33
Cloud execution	-	2	11	AI data-driven test decisions	-	-	13
Decoupling test framework from host	-	2	8	AI (Chat-bots) for communication	-	-	4
Smart test execution	-	3	19	Shift Left Testing	-	-	2
Anomaly detection	-	1	1	Hyper Automation Testing	-	-	1
Headless execution	-	-	1				
				Test type	-	-	4
				IOT testing	-	-	2
				Blockchain testing	-	-	2
Oracle	38	46	70	Unspecified	91	99	119
Visual testing	38	46	70	Generic	25	30	55
Total					466	607	972

and in some cases, it can even be the most costly test automation activity [51].

Other representative subcategories, even if less numerous, are: *Untested code* (4.3%), *Visual analysis* (4.2%), and *Manual data creation* (4%).

Untested code and *Manual data creation* are key aspects to optimize the generation of effective test cases. On the one side, it is impossible to test everything in a software application, so strategies and techniques to identify the less tested portion of the application are fundamental for reducing the overall effort by focusing on where it is needed. For instance, it has been demonstrated that test suites with low coverage of the app code have a lower chance of detecting bugs [7]. On the other side, the use of high-quality test data is a critical part of testing [57], but producing such high-quality test data can be time-consuming.

Validating the visual correctness of a GUI (*Visual analysis* subcategory) is a particularly challenging task. When done manually, testers must visually inspect all elements of the application to ensure they appear as intended, often across multiple devices and platforms. This process generally involves comparing screenshots of the current application against a previously established baseline, or "golden master", and reporting any significant visual discrepancies.

Another significant challenge in test automation is test flakiness (*Flakiness subcategory*). A test script is considered flaky if its execution on the same application results in inconsistent outcomes due to environmental factors like screen size, browser version, or network conditions [30, 39]. This issue undermines the reliability of test automation, as flaky tests are more prone to missing defects (false negatives) or reporting incorrect errors (false positives).

4.2. RQ$_2$ (Solutions/approaches)

Table 5 reports the existing AI-based solutions to the evidenced TA problems and issues (RQ$_1$). The table reports the data collected in the first [55] and in the second [56] (in green) iteration, as well as the updated version related to the third iteration (in light blue).

The solutions have been grouped into eight main categories. In the table, TA solutions and their occurrences are listed for iterations. Overall, in the three iterations, we identified 972 individual occurrences (607 after the second iteration, and 466 after the first one) into the eight main categories: (1) test generation, 31% of occurrences; (2) test oracles, 7.2% of occurrences; (3) debugging, 15.3% of occurrences; (4) test maintenance, 20.6% of occurrences; (5) test process, 2% of occurrences; (6) test execution, 4.1% of

Table 6
The most popular AI-based tools used in the context of software testing, as identified in the first [55], second [56] (in green), and third iteration (in light blue).

	[55]		[56]		Third Iteration	
	Tool	**#**	**Tool**	**#**	**Tool**	**#**
1	Testim	14	Applitools	24	Applitools	50
2	Applitools	12	Testim	20	Testim	38
3	Functionize	11	Functionize	16	Functionize	35
4	Mabl	7	Mabl	14	AccelQ	32
5	LambdaTest	6	TestCraft	9	Mabl	29
6	Laucnhable	5	Parasoft SOA test's Smart API Test Generator	8 8	TestCraft	21
7	Parasoft SOA test's Smart API Test Generator	5	SauceLabs	8 8	Testsigma	18
8	QMetry Digital Quality Platform	5 5	TestComplete	8	Katalon	15
9	Testsigma	5	UiPath Test Suite	8	ChatGPT	14
10	UiPath Test Suite	5	AccelQ	7	TestComplete	12
11	Test.AI	4	ChatGPT	7	Parasoft SOA test's Smart API Test Generator	10
12	Tricentis Tosca	4	LambdaTest	7	UiPath Test Suite	10
13	AccelQ	3	Testsigma	7	LambdaTest	9
14	Appium	3	Test.AI	6	SauceLabs	9
15	Google OSS-Fuzz	3	Eggplant AI	5	Aqua	8
16	Kobiton	3	Katalon	5	Test.AI	7
17	Percy	3	Laucnhable	5	Testrigor	7
18	SauceLabs	3	Parasoft Selenic	5	Appium	6
19	TestCraft	3	QMetry Digital Quality Platform	5	Eggplant	6
20	TestProject	3	Tricentis Tosca	5	Parasoft Selenic	6
21	Appvance IQ	2	Appium	4	pCloudy	6
22	Browsershots	2	BrowserStack	4	Tricentis Tosca	6
		50		**67**		**100**

occurrences; (7) test optimization, 1.2% of occurrences; and (8) test type, 0.4% of occurrences. Other two categories such as Unspecified and Generic collected respectively 12.2% and 5.6% of occurrences.

The most represented subcategory is *Automatic test generation* (189 occurrences out of 972, 19.4%, by considering all automatic test generation subcategories). Automatic test case generation, however, remains an ambitious task, even when AI-based technologies are used. We could observe that 7.6% of the occurrences related to automatic test generation did not give any indication about how automated test code generation is implemented. Existing works generate tests automatically by: (i) starting from the analysis of the real users' behaviors (2.9%); (ii) by applying machine translation techniques (2.5%), for example, natural language processing techniques that develop test scripts starting from test requirements descriptions written in natural language; and (iii) recently, by adopting generative AI strategies (1.8%).

The second most mentioned subcategory is related to *Visual testing* (7.2%). To implement visual testing strategies different computer vision solutions are applied to automatically identify functional and visual problems in the application GUI, in particular, by applying image-recognition and OCR techniques to identify graphical elements and detect changes among the different tested versions.

Another representative subcategory, even if less numerous, is related to test maintenance, i.e., *Self-healing mechanisms* (6.4%). Self-healing refers to the capability of automatically applying corrective actions when a test script is broken, for instance, after an application maintenance or evolution task, without human intervention. The continuous evolution of modern applications breaks test scripts, thus requiring a large effort to repair such test scripts. Following the analysis of the documents, we further split this subcategory into two distinct categories: *Self-healing test script* (5.6%) and *Smart Locators* (2.4%). The adoption of smart locators can be relevant to prevent the need for repairing interventions. By using multiple constructs that can be updated dynamically as the application evolves smart locators tend to be resilient to test breakages. In detail, by using multiple attributes per web element to locate, smart locators improve the robustness of test scripts.

4.3. RQ$_3$ (Tools and Platforms)

Table 6 shows that we identified in total 100 tools that support the TA solutions strategies (RQ$_2$) and that were discussed in the analyzed documents in the three iterations (67 after the second iteration, and 50 after the first one). Among these tools, the table lists the twenty-two most frequently cited and discussed ones. Considering the third iteration, the top-5 tools are: *Applitools*,[1] *Testim*,[2] *Functionize*,[3] *AccelQ*,[4] and *Mabl*.[5]

Applitools focuses on GUI testing by applying automated visual testing strategies based on different computer vision technologies and algorithms. Testim applies intelligent capture-replay approaches and GPT-based technology

[1]https://applitools.com
[2]https://www.testim.io
[3]https://www.functionize.com
[4]https://www.accelq.com
[5]https://www.mabl.com

for automatic test generation and adopts smart locators to prevent test breaking. Testim also supports AI data-driven testing decision strategies. Functionize uses advanced NLP technologies for automatic test generation and adopts AI-based strategies for applying self-healing maintenance, in particular, for dynamically updating test scripts based on the application changes. Mabl uses a crawler for exploring a web application aiming at automatically generating test scripts by covering all reachable parts of the application under test. Mabl offers also a self-healing test script solution and supports AI data-driven testing decision strategies. Finally, AccelQ uses generative AI to automatically generate tests and offers self-healing test scripts, as well as an efficient cloud-based test execution.

Our survey extension highlighted a significant increase in the availability of new tools for developers. Their number has grown significantly in a short time, and new options have appeared on the market. We find ChatGPT, Aqua, TestRigor, and pCloudy among the new tools and platforms. ChatGPT, an advanced language model from OpenAI that generates human-like text, is primarily used for creating test scripts in software testing and was the most frequently mentioned tool in the documents we reviewed, with 14 total occurrences. Following that, we have Aqua, TestRigor, and pCloudy. Aqua is a test management platform that leverages AI to optimize test planning, execution, and tracking. Offering cloud-based solutions for both manual and automated testing, it integrates with various tools to improve testing efficiency and effectiveness. TestRigor is an automated testing tool designed for creating and running end-to-end tests using plain English commands. It simplifies test creation and maintenance. pCloudy is a unified app testing platform that ensures app quality across various devices and browsers. It offers cloud-based manual and automated testing with access to thousands of real devices. Leveraging AI, pCloudy enhances integration, efficiency, and effectiveness in testing.

4.4. RQ_4 (Problems vs. Solutions)

Figure 3 utilizes a Sankey diagram to illustrate the relationships between problems and AI-based solutions in the test planning phase: problems are shown on the left, while solutions are shown on the right. In the planning phase, the most relevant problem (the larger node on the left) identified by the community is the identification of test paths, i.e., a specific sequence of actions that are executed to verify that a particular feature or functionality of the application under test is working as expected. This sequence can involve clicking on links, filling out forms, submitting data, and interacting with various elements on the web pages. In particular, testers are interested in identifying the critical paths in the app to test them thoroughly. In terms of solutions adopted, automatic test generation based on crawling and user behaviors are the most frequently discussed. However, as shown in the diagram, other solutions for identifying critical paths are predicting faulty areas in the application under test and intelligent test analytics. It is worth noting that test process management has become an increasingly relevant issue over the years. In fact, there has been a noticeable rise in individual discussions on this topic within the community. As shown in Table 4, there were no occurrences of this issue identified in the first iteration, eight occurrences in the second iteration, and twenty occurrences in the most recent iteration. The most commonly used solution for this problem is using AI and data analytics to guide the software testing lifecycle. This approach leverages various technologies and methods to make informed decisions based on data rather than relying solely on manual processes or intuition.

Similarly, Figure 4 shows that the most relevant test problem when designing tests, concerns the programming skills required to develop test scripts (the larger node on the left), thus as expected, several different strategies and technologies are proposed (we can see in the diagram different output nodes for programming skills) for automatically generating test scripts and test data, e.g., the adoption of generative AI, NLP, codeless test scripts (e.g., by visual testing mechanisms). The second most important problem we can note is the design of test cases, which involves determining what to test, the steps to take to create a test case, and what the oracle is. We can see from the diagram that various solutions are proposed for this problem, ranging from using automated exploratory testing to automatic generation, for example by using generative AI.

Figure 5 represents the Sankey diagram problems/solutions in the authoring phase and shows that the most relevant problems in this phase are related to the manual effort needed to develop/implement test cases in test scripts and create effective test data (see the two larger nodes on the left). Among the solutions to limit the manual effort in implementing test cases, we find that the predominant category, as we would expect, is automatic generation. The subcategories include all the automatic generation methods found, such as using screen mockups as a starting point, employing a web crawler, and even utilizing generative AI tools/platforms. Regarding the problem of generating effective test data, the main solution is to rely on AI-based tools capable of generating it. It is interesting to note that test object identification—the ability of testing tools/platforms to correctly recognize and interact with various elements (objects) on a web page, e.g. dynamic or modal elements, and cross-platform problems—cross-platform testing involves verifying that a software application functions correctly across different environments, including various operating systems, browsers, and devices. Planning cross-platform testing presents unique challenges due to the diversity and complexity of environments on which the application must be tested—remain among the most relevant problems, in all the three considered iterations. In particular, the identification of test objects is faced with several strategies such as (i) object recognition engines (i.e., identification of testing elements in the GUI); (ii) robust element localization (e.g., use of multiple sources to localize elements in the GUI); (iii) visual testing (i.e., automated visual checks of the GUI by using of computer vision); and (iv) intelligent fault prediction strategies.

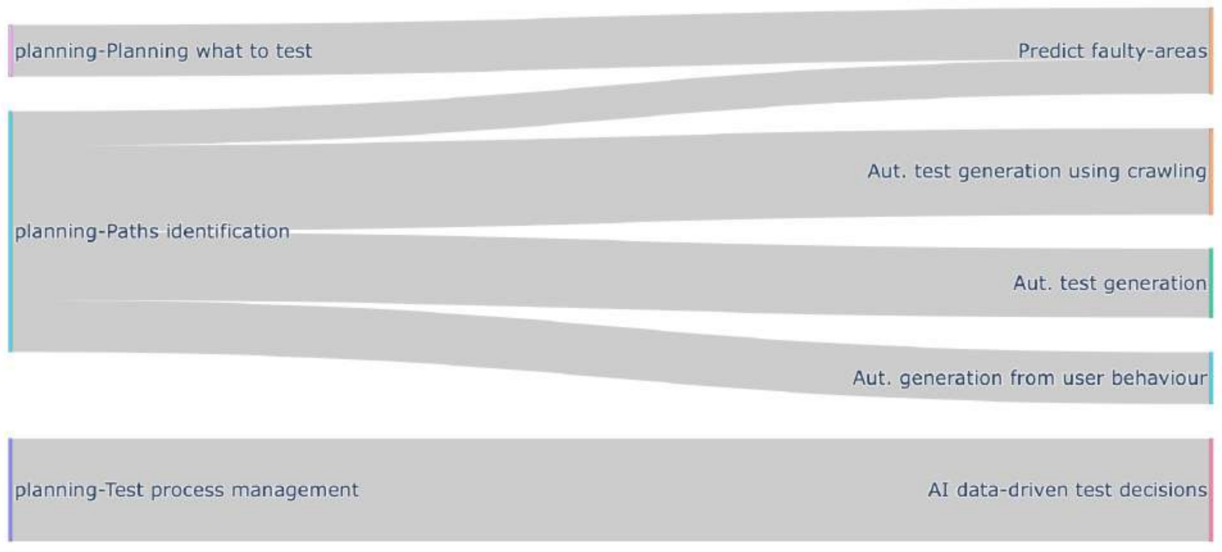

planning-Planning what to test
planning-Paths identification
planning-Test process management
Predict faulty-areas
Aut. test generation using crawling
Aut. test generation
Aut. generation from user behaviour
AI data-driven test decisions

Figure 3: Problems vs Solutions: Test planning Sankey diagram.

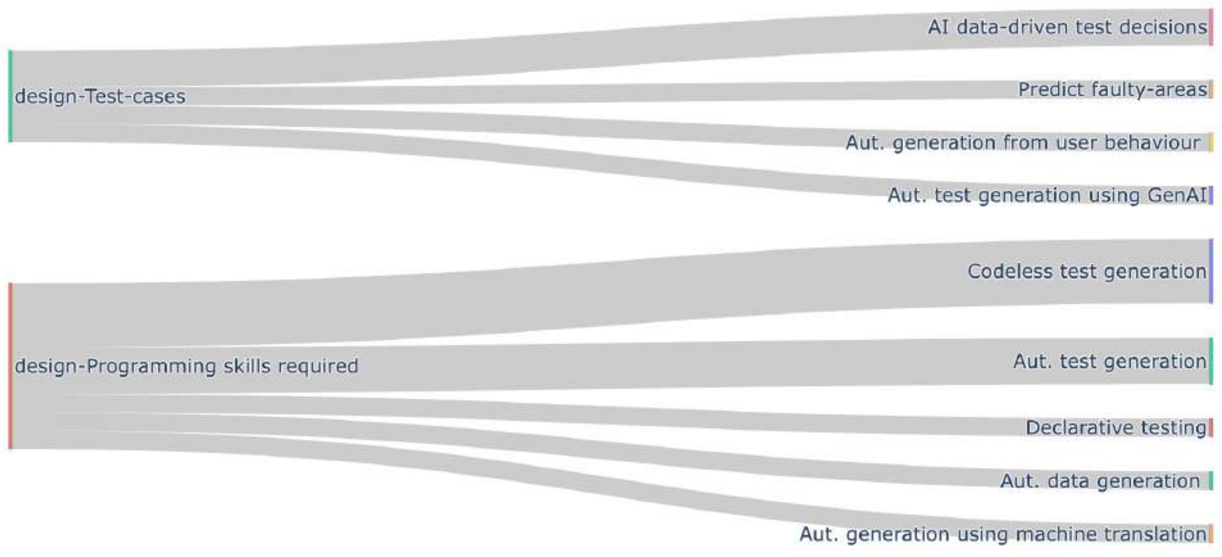

design-Test-cases
design-Programming skills required
AI data-driven test decisions
Predict faulty-areas
Aut. generation from user behaviour
Aut. test generation using GenAI
Codeless test generation
Aut. test generation
Declarative testing
Aut. data generation
Aut. generation using machine translation

Figure 4: Problems vs Solutions: Test design Sankey diagram.

The Sankey diagram represented in Figure 6 shows that several different problems seem to affect the test execution phase. Among the various problems, the most significant and those that attract the community's attention seem to be flakiness and untested code. In the last iteration (see Table 4), flakiness—flakiness in tests occurs when test results are unreliable, showing varying outcomes across multiple runs even though the application state remains unchanged—appears to be recognized as less relevant compared to the first and second iteration. Conversely, the problem of identifying the untested code is becoming more and more relevant with the second and third iteration. A multitude of intelligent mechanisms (e.g., fault prediction, code change analysis, smart locators, API generation, exploratory testing, and test generation) are adopted to increase code coverage during testing. In particular, the automatic generation of coverage reports is a key aspect for keeping developers and testers continuously informed about coverage information, enabling

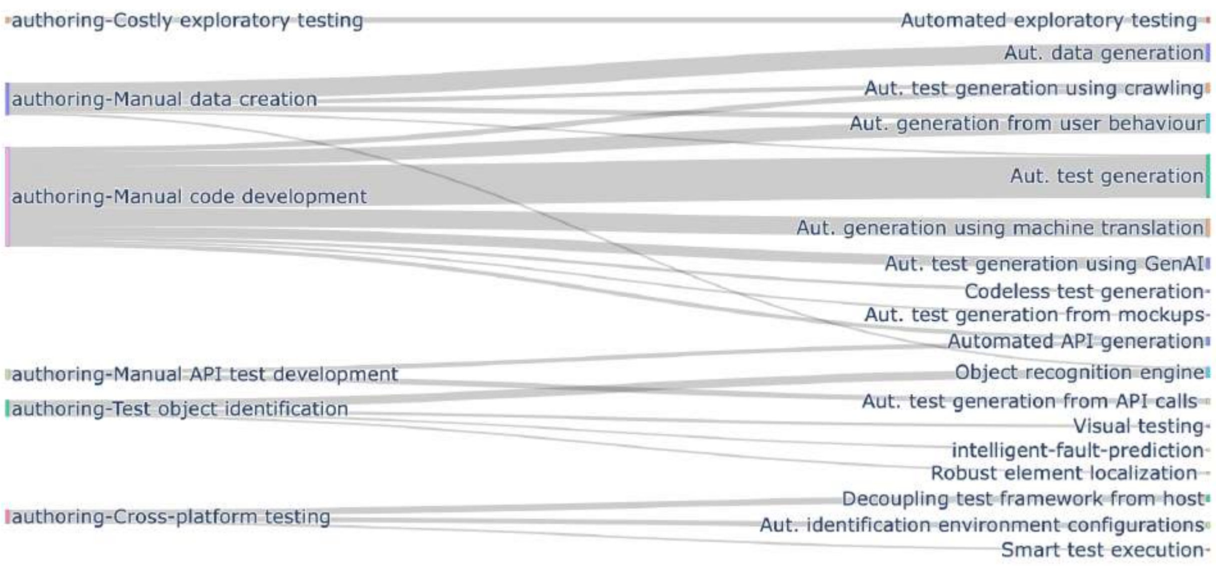

authoring-Costly exploratory testing
authoring-Manual data creation
authoring-Manual code development
authoring-Manual API test development
authoring-Test object identification
authoring-Cross-platform testing
Automated exploratory testing
Aut. data generation
Aut. test generation using crawling
Aut. generation from user behaviour
Aut. test generation
Aut. generation using machine translation
Aut. test generation using GenAI
Codeless test generation
Aut. test generation from mockups
Automated API generation
Object recognition engine
Aut. test generation from API calls
Visual testing
intelligent-fault-prediction
Robust element localization
Decoupling test framework from host
Aut. identification environment configurations
Smart test execution

Figure 5: Problems vs Solutions: Test authoring Sankey diagram.

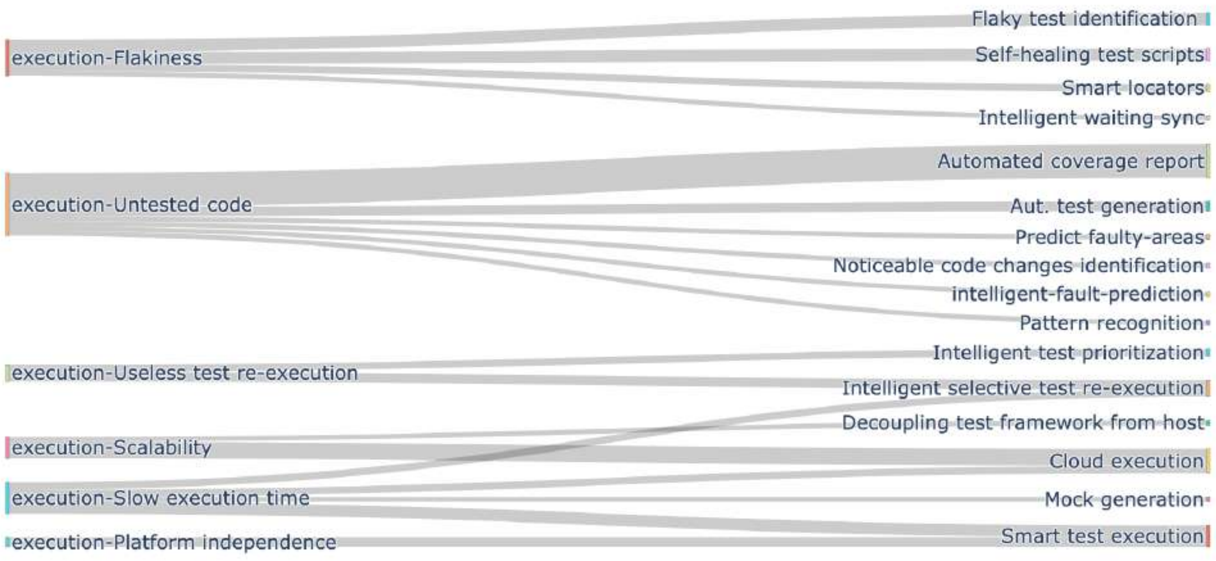

execution-Flakiness
execution-Untested code
execution-Useless test re-execution
execution-Scalability
execution-Slow execution time
execution-Platform independence
Flaky test identification
Self-healing test scripts
Smart locators
Intelligent waiting sync
Automated coverage report
Aut. test generation
Predict faulty-areas
Noticeable code changes identification
intelligent-fault-prediction
Pattern recognition
Intelligent test prioritization
Intelligent selective test re-execution
Decoupling test framework from host
Cloud execution
Mock generation
Smart test execution

Figure 6: Problems vs Solutions: Test execution Sankey diagram.

them to take compensatory actions if necessary. Another notable issue in this phase is the lengthy execution time of test suites, which can be quite significant in some cases. The grey literature presents a variety of solutions for this problem, including 'smart' test execution approaches—smart test execution scans your application for code changes and runs tests specifically to validate those changes—, runtime monitoring, and the exclusion of non-essential test cases.

Figure 7 and Figure 8 respectively depict the relevant problems and the adopted solutions for test closure and maintenance using Sankey diagrams. Regarding test closure, the most relevant problem is still the manual overhead needed to debug the code. Two strategies can be identified among the provided solutions. On the one side, solutions such as root-cause analysis and automatic identification of code changes are proposed to identify and fix the issues by considering multiple information sources. On the other

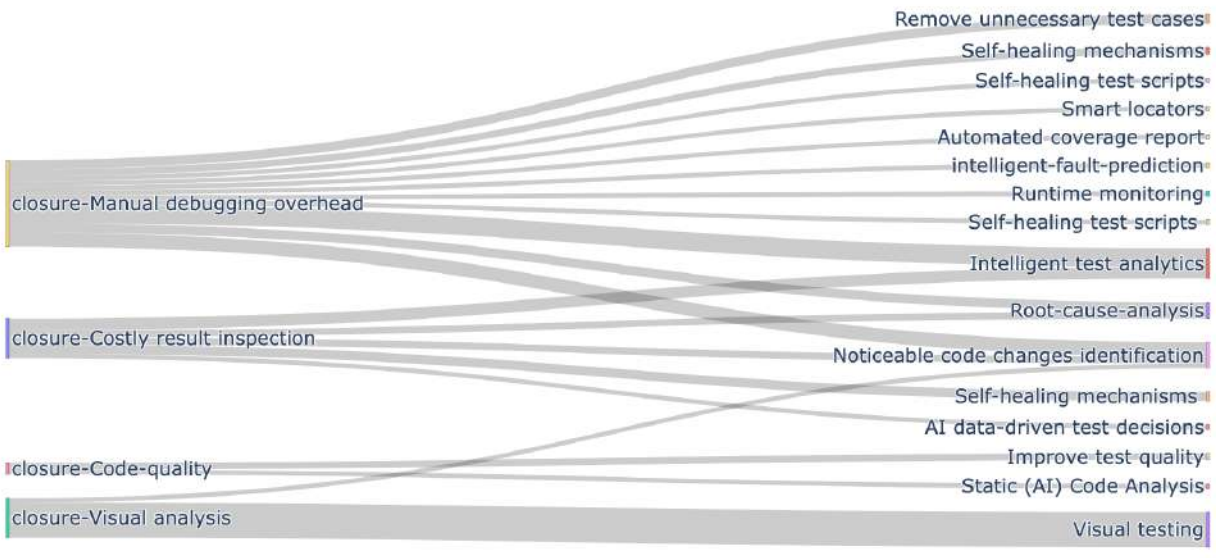

closure-Manual debugging overhead
closure-Costly result inspection
closure-Code-quality
closure-Visual analysis
Remove unnecessary test cases
Self-healing mechanisms
Self-healing test scripts
Smart locators
Automated coverage report
intelligent-fault-prediction
Runtime monitoring
Self-healing test scripts
Intelligent test analytics
Root-cause-analysis
Noticeable code changes identification
Self-healing mechanisms
AI data-driven test decisions
Improve test quality
Static (AI) Code Analysis
Visual testing

Figure 7: Problems vs Solutions: Test closure Sankey diagram.

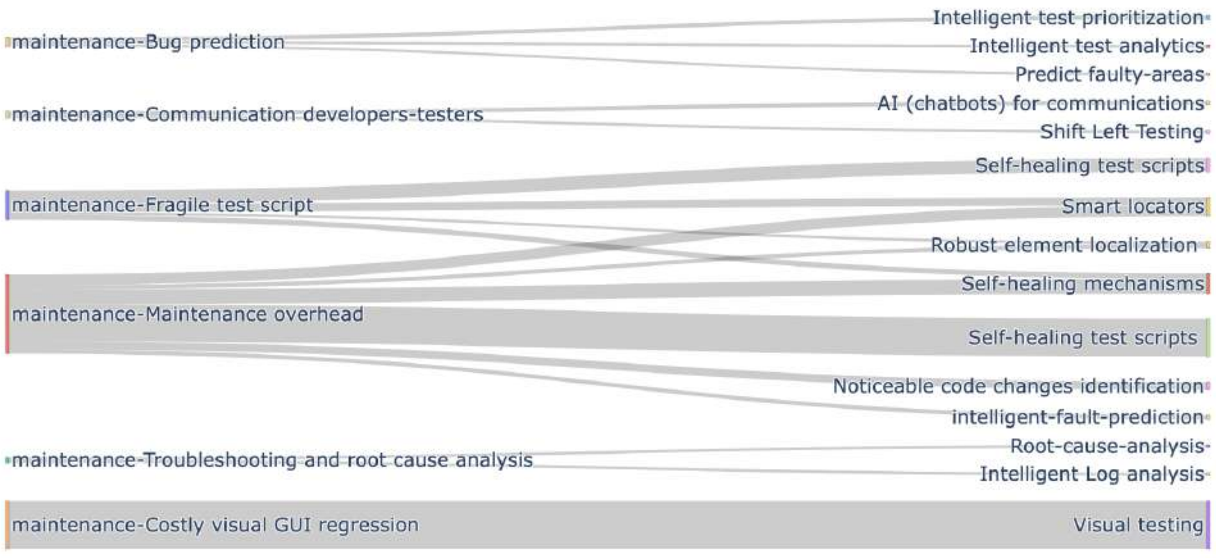

maintenance-Bug prediction
maintenance-Communication developers-testers
maintenance-Fragile test script
maintenance-Maintenance overhead
maintenance-Troubleshooting and root cause analysis
maintenance-Costly visual GUI regression
Intelligent test prioritization
Intelligent test analytics
Predict faulty-areas
AI (chatbots) for communications
Shift Left Testing
Self-healing test scripts
Smart locators
Robust element localization
Self-healing mechanisms
Self-healing test scripts
Noticeable code changes identification
intelligent-fault-prediction
Root-cause-analysis
Intelligent Log analysis
Visual testing

Figure 8: Problems vs Solutions: Test maintenance Sankey diagram.

side, self-healing and intelligent analytic mechanisms are applied to avoid and predict issues, thus reducing the manual debugging overhead. Concerning test closure, in the three iterations, we also observed an increasing interest in techniques related to failure prediction and static code analysis, aiming at improving the code and data quality. The most significant problem in test maintenance is still the overhead involved. The strategies proposed in the solutions fall into three categories: (i) the development of self-healing test scripts and mechanisms to avoid costly maintenance interventions; (ii) the application of fault prediction and code change detection techniques to better target and minimize maintenance efforts; and (iii) the implementation of smart locators and robust element localization methods to prevent test maintenance activities. Other issues affecting test maintenance are automatically validating the visual correctness of the application and the fragility of tests. A solution to the problem of automatically validating the visual correctness of

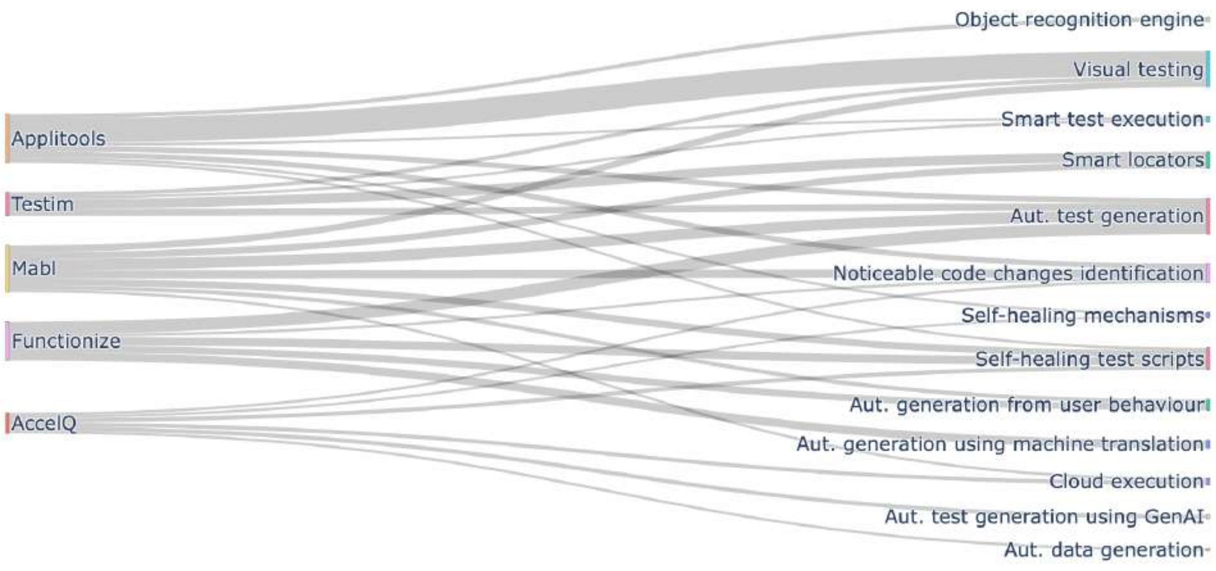

Applitools
Testim
Mabl
Functionize
AccelQ
Object recognition engine
Visual testing
Smart test execution
Smart locators
Aut. test generation
Noticeable code changes identification
Self-healing mechanisms
Self-healing test scripts
Aut. generation from user behaviour
Aut. generation using machine translation
Cloud execution
Aut. test generation using GenAI
Aut. data generation

Figure 9: Tools vs Solutions: Sankey diagram.

the application is automated visual verification of the web app GUI using computer vision techniques. Test fragility, defined as the tendency of tests to break easily due to minor changes in the application, can be addressed with various solutions, ranging from self-healing mechanisms to improving test script quality.

4.5. RQ$_5$ (Tools vs. Solutions)

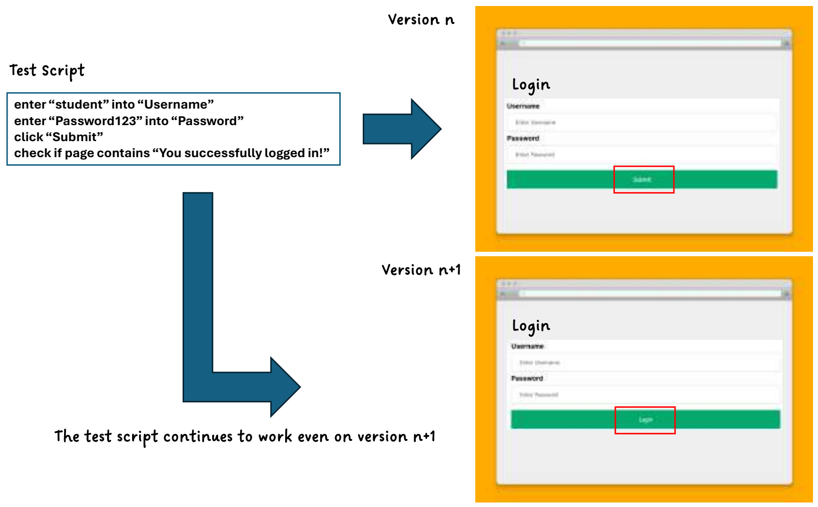
Figure 9 presents a Sankey diagram featuring the five most frequently cited tools in the third iteration—Applitools, Testim, Mabl, Functionize, and AccelQ—as listed in Table 6. Each flow in the diagram represents a relationship identified in the analyzed documents between a tool (left node in the diagram) and the AI-based solutions (right node in the diagram) it supports, as listed in Table 5. We emphasize that, in our case, the size of the flows and nodes in the diagram indicates relationships that are more comprehensively addressed in the analyzed documents. In the figure, it is evident that Functionize demonstrates notable strengths and diverse capabilities. Specifically, Functionize excels in using NLP and AI for automatic test generation and employs advanced techniques for self-healing test maintenance (see Figure 9). Similarly, Applitools provides flexibility and a comprehensive set of features. Notably, Applitools excels in visual testing and employs advanced strategies for automated test generation, including object recognition systems, user behavior observation, and generative AI mechanisms. Applitools provides also test maintenance (e.g., self-healing mechanisms, code change identification capability) and debugging features (e.g., root-cause analysis, and intelligent test analytics). In general, as we can also see from the figure, testing tools, and platforms are very flexible and support a wider range of solutions to different problems. However, there are exceptions, such as Appium and Testsigma (not present in the figure), which are more specialized and target a narrower set of capabilities.

5. Qualitative Examples: Automated Generation and Self-healing Test Scripts

In this section, we exemplify three of the most popular AI-based TA features, aiming to provide a more concrete understanding of some of the AI solutions proposed in Table 5: codeless test script generation, automated test script generation using GenAI and self-healing test scripts. We illustrate these features in the web domain using testRigor,[6] one of the most popular tools identified in the third iteration (see Table 6). We selected testRigor because it offered a 14-day trial license for full use of the tool (although limited to a single test suite). It is important to emphasize that similar features are also implemented, perhaps in different ways, in other tools identified by our taxonomy.

testRigor falls into the category of *codeless testing tools*. It enables the creation of test scripts using plain English commands, eliminating the need for traditional programming. By leveraging advanced NLP algorithms, testRigor translates these plain English instructions into executable test scripts and executes them. testRigor is capable of interpreting and translating straightforward commands in test scripts, such as "open" to launch a webpage in a browser, "click" to interact with a button, "insert" to input text into a field, and "check" to validate the presence of specific text on a webpage. For example, a simple testRigor test to verify the successful log-in on a web application could look like this:

[6]https://testrigor.com/



```
enter "student" into "Username"
enter "Password123" into "Password"
click "Submit"
check if page contains "You successfully logged in!"
```

Figure 10: Representation of the self-healing mechanism on the Login example.

```
enter "student" into "Username"
enter "Password123" into "Password"
click "Submit"
check if page contains "You successfully logged in!"
```

In addition to being codeless, the testRigor tool leverages *GenAI to automatically generate NLP-based test scripts* from informal natural language descriptions of a test case, simplifying the test development process. For instance, a description like "Test checkout process" enables the AI to generate a relevant test script description, reducing the need for manual input. Similarly, a more detailed description, such as "Develop a full test for adding a MacPro to a shopping cart", for a given e-commerce web app, allows testRigor to produce the following complete test script:

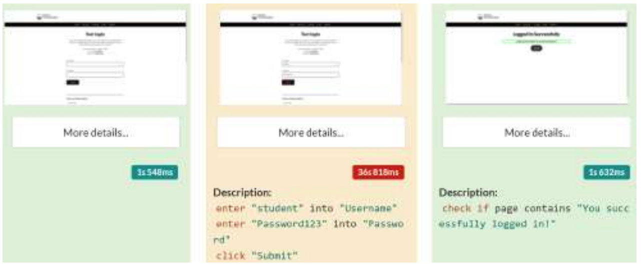
```
Click on "search bar" at the top of the page
Enter "MacBook" into "search bar"
Press Enter
Click on the first hyperlink containing "MacBook"
Click on "Add to Cart"
Click on "Go to Cart"
Check that page contains "MacBook"
Check that page contains "Quantity: 1"
```

Although this approach minimizes manual effort, its accuracy and usefulness depend on the clarity and specificity of the descriptions provided.

In addition to its generation features, testRigor also offers the ability to enable *self-healing test scripts*. Self-healing in software engineering refers to systems that automatically detect and fix errors without human intervention. Through advanced algorithms and monitoring, these systems identify issues, take corrective actions, and restore functionality, minimizing downtime and improving reliability. In test automation, self-healing scripts can adapt to changes in the application under test (AUT) by automatically correcting themselves when a test fails.

Imagine an automated test script designed to interact with a button on a web page or software interface by identifying it through a specific ID locator. This method functions seamlessly until a new version of the software alters the button's ID. As a result, the test fails, unable to locate and interact with the button. Such failures lead to maintenance challenges that require developers or testers to update the test scripts with the new locator details. This process is often tedious and resource-intensive. Self-healing test automation addresses this issue by introducing adaptability directly into the testing framework. Instead of failing outright when encountering a modified element, such as a button with a changed ID, a self-healing system actively attempts to identify the element using alternative methods, such as analyzing attributes, context, and surrounding elements or using computer vision algorithms. This capability allows the test to continue running without human intervention, minimizing disruptions and significantly reducing the effort required for test maintenance.

testRigor implements self-healing tests by leveraging a component called Vision AI to adapt to specification changes for rules and individual commands. Instead of failing, testRigor analyzes the screen to find alternative ways to achieve the intended actions. Vision AI uses computer vision to enable machines to interpret and understand visual elements of user interfaces—such as images, icons, buttons, and text—just like a human tester. This approach allows the test script to quickly adjust to changes in a web app, making it particularly effective for testing complex graphical user interfaces. To better understand the concept of self-healing test scripts and how this mechanism is implemented in testRigor, let us consider a web page containing a Login form (version n), as shown above in Figure 10. Since testRigor operates as a human tester would, we created a test script to perform the Sign-In action using the UI text as locators, as already shown above in the first test script example. Now, let us suppose that the application under test has been updated (version n+1),



Description:
```
enter "student" into "Username"
enter "Password123" into "Passwo
rd"
click "Submit"
```

Description:
```
check if page contains "You succ
essfully logged in!"
```

Figure 11: Test script execution with the self-healing mechanism enabled (represented by the box marked in orange).

and the button web element has been changed to 'Submit' (as shown below in Figure 10). A standard, non-self-healing test script would fail in this scenario because it would not locate the 'Login' text on the button. However, when executed with testRigor, which incorporates a self-healing feature, the test script successfully adapts and does not fail. As illustrated in Figure 11, the execution results show that the first step and the third step—corresponding to opening the webpage and executing the check command after clicking the Login button—are marked in green, indicating successful completion. Step two, however, is highlighted in orange, signaling that the self-healing mechanism was activated after entering the username and the password, to identify and adjust for the modified button in the updated version. Additionally, a link labeled "More details..." provides access to a detailed summary of the adjustments made. The test script execution demonstrates that when UI changes occur, testRigor's AI attempts to identify alternative ways to interact with the affected element. By clicking the "More details..." button, a popup appears displaying the original test script alongside the AI-modified version with updated test steps. If reverting to the previous version is preferred, the "Rollback to this" option can be selected. Alternatively, if the AI-generated modifications are satisfactory, they can be retained as the final version.

6. Developers Interviews

While grey literature provided valuable information for our study, the nature of these sources limits to suggestions on how to use AI for TA, without presenting concrete evidence of its practical application or usage details.

To get a more complete picture, we have conducted semi-structured interviews with researchers and practitioners with various backgrounds and levels of expertise, focusing on the types of problems and solutions found in our taxonomies.

6.1. Interviews Design

The study focused on software engineers with experience in software testing and artificial intelligence. Convenience sampling was utilized to recruit participants, allowing for efficient enrollment within the study's constraints. However, this approach inherently carries a risk of bias due to the non-random selection process. To address this limitation, we encouraged word-of-mouth promotion of the interview opportunity, leading to the recommendation of diverse and appropriate candidates. Only individuals with a minimum of five years of professional experience in software engineering were included. This criterion ensured that participants had sufficient exposure to both software testing and AI practices. Efforts were made to select participants representing a range of ages, genders, and professional roles. This approach aimed to incorporate diverse perspectives and reduce the risk of homogeneity in viewpoints. Ultimately, five developers/researchers agreed to participate in our study and conducted semi-structured interviews to understand whether they perceive AI as a valuable asset that aligns with their needs and objectives in their testing activities.

Table 7
Interview participants details.

Participant	# Position	# Expertise	# Experience
Id1	Q/A Manager	Q/A Strategy	17+ years
Id2	Senior Developer	Quality Monitoring	6+ years
Id3	Senior Developer	Data Science/LLMs	10+ years
Id4	Full Stack Developer	TA/DevOps	14+ years
Id5	Researcher	QA/Testing	20+ years

Table 7 provides details about the interviewees, including their position, expertise, and years of experience. Among the interviewed candidates the lowest value is 6 years and the highest is 20 years (median=14). Our pool includes professionals from a company in Sweden involved in strategic decision-making, as well as technical personnel and a researcher at an applied science university in Austria with substantial experience in the industry.

We conducted semi-structured interviews [61], integrating open-ended questions to gather unexpected insights and specific questions to maintain focus and assist interviewees. After obtaining background information on the interviewees' general and AI-specific experiences, we began our questioning. The first question was deliberately broad: "*What types of AI have you been using in your work?*". This approach aimed to introduce the topic openly, encouraging interviewees to discuss their experiences without steering them toward any particular problems/solutions.

We then proceeded to more specific questions covering a wide range of topics in test automation, including test creation, maintenance, execution, and the tools used. We asked interviewees if they had encountered any issues or problems related to these topics. If they responded affirmatively, we provided more detailed questions to better understand the discussed topic.

All interviews were conducted remotely via Teams video calls, each lasting approximately 60 minutes. The interviews were transcribed using Teams' automated speech recognition tool, which converts audio/video files into text. After generating the automated transcriptions, we reviewed and manually corrected transcription errors. Finally, we proceeded with open coding of the transcribed interviews in which different parts of the transcribed text were tagged. The sanitized text is available in our replication package [53].

6.2. Interviews Results

6.2.1. Problems

The usage of AI touched on several aspects of the software development process. For example, it is used for requirement elicitation during interviews with clients and customers to complement information and write summary reports.

The participants highlighted the use of AI to enhance test execution, specifically through test prioritization and AI-assisted test refactoring. On a broader picture, AI was leveraged to analyze existing codebases for predictive maintenance and risk assessment. This approach helped reduce technical debt and maintenance challenges caused by the low skill levels of consultants that exhibit high turnover.

In end-to-end testing, the interviewees mentioned that a primary challenge is testability. The lengthy and complex nature of E2E test cases makes testing at the API level difficult, especially without a clear specification. E2E tests are often fragile due to the lack of a clear API, static checking, synchronization issues, and missing types. Creating a model for these tests is considered a valuable investment, as it can result in more maintainable and robust tests, especially if the web app model creation and maintenance can also be automated.

6.2.2. Solutions

The participants emphasized the use of AI to identify dependencies between services and detect obsolete test cases due to the lack of domain knowledge in complex codebases. Another proposed solution is to implement anomaly detection using test execution logs to monitor the system and production data to help anticipate failures before they occur. An interesting case was the usage of LLMs for mutation testing: when new test cases were introduced in the test suite, the web app was modified with artificially generated faults to assess the quality of the evolved test suite.

Regarding self-healing in practice, the participants were quite critical, noting that not all issues can be "self-healed". They suggested limiting self-healing to bug classes that can be precisely described, otherwise bugs can go unnoticed. Some changes, such as updated IDs, can be managed with hash maps to fix mappings automatically or by rewriting page objects if there are changes.

Ultimately, the participants see AI as a collaborative partner in testing, code review, and pair programming. With accountability resting on the developers, they write the tests (serving as specifications), while the AI generates the code. The usage of "*AI to create test cases*" was reported to be "*in the very early stages at the moment*". It was mentioned that AI was used to create API test mockups that developers complement manual testing by adding logic.

The final message was to use AI to generate boilerplate code while keeping decision-making a human task.

6.2.3. Tools

Our participants highlighted their reliance on in-house tools and existing frameworks like Microsoft Azure DevOps, over individual tools or custom code bases that are hard to adapt to each specific customer's need. They also noted the integration of advanced generative AI models, such as OpenAI's ChatGPT and Microsoft's Copilot, into their daily activities for code generation and review.

The interviewees described employing a diverse collection of AI algorithms in their work. These range from traditional machine learning algorithms, such as Support Vector Machines (SVM) and decision trees. An interview mentioned "conventional AI" strategies such as search-based optimization algorithms, model-based approaches such as user interaction flows, or Markov Chain Monte Carlo for prioritizing tests and tracking the frequency of test changes. The participants also mentioned the usage of computer vision techniques like object recognition to find web elements that are difficult to find in the DOM, internationalization (i18n) testing, or automating the visual oracle in tests.

7. Threats to Validity

This section discusses the limitations of this research and the validity of the results presented. The primary issues related to the validity of this grey literature review involve inaccuracies in data extraction, an incomplete set of studies due to the limitations of search terms and search engines, and potential researcher bias concerning the criteria for study inclusion and exclusion. In this section, we discuss four types of validity threats plus reproducibility according to a standard validity checklist [72].

Threats to *internal validity* relate to potential biases and errors in the selection of documents (exclusion criteria) and the classification of the considered items. The classification task is particularly challenging in the context of grey literature because web documents are often informative rather than technical, and the terminology used can be vague and ambiguous. To mitigate as much as possible classification errors, we adhered to a systematic and structured procedure (Section 3.1) with multiple iterations, starting with a small pilot study for each iteration and ensuring a consistent approach. Another potential threat to the internal validity of our grey literature review is the search string used in the review process. Modifications to the search string could lead to different results, highlighting a limitation of our approach. The search string we used was carefully designed, but any changes to it could yield different sets of documents, thus affecting the scope and findings of the review. This underscores the importance of a well-defined search strategy and acknowledges that our conclusions are based on the specific search parameters we employed.

The *external validity* of our study is primarily limited by our selection of sources. We considered only documents available on Google within a specific time frame. Consequently, our findings may not be generalizable to documents from other search engines, repositories, or different time periods. Future research should expand the scope to include additional sources and broader time frames to validate and extend our findings. However, given the number of documents analyzed and considering that the analysis was conducted at different times in three separate iterations, we are quite confident in our results.

Another potential threat to validity pertains to the interviews and the open coding process. While these were conducted solely by the third author, his prior experience with similar studies and analyses [22] lends credibility to the approach. The structured interviews were limited to five participants, a constraint largely due to the difficulty of engaging developers actively engaged in their professional responsibilities. Despite the small sample size, it is important to highlight that all participants were senior developers. This ensures a solid foundation for the insights gathered, given the depth of experience and expertise they brought to the discussions. Although larger or more diverse samples could potentially lead to different conclusions, our analysis of the open-ended responses revealed recurring themes and no emergence of new ones, suggesting that saturation was reached. Consequently, we believe that, despite the limitations, the sample size is adequate for this study. The use of convenience sampling could lead to a lack of representativeness in the sample, potentially skewing findings. To mitigate this, efforts were made to diversify the sample as much as possible within the method's constraints. However, it is acknowledged that the findings may not be fully generalizable to the broader population of software engineers.

Construct validity concerns the extent to which our methodology accurately captures the constructs of interest, such as the identification of AI-based solutions for test automation and their associated issues. The inherent vagueness and ambiguity in grey literature terminology pose a risk to construct validity. To address this, we defined clear criteria for inclusion and exclusion and employed a rigorous procedure with reliability checks through an initial pilot study conducted at each iteration.

Conclusion validity pertains to the degree to which our conclusions are credible and dependable. The iterative nature of our experiment, along with a pilot study conducted at each iteration, helps to ensure that our conclusions are based on systematically derived evidence. Nonetheless, the subjective nature of document interpretation in grey literature reviews remains a potential threat.

To enhance *reproducibility*, we have made all our results, including data, plots, and references, available in our replication package [53]. This transparency allows other researchers to verify our findings and procedure. However, the inherent variability in grey literature sources means that exact replication may be challenging. We encourage future researchers to apply our procedure in different contexts to assess its robustness and adaptability.

8. Discussion

8.1. Observations from the Survey

8.1.1. Generalizability

In the first iteration, we analyzed 156 documents (see Table 1), in the second one we additionally considered 95 documents (+60.8% to the initial set of 156 documents of [55]), and in this third iteration, we additionally considered 91 documents (+36.2% concerning the total number of documents, that is, 251 found in the first two iterations). Concerning the two initial taxonomies, in detail, we have only partially extended the taxonomy related to TA problems, keeping it largely unchanged from the original version [55]. In fact, during the second iteration, we only introduced 5 new subcategories of TA problems, resulting in a total of 44 individual occurrences out of 545. In the third iteration, we added 7 additional subcategories, which accounted for 25 individual occurrences out of 917. We also made only partial progress in extending the taxonomy related to AI-based solutions. Specifically, during the second iteration, we introduced one new category (i.e., Test Execution) and 9 subcategories, which accounted for 80 individual occurrences out of 607. In the third iteration, we added 3 additional categories (i.e., Test Optimization, Test Process, and Test Type) along with 15 subcategories, resulting in 77 individual occurrences out of 972. In conclusion, regarding the generalizability of the taxonomies, we can state that in our analysis, the list of problems and solutions has remained largely consistent with the previous versions, with only a few minor additions. On the other hand, the list of tools has shown significant and consistent growth across the iterations, doubling their number from the first to the third iteration.

8.1.2. Prevalence of Web E2E Testing

Our search query was designed to encompass software testing broadly, without focusing on any specific type of system. Nevertheless, our survey of the grey literature revealed that most articles centered on E2E testing for mobile and web applications, followed by unit and API testing.

8.1.3. AI for Innovative Test Authoring

We identified TA problems for which AI-enhanced existing solutions seem to be promising and largely studied. In particular, we observed that test authoring (i.e., test creation) is the most investigated TA problem. We identified 34 tools (e.g, Applitools, Functionize, Testim, Testrigor) that use AI-based technology for automatically generating test cases using different technologies, such as NLP techniques or GUI-based capture and reply techniques to generate test scripts, aiming at limiting the manual human intervention. Research approaches have been proposed to this aim [4, 12]. Practitioners can use our taxonomy to easily identify the problems better supported by existing tools whereas researchers can identify unexplored problems for which they can propose innovative solutions.

8.1.4. AI-assisted Visual Oracle

We identified TA problems that could be faced by AI-enhanced existing solutions implemented in AI-based tools. For instance, a well-known challenge in GUI testing is the creation of effective oracles [43, 24, 25]. The use of oracles based on visual testing, using AI and computer vision approaches, is suggested in the grey literature as one of the possible ways to face this challenge. This aligns with recent research visual approach to web testing [66, 71]. Furthermore, 13 tools that support oracle visual testing have been identified, e.g., Applitools, AI Testbot, Mabl, Sealights, Testim, and Test.ai. Practitioners can use our findings to quickly select the most appropriate solutions for their TA problems and the most adequate tools that support the solutions to their TA problems.

8.1.5. Phase-specific and Multi-Phase Solutions for Smarter Test Automation

In general, we identified tools that support solutions across various phases of test automation and address multiple test automation problems. However, there are also, though less common, tools that are *specific* to a particular phase or problem. For instance, tools such as Applitools, Functionize, and Mabl can support different TA phases, e.g., test creation, maintenance, and execution. Conversely, tools such as Appium, Testsigma, and TestComplete seem to be more specific, thus mainly supporting a given testing phase, e.g., Appium and Testsigma focus on test creation, while TestComplete focuses on visual testing. Our findings are of interest to practitioners in selecting the most appropriate tools to use in their business, by taking into account the problems they have to face and also other aspects such as the specificity and flexibility of tools.

We also identified TA problems that require solutions involving *several phases* of TA. For instance, in the test planning phase, we defined problems such as planning what to test, identifying critical paths in the application under test, and managing the entire test process, among others. We observe that the latter problem raised attention only recently (in the third iteration of our study), and it is faced by mainly applying AI data-driven decision strategies [14]. The second problem is mainly faced with automatic test creation solutions [3], ranging from crawling the application under test [4] to the creation of test scripts using user behaviors [12]. The first problem instead is faced by solutions involving test script creation (e.g., automatic test creations by focusing on application areas that are predicted as more buggy), test selection (e.g., based on fault prediction), test execution (e.g., adoption of intelligent test re-execution strategies), and, finally, test debugging approaches (e.g., adoption of test analytics). Researchers can use our findings to better highlight as some TA problems are addressed from different perspectives, i.e., for some problems, specific ad-hoc solutions can be adequate while, for other problems, more complex solutions need to be studied.

8.1.6. Open Challenges for AIaTA

We identified solutions presented in the grey literature that are not supported by existing available tools. For instance, nevertheless, test generation with mockups and dynamic properties identified from observed user behaviors are listed among possible solutions supported by AI, for automatically generating tests, we did not identify any existing tools that provided such capabilities. Non-AI research tooling has been proposed in literature [10], but AI-driven solutions remain limited, with only a few proposals, such as those leveraging deep learning [59], introduced to date. However, it is important to keep in mind that the absence of a tool does not necessarily indicate a lack of existing solutions for a particular problem. It could simply mean that the specific tooling solutions were not mentioned in the literature due to the incompleteness of our analysis. For instance, decoupling the test framework from the host environment is referenced, in the grey literature, as one possible solution for facilitating cross-platform testing.

It seems that this solution is not adequately supported by the existing AI-enhanced tools, that mainly provide approaches that allow the identification of different environmental configurations, to face cross-platform testing. Conversely, this issue has been addressed in the white literature for web [77] and mobile domains [77] with approaches based on robotic arms [75], deep reinforcement learning [79], and computer vision to enable the Chouette Crawler to help accelerate the mobile and web app testing cycle at Duolingo [73].

Another solution that seems to be not adequately supported by tools concerns the execution of test cases with mock responses: no tools support the construction of mock objects that can be used in TA. Concerning the test selection and optimization, solutions aiming at prioritizing test cases, removing unnecessary test cases and GUI-based testing seem to be not adequately supported by existing AI-enhanced tools. Our findings are of interest to both professionals and researchers to develop innovative tools and technologies capable of supporting the identified solutions.

8.2. Observations from the Interviews

We questioned whether AI is merely a more sophisticated form of traditional automation or a transformative technology that is here to stay. The consensus was largely positive, particularly regarding large language models. However, these tools require strict guidelines for junior developers, who might overly depend on their outputs without proper verification. Indeed, LLMs are prone to hallucinations, where the output appears plausible but is largely incorrect. For test data generation, the creativity aspect of large language models can create interesting and useful test data in situations where optimization methods like fuzzing are not applicable. However, a clear limitation is AI's lack of domain knowledge. While AI can access vast amounts of data, it lacks specific domain or company-specific knowledge, making the pre-trained models insufficient.

As such, our participants emphasized developing self-check mechanisms and best practices for using AI in TA. LLMs often attempt to create solutions rather than acknowledging uncertainty. LLMs have a user-pleasing tendency, finding workarounds instead of reporting errors (obviating the oracle problem). This can lead to excessively positive results, while testers typically adopt a more cautious, critical, and pessimistic perspective. Our interviews revealed that quality is not just about coverage; relevance and better assertions are crucial to avoid potential risks in production.

In practice, the testing team creates test suites exhibiting "*a reasonable coverage for their needs and for their criticality*".

Other research areas that deserve attention include human-computer interaction, focusing on ensuring that the results of AI are clear and explainable. Additionally, developing quality metrics to assess the output of AI models is essential, such as measuring the frequency of hallucinations accurately or evaluating the quality of the output they produce.

9. Related Work

This section of related works is divided into three subsections: 1) works where AI and ML are applied to TA, 2) analyses of grey literature in the context of test automation and multivocal literature reviews, and 3) secondary studies focused on AIaTA.

9.1. Artificial Intelligence in Test Automation

In the software testing community, AI/ML solutions are increasingly adopted to automate various testing activities (e.g., test data generation) and address issues such as test suite maintenance and test case prioritization, ushering in a new era of smarter and more efficient QA processes. Test generation is a significant area where AI/ML has been explored. Zhang et al. [78] and Walia et al. [69] propose, using computer vision, approaches to automate GUI test generation, aiming to reduce human effort. Qian et al. [50] adopt an OCR-based technique to localize GUI elements for test generation. Test maintenance, which traditionally requires significant human effort (e.g., for page object generation [64]), can also benefit from AI/ML. Computer vision approaches [2] have been widely used for web test migration [37, 35, 63] and test repair. Code-less functional test automation is investigated by Vos et al. [68] and Phuc Nguyen et al. [48] for test maintenance. The latter paper combines Selenium and ML techniques to reduce the time testers spend modifying test code. Other testing issues addressed by AI/ML include ML-based detection of flaky tests (Camara et al. [8]), cross-browser incompatibility detection (Mahajan et al. [40]), test case prioritization (Feng et al. [13]), and identifying areas of the application under test for increased test coverage such as the work by Yadav et al. [76]. Unlike these works, our study does not focus on a specific AI/ML technique for test automation. Instead, we conduct a multi-year grey literature review to capture the state-of-the-art concerning test automation problems, proposed solutions, and existing tools.

9.2. Grey Literature and Multivocal Literature Reviews on Test Automation

One of the early works in grey literature analysis on test automation is that of Päivi Raulamo-Jurvanen and colleagues [52]. In this paper, the researchers investigate how practitioners address the challenge of selecting the appropriate test automation tool. Their methodology involves consolidating insights from practitioners through a review of grey literature sourced from 53 distinct companies. The findings reveal a shared understanding of important selection criteria, albeit with inconsistent application. To address this, the authors distill insights from various sources into a cohesive 12-step process and identify 14 distinct criteria for effective tool selection. The study indicates that practitioners generally exhibit a keen interest in and are influenced by related grey literature, as evidenced by the substantial number of backlinks to the sources. Despite the abundance of available software testing tools, practitioners tend to gravitate towards well-known and widely adopted options (e.g., Selenium, QTP/UFT, and TestComple). This work, although its objective differs from ours, shares similarities.

Another work falling into this category is that of Yuqing Wang et. al. [70]. In their paper, the authors present a multivocal literature review aimed at surveying and synthesizing guidelines from existing literature on enhancing test automation maturity. A multivocal literature review is a type of systematic literature review that includes both academic literature and grey literature. They conducted a review of 81 primary studies. From these studies, they extracted 26 test automation best practices and collected various pieces of advice on how to implement them effectively. The main observations include: (1) Only 6 best practices have been empirically evaluated for their positive impact on maturity improvement; (2) Some technical best practices identified in the review were not previously included in test maturity models; (3) Certain best practices correlate with success factors and maturity impediments identified by other researchers; (4) Many pieces of advice on implementing best practices are derived from experience studies and require further empirical evaluation using formal methods; (5) Some advice on implementing certain best practices conflicts in the literature. The objective of this work is completely different from ours, and also the research method is different (multivocal literature review vs. grey literature analysis).

The last work we have selected in this category is that of Garousi and Mäntylä [17]. The study investigates decision-making in software test automation within the context of software development and tries to answer the question: when and what to automate in software testing? While many organizations view test automation as a means to cut costs and expedite development, its effectiveness depends on various factors like timing, context, and approach. To address this, the researchers conducted a multivocal literature review of 78 sources, including formal and grey literature. They categorized factors influencing automation decisions into five groups and identified prevalent ones such as regression testing needs, economic factors, and SUT maturity. The study concludes that current decision-support in software test automation offers reasonable advice for the industry. As a practical outcome, the findings have been synthesized into a checklist for practitioners. However, there is a recommendation to develop systematic, empirically validated decision-support approaches, as existing advice often lacks systemization and is based on weak empirical evidence. Our work differs from this one in terms of goals and research method, but we share with it the procedure of grey literature analysis, which we took inspiration from for our study.

9.3. Secondary Studies on AI in Test Automation

In the literature, there are numerous reviews and surveys focusing on Test Automation via AI/ML. For instance, Trudova et al. [67] conducted a systematic literature review to explore the role of AI/ML in TA. Their findings based on 34 primary studies on AI implementation in software testing reveal that most studies in the literature investigate the application of ML and computer vision techniques to reduce manual intervention in software testing and enhance the effectiveness and reusability of test suites. In particular, the activities which could be improved by the adoption of AI techniques are as follows: test case generation, test oracle generation, test execution, test data generation, test results reporting, test repair, test case selection, flaky test prediction, and test order generation. Also in their case, as in our analysis, the analyzed papers mainly addressed test case generation.

The study conducted by Lima et al. [38] presents a concise overview of the current state of software testing, with a specific focus on the integration of ML and AI algorithms. It evaluates the progress made in AI and ML techniques for software testing over the last three years, drawing from databases including Scopus Elsevier, Web of Science, and Google Scholar. The algorithms are classified into white-box, grey-box, and black-box testing, with an examination of their respective application domains. The authors conclude that black-box testing emerges as the predominant approach in software testing involving AI. Furthermore, within black-box testing, all three ML methods—supervised, unsupervised, and reinforcement learning—are commonly utilized. Notably, techniques such as clustering, Artificial Neural Networks, and Genetic Algorithms find extensive use in tasks such as fuzzing and regression testing. Differently from this study, which aims to infer the AI and ML algorithms used in software testing, we focused on other aspects as well because grey literature is more descriptive and often lacks technical details.

Our work falls within the scope of these secondary studies. Unlike previous works, we focused on the grey literature to gain insights into practitioners' perceptions of AI/ML adoption in TA. Moreover, we corroborated our taxonomies with interviews with industrial professionals.

10. Conclusions and Future Work

Artificial intelligence proposes to revolutionize the way we develop and test software systems. Novel tools and testing platforms 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, what tools are available, and for what scope.

To fill this gap, in this paper, we present a multi-year study of the grey literature concerning AI solutions for test automation. We manually analyzed several thousands 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 the most popular tools available. The insights provided by our taxonomies were corroborated by five interviews with industrial practitioners, who also provided further insights about the usage of AI in TA.

Future research directions consist of conducting a multivocal literature review by integrating the findings gathered from the grey literature with those of the white literature. It would be also interesting to conduct controlled experiments with existing AI-enhanced tools, to quantify the benefits they provide and to validate the observed connections with the TA problems and the investigated solutions.

11. Acknowledgments

This research was funded by the Bavarian Ministry of Economic Affairs, Regional Development and Energy. We extend our deepest gratitude to the professionals who generously took the time to participate in our interviews.

References

[1] Amalfitano, D., Faralli, S., Hauck, J.C.R., Matalonga, S., Distante, D., 2023. Artificial intelligence applied to software testing: A tertiary study. ACM Comput. Surv. 56. URL: https://doi.org/10.1145/3616372, doi:10.1145/3616372.

[2] Bajammal, M., Stocco, A., Mazinanian, D., Mesbah, A., 2020. A Survey on the Use of Computer Vision to Improve Software Engineering Tasks. TSE .

[3] Biagiola, M., Stocco, A., Ricca, F., Tonella, P., 2019. Diversity-based web test generation, in: Proc. of 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM. p. 12 pages.

[4] Biagiola, M., Stocco, A., Ricca, F., Tonella, P., 2020. Dependency-aware web test generation, in: Proc. of 13th IEEE International Conference on Software Testing, Verification and Validation, IEEE. p. 12 pages.

[5] Bishop, C.M., 2006. Pattern Recognition and Machine Learning (Information Science and Statistics). Springer-Verlag, Berlin, Heidelberg.

[6] Borjesson, E., Feldt, R., 2012. Automated system testing using visual gui testing tools: A comparative study in industry, in: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, pp. 350–359. doi:10.1109/ICST.2012.115.

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

[8] Camara, B., Silva, M., Endo, A., Vergilio, S., 2021. On the use of test smells for prediction of flaky tests, Association for Computing Machinery. p. 46–54.

[9] Contan, A., Dehelean, C., Miclea, L., 2018. Test automation pyramid from theory to practice, in: 2018 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR), pp. 1–5. doi:10.1109/AQTR.2018.8402699.

[10] D'Souza, C., Deufemia, V., Ginige, A., Polese, G., 2018. Enabling the generation of web applications from mockups. Software: Practice and Experience 48, 945–973.

[11] Durelli, V.H.S., Durelli, R.S., Borges, S.S., Endo, A.T., Eler, M.M., Dias, D.R.C., Guimarães, M.P., 2019. Machine learning applied to software testing: A systematic mapping study. IEEE Transactions on Reliability 68, 1189–1212. doi:10.1109/TR.2019.2892517.

[12] Erdem, I., Oguz, R.F., Olmezogullari, E., Aktas, M.S., 2021. Test script generation based on hidden markov models learning from user browsing behaviors, in: 2021 IEEE International Conference on Big

Data (Big Data), pp. 2998–3005. doi:10.1109/BigData52589.2021.9671312.

[13] Feng, Y., Jones, J.A., Chen, Z., Fang, C., 2016. Multi-objective test report prioritization using image understanding, in: Proceedings of 31st IEEE/ACM International Conference on Automated Software Engineering, ACM, New York, NY, USA. pp. 202–213.

[14] Gao, J., Tao, C., Jie, D., Lu, S., 2019. Invited paper: What is ai software testing? and why, in: 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE), pp. 27–2709. doi:10.1109/SOSE.2019.00015.

[15] Garousi, V., Elberzhager, F., 2017. Test automation: Not just for test execution. IEEE Software 34, 90–96. doi:10.1109/MS.2017.34.

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

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

[18] Graham, D., Fewster, M., 2012. Experiences of Test Automation: Case Studies of Software Test Automation. 1st ed., Addison-Wesley Professional.

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

[20] Harman, M., 2012. The role of artificial intelligence in software engineering, in: 2012 First International Workshop on Realizing AI Synergies in Software Engineering (RAISE), pp. 1–6. doi:10.1109/RAISE.2012.6227961.

[21] Hourani, H., Hammad, A., Lafi, M., 2019. The impact of artificial intelligence on software testing, in: 2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT), pp. 565–570. doi:10.1109/JEEIT.2019.8717439.

[22] Humbatova, N., Jahangirova, G., Bavota, G., Riccio, V., Stocco, A., Tonella, P., 2020. Taxonomy of real faults in deep learning systems, in: Proceedings of the ACM/IEEE 42nd international conference on software engineering, pp. 1110–1121.

[23] Inc., P.T., 2015. Collaborative data science. URL: https://plot.ly.

[24] Jahangirova, G., Clark, D., Harman, M., Tonella, P., 2016. Test oracle assessment and improvement, in: Proceedings of the 25th International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA. p. 247–258. URL: https://doi.org/10.1145/2931037.2931062, doi:10.1145/2931037.2931062.

[25] Jahangirova, G., Clark, D., Harman, M., Tonella, P., 2021. An empirical validation of oracle improvement. IEEE Transactions on Software Engineering 47, 1708–1728. doi:10.1109/TSE.2019.2934409.

[26] Jha, N., Popli, R., 2021. Artificial intelligence for software testing-perspectives and practices, in: CCICT '21, pp. 377–382.

[27] Kim, M., Xin, Q., Sinha, S., Orso, A., 2022. Automated test generation for rest apis: no time to rest yet, in: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA. p. 289–301. URL: https://doi.org/10.1145/3533767.3534401, doi:10.1145/3533767.3534401.

[28] Krichen, M., 2023. How artificial intelligence can revolutionize software testing techniques, in: Abraham, A., Bajaj, A., Gandhi, N., Madureira, A.M., Kahraman, C. (Eds.), Innovations in Bio-Inspired Computing and Applications, Springer Nature Switzerland, Cham. pp. 189–198.

[29] Leger, G., Barragan, M.J., 2018. Mixed-signal test automation: Are we there yet?, in: 2018 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 1–5.

[30] Leinen, F., Elsner, D., Pretschner, A., Stahlbauer, A., Sailer, M., Jürgens, E., 2024. Cost of flaky tests in continuous integration: An industrial case study, in: 2024 IEEE Conference on Software Testing, Verification and Validation (ICST).

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

[32] Leotta, M., Clerissi, D., Ricca, F., Tonella, P., 2016a. Approaches and tools for automated end-to-end web testing. Advances in Computers 101, 193–237.

[33] Leotta, M., Garcia, B., Ricca, F., Whitehead, J., 2023a. Challenges of end-to-end testing with selenium webdriver and how to face them: A survey, in: Proceedings of the 16th IEEE International Conference on Software Testing, Verification and Validation, pp. 339–350.

[34] Leotta, M., Ricca, F., Marchetto, A., Olianas, D., 2023b. An empirical study to compare three web test automation approaches: Nlp-based, programmable, and capture&replay. J. Softw. Evol. Process 36. URL: https://doi.org/10.1002/smr.2606, doi:10.1002/smr.2606.

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

[36] Leotta, M., Stocco, A., Ricca, F., Tonella, P., 2016b. ROBULA+: An algorithm for generating robust XPath locators for web testing. Journal of Software: Evolution and Process , 28:177–204.

[37] Leotta, M., Stocco, A., Ricca, F., Tonella, P., 2018. PESTO: Automated migration of DOM-based web tests towards the visual approach. Software Testing, Verification And Reliability 28.

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

[39] Luo, Q., Hariri, F., Eloussi, L., Marinov, D., 2014. An empirical analysis of flaky tests, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA. p. 643–653.

[40] Mahajan, S., Halfond, W.G.J., 2015. 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, pp. 1–10.

[41] Massol, V., Husted, T., 2003. JUnit in Action. Manning Publications Co., USA.

[42] McKinney, W., et al., 2010. Data structures for statistical computing in python, in: Proceedings of the 9th Python in Science Conference, Austin, TX. pp. 51–56.

[43] Memon, A., Banerjee, I., Nagarajan, A., 2003. What test oracle should i use for effective gui testing?, in: 18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings., pp. 164–173.

[44] Mesbah, A., Prasad, M.R., 2011. Automated cross-browser compatibility testing, in: Proceedings of the 33rd International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA. p. 561–570. URL: https://doi.org/10.1145/1985793.1985870, doi:10.1145/1985793.1985870.

[45] Myers, G.J., Sandler, C., Badgett, T., 2011. The Art of Software Testing. 3rd ed., Wiley Publishing.

[46] Nass, M., Alégroth, E., Feldt, R., 2021. Why many challenges with gui test automation (will) remain. Information and Software Technology 138, 106625. URL: https://www.sciencedirect.com/science/article/pii/S0950584921000963, doi:https://doi.org/10.1016/j.infsof.2021.106625.

[47] Pham, P., Nguyen, V., Nguyen, T., 2023. A review of ai-augmented end-to-end test automation tools, in: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, Association for Computing Machinery, New York, NY, USA. URL: https://doi.org/10.1145/3551349.3563240, doi:10.1145/3551349.3563240.

[48] Phuc Nguyen, D., Maag, S., 2020. Codeless web testing using Selenium and machine learning, in: ICSOFT 2020: 15th International Conference on Software Technologies, ScitePress, Online, France. pp. 51–60.

[49] Polo, M., Reales, P., Piattini, M., Ebert, C., 2013. Test automation. IEEE Software 30, 84–89. doi:10.1109/MS.2013.15.

[50] Qian, J., Ma, Y., Lin, C., Chen, L., 2023. Accelerating OCR-Based Widget Localization for Test Automation of GUI Applications. Association for Computing Machinery.

[51] Rafi, D.M., Moses, K.R.K., Petersen, K., Mäntylä, M., 2012. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. 2012 7th International Workshop on Automation of Software Test (AST) , 36–42.

[52] Raulamo-Jurvanen, P., Mäntylä, M., Garousi, V., 2017. 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, ACM. p. 21–30.

[53] rep-pack, 2024. Replication Package. https://github.com/riccaF/grey-literature-review-ai-test-automation.

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

[55] Ricca, F., Marchetto, A., Stocco, A., 2021. AI-based Test Automation: A Grey Literature Analysis, in: Proceedings of 14th IEEE International Conference on Software Testing, Verification and Validation Workshops, Springer.

[56] Ricca, F., Marchetto, A., Stocco, A., 2023. A retrospective analysis of grey literature for ai-supported test automation, in: Fernandes, J.M., Travassos, G.H., Lenarduzzi, V., Li, X. (Eds.), Quality of Information and Communications Technology, Springer Nature Switzerland, Cham. pp. 90–105.

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

[58] Riccio, V., Jahangirova, G., Stocco, A., Humbatova, N., Weiss, M., Tonella, P., 2020. Testing machine learning based systems: a systematic mapping. Empirical Software Engineering 25.

[59] Samir, M., Elsayed, A., Marie, M.I., 2024. A model for automatic code generation from high fidelity graphical user interface mockups using deep learning techniques. International Journal of Advanced Computer Science & Applications 15.

[60] Schmidt, M., 2008. The sankey diagram in energy and material flow management. Journal of Industrial Ecology 12, 173–185.

[61] Seaman, C., 1999. Qualitative methods in empirical studies of software engineering. IEEE Transactions on Software Engineering 25, 557–572. doi:10.1109/32.799955.

[62] Selenium, 2020. SeleniumHQ web browser automation. https://www.selenium.dev/. Accessed: 2020-07-22.

[63] Stocco, A., Leotta, M., Ricca, F., Tonella, P., 2014. PESTO: A tool for migrating DOM-based to visual web tests, in: Proceedings of 14th International Working Conference on Source Code Analysis and Manipulation, IEEE Computer Society. pp. 65–70.

[64] Stocco, A., Leotta, M., Ricca, F., Tonella, P., 2017. APOGEN: Automatic Page Object Generator for Web Testing. Software Quality Journal 25, 1007–1039.

[65] Stocco, A., Yandrapally, R., Mesbah, A., 2018a. Visual web test repair, in: Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM.

[66] Stocco, A., Yandrapally, R., Mesbah, A., 2018b. 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), p. 12 pages.

[67] Trudova., A., Dolezel., M., Buchalcevova., A., 2020. Artificial intelligence in software test automation: A systematic literature review, in: Proceedings of the 15th Int. Conf. on Evaluation of Novel Approaches to Software Engineering - Vol. 1: ENASE,, INSTICC. SciTePress. pp. 181–192. doi:10.5220/0009417801810192.

[68] Vos, T.E.J., Aho, P., Pastor Ricos, F., Rodriguez-Valdes, O., Mulders, A., 2021. Testar – scriptless testing through graphical user interface. Software Testing, Verification and Reliability 31, e1771.

[69] Walia, R., 2022. Application of machine learning for gui test automation, in: 2022 XXVIII International Conference on Information, Communication and Automation Technologies (ICAT), pp. 1–6.

[70] Wang, Y., Mäntylä, M.V., Liu, Z., Markkula, J., Raulamo-jurvanen, P., 2022. Improving test automation maturity: A multivocal literature review. Software Testing, Verification and Reliability 32. URL: http://dx.doi.org/10.1002/stvr.1804, doi:10.1002/stvr.1804.

[71] Wen, Z., Lu, Y., Xu, T., Pan, M., Zhang, T., Li, X., 2024. Enhancing web test script repair using integrated ui structural and visual information, in: 2024 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 75–86. doi:10.1109/ICSME58944.2024.00018.

[72] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., Wesslén, A., 2000. Experimentation in Software Engineering - An Introduction. Kluwer Academic Publishers.

[73] Wong, T., 2023. Chouette: An automated cross-platform ui crawler for improving app quality, in: 2023 38th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW), pp. 175–178. doi:10.1109/ASEW60602.2023.00029.

[74] Wong, W., Horgan, J., London, S., Agrawal, H., 1997. A study of effective regression testing in practice, in: Proceedings The Eighth International Symposium on Software Reliability Engineering, pp. 264–274. doi:10.1109/ISSRE.1997.630875.

[75] Xie, M., Ye, J., Xing, Z., Ma, L., 2023. Nicro: Purely vision-based, non-intrusive cross-device and cross-platform gui testing. URL: https://arxiv.org/abs/2305.14611, arXiv:2305.14611.

[76] Yadav, V., Botchway, R.K., Senkerik, R., Kominkova, Z.O., 2021. Robot testing from a machine learning perspective, in: 2021 International Conference on Electrical, Computer and Energy Technologies (ICECET), pp. 1–4.

[77] Yu, S., Fang, C., Yun, Y., Feng, Y., 2021. Layout and image recognition driving cross-platform automated mobile testing, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), IEEE. pp. 1561–1571.

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

[79] Zhang, S., Wu, L., Li, Y., Zhang, Z., Lei, H., Li, D., Guo, Y., Chen, X., 2023. Resplay: Improving cross-platform record-and-replay with gui sequence matching, in: 2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE), pp. 439–450. doi:10.1109/ISSRE59848.2023.00056.