Practitioners' Expectations on Automated Test Generation

Xiao Yu
Huawei
Hangzhou, China
yuxiao25@huawei.com

Lei Liu
Faculty of Electronic and Information Engineering, Xi'an Jiaotong University
Xi'an, China
Lei.Liu@stu.xjtu.edu.cn

Xing Hu*
The State Key Laboratory of Blockchain and Data Security, Zhejiang University
Hangzhou, China
xinghu@zju.edu.cn

Jacky Keung
Department of Computer Science, City University of Hong Kong
Hong Kong, China
jacky.keung@cityu.edu.hk

Xin Xia
Huawei
Hangzhou, China
xin.xia@acm.org

David Lo
Singapore Management University
Singapore
davidlo@smu.edu.sg

Abstract

Automated test generation can help developers craft high-quality software tests while mitigating the manual effort needed for writing test code. Despite significant research efforts in automated test generation for nearly 50 years, there is a lack of clarity about what practitioners expect from automated test generation tools and whether the existing research meets their needs. To address this issue, we follow a mixed-methods approach to gain insights into practitioners' expectations of automated test generation. We first conduct the qualitative analysis from semi-structured interviews with 13 professionals, followed by a quantitative survey of 339 practitioners from 46 countries across five continents. We then conduct a literature review of premier venue papers from 2022 to 2024 (in the last three years) and compare current research findings with practitioners' expectations. From this comparison, we outline future research directions for researchers to bridge the gap between automated test generation research and practitioners' expectations.

CCS Concepts

• **Software and its engineering → Software maintenance tools**.

Keywords

Test Generation, Empirical Study, Practitioners' Expectations

ACM Reference Format:
Xiao Yu, Lei Liu, Xing Hu, Jacky Keung, Xin Xia, and David Lo. 2024. Practitioners' Expectations on Automated Test Generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680386

1 Introduction

Crafting high-quality software tests manually is challenging and time-consuming for developers [46, 71]. To alleviate the issue, researchers have proposed numerous automated test generation techniques or tools for nearly 50 years [55]. However, no prior studies have investigated practitioners' expectations of these techniques or tools. There is a lack of clarity on whether practitioners appreciate the current automated test generation techniques or tools, what factors influence their decisions to adopt them, and what their minimum thresholds for adoption are. Gaining insights from practitioners is essential to uncover critical issues and guide researchers in developing solutions that meet the needs of practitioners.

In this paper, we follow a mixed-methods approach to gain insights into practitioners' expectations of automated test generation. We begin with semi-structured interviews with 13 professionals with an average of 6.85 years of software programming experience. Through these interviews, we explore whether the interviewees rely on requirement descriptions or the code under test when crafting testing code in practical software development, the state of automated test generation practices, issues faced by the interviewees when using automated test generation tools, and their expectations of automated test generation. We then conduct an exploratory survey with 339 software practitioners from 46 countries across five continents to quantitatively validate practitioners' expectations uncovered in our interviews. Finally, we perform a literature review of research papers published in premier venues from 2022 to 2024 (in the last three years), comparing the techniques proposed against the criteria that practitioners have for adoption. We address the following five Research Questions (RQs):

RQ1: What aspect is primarily relied on when writing testing code in software development practice? Most surveyed practitioners (57%) tend to develop their test code primarily based on requirement descriptions, while a smaller group (26%) bases their test code primarily on the code under test. 17% of the surveyed practitioners indicate that they do not partake in test code writing during the software development process due to various reasons.

RQ2: What is the state of automated test generation tools, and what are the issues? Among the surveyed practitioners, 28% report that they have used automated test generation tools, with Large Language Model (LLM) tools like ChatGPT being the

[*] Corresponding author

ISSTA '24, September 16–20, 2024, Vienna, Austria
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680386

most popular choice. Uncertainty about effectiveness and reliability is the main reason that surveyed practitioners abstain from using them. Over half of those with experience in the tools express dissatisfaction and highlight "Struggling to handle complex or specific scenarios" and "Limited support for different programming languages and products" as the most prominent issues.

RQ3: Are automated test generation tools important for practitioners? 95% of surveyed practitioners consider such tools to be worthwhile and essential for their software development. Furthermore, over 81% agree that these tools can significantly boost the efficiency of writing test code and aid in regression testing.

RQ4: What are practitioners' expectations of automated test generation tools? Practitioners prefer utilizing such tools within the internal network to generate test cases for unit testing. The correct rate (the proportion of test cases that accurately reflect the requirements) and bug detection capability emerge as the most critical evaluation metrics influencing practitioners' acceptance of these tools. Over half of them anticipate the correct rate and bug detection rate (the percentage of bugs identified by the generated test code) to exceed 80%. Additionally, most of them expect the tools to handle at least 10,000 lines of code, with an installation, configuration, and learning process taking under one hour. They also prefer the generation of a single test code to take less than 10 seconds and favor concise test code, not exceeding 100 lines.

RQ5: How close are the current state-of-the-art automated test generation studies to satisfying practitioners' needs before adoption? We identify 83 papers in automated test generation from 2022 to 2024. We find that 70% of studies primarily generate tests based on the code under test, contradicting surveyed practitioners' preference for generating tests based on requirement descriptions. No or very few recent studies propose automated test generation techniques for regression testing, acceptance testing, integration testing, load testing, and beta testing, which are areas of concern for surveyed practitioners. While coverage is widely used in the literature, surveyed practitioners prefer metrics that prioritize the correct rate and bug detection capability. When faced with complex real-world scenarios, the correct rate and bug detection rate of most proposed automated test generation techniques satisfy at most 47% of practitioners' expectations. This indicates a significant need for enhancements to align with practitioners' demands. Additionally, current tools based on LLMs fall short of meeting surveyed practitioners' expectations in handling large-scale projects.

In summary, our paper makes the following contributions:

(1) We interview 13 professionals and survey 339 practitioners from 46 countries to gain insights into their expectations. This includes their perspectives on the current automated test generation tools, the importance of automated test generation, their criteria for adopting such tools, and the factors influencing their decisions.

(2) We conduct an extensive literature review of papers published in the leading publications over the last three years. Then, we compare the current state of research with practitioners' expectations and outline potential implications to align research efforts with practitioners' needs and demands.

The subsequent sections of our paper are structured as follows: Section 2 outlines the methodology employed in our study. Section 3 presents the results obtained from our research. In Section 4, we discuss the implications of our findings and address potential threats to validity. Section 5 introduces related work in the field. Finally, Section 6 concludes our paper and discusses future work.

2 Research Methodology

The research methodology employed in this study follows a mixed-methods approach [39], as depicted in Figure 1 and comprises three stages. ***Stage* 1**: Conduct interviews with professionals to explore their practices in writing test code, their experiences and issues encountered when using automated test generation tools, and their expectations of such tools. ***Stage* 2**: Carry out an online survey designed to validate and broaden the conclusions obtained from the interviews regarding automated test generation. ***Stage* 3**: Perform a comprehensive literature review to assess the extent to which current state-of-the-art research fulfills practitioners' needs and expectations. Both the interviews and survey receive approval from the relevant institutional review board.

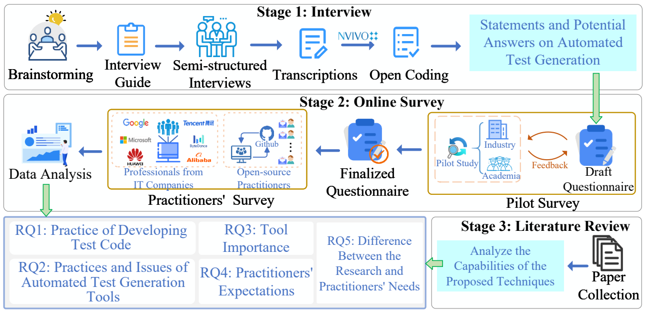
Stage 1: Interview
Brainstorming
Interview Guide
Semi-structured Interviews
Transcriptions
NVIVO
Open Coding
Statements and Potential Answers on Automated Test Generation
Stage 2: Online Survey
Data Analysis
Google
Microsoft
HUAWEI
Tencent 腾讯
ByteDance
Alibaba
Professionals from IT Companies
Github
Open-source Practitioners
Practitioners' Survey
Finalized Questionnaire
Pilot Study
Industry
Academia
Feedback
Draft Questionnaire
Pilot Survey
RQ1: Practice of Developing Test Code
RQ2: Practices and Issues of Automated Test Generation Tools
RQ3: Tool Importance
RQ4: Practitioners' Expectations
RQ5: Difference Between the Research and Practitioners' Needs
Stage 3: Literature Review
Analyze the Capabilities of the Proposed Techniques
Paper Collection

Figure 1: The overview of the research methodology.

2.1 Stage 1: Interview

Protocol: The first author conducts a series of face-to-face, in-depth, semi-structured interviews to thoroughly explore the practitioners' practices, issues, and expectations on automated test generation. An interview guide is developed through a brainstorming process to facilitate this exploration. Each interview consists of three parts. In the first part, demographic questions are asked to gather information about the interviewee's background, such as their job role, year of work experience, and team size. In the second part, the interviewees are asked to discuss their test code writing practice. In the third part, the concept of automated test generation is introduced to ensure that the interviewees understand how these techniques work and their potential benefits. Open-ended questions are then posed to understand the interviewees' experiences and expectations regarding automated test generation techniques.

Interviewees: Professionals from various roles, such as development, testing, and project management, are invited to participate in the interviews via our networks in the software industry. A total of 13 software practitioners from nine IT companies worldwide participated in the interviews. Each interview lasts between 40-60 minutes. They have an average 6.85 years of professional experience in software development (minimum: 2, median: 7, maximum: 15, standard deviation: 3.98 years). At the end of the interviews, the interviewees are thanked and briefly informed about our next plan.

Transcription and Open Coding: The first author transcribes and analyzes the interviews, using NVivo qualitative analysis software for open coding to generate opinion cards of the interview contents. The second author verifies the initial opinion cards created by the first author and provides suggestions for improvement. After incorporating these suggestions, the two authors separately analyze the opinion cards and sort the generated cards into potential statements and answers. The overall Cohen's Kappa value between the two authors is 0.81, indicating substantial agreement. Disagreements are discussed to reach a common decision. To reduce bias, both authors review and agree on the final set of statements. Eventually, based on the results of the interviews, we identify one test code development practice, eight issues with using automated test generation tools and seven reasons for not using them, five conclusions regarding the importance of automated test generation tools, and eight aspects concerning expectations for automated test generation tools.

2.2 Stage 2: Online Survey

Design: The survey consists of various question types, including single-choice questions, multiple-choice questions, short answer questions, and rating questions on a 5-point Likert scale, ranging from "Strongly Disagree" to "Strongly Agree". We also include an "I don't know" option for surveyed practitioners who do not comprehend our descriptions or prefer not to answer. To minimize bias due to surveyed practitioners' unfamiliarity with automated test generation, we provide a detailed explanation. Our description covers typical usage scenarios and input/output aspects. The survey consists of five sections:

(1) Demographics: This section gathers information about the surveyed practitioners, including their country/area of residence, primary job role, years of programming experience, and team size.

(2) Practice of Writing Test Code: This examines the aspects (requirement descriptions or code under test) surveyed practitioners rely on when writing test code in the development process.

(3) Automated Test Generation Tools: This section provides surveyed practitioners with a brief description of automated test generation tools and explores whether they have used such tools for software testing. It also asks about specific tools used and any issues faced while using them. For those who have not used such tools, we inquire about their primary reasons for not doing so.

(4) Tool Importance: In this section, surveyed practitioners are asked to indicate how they perceive the importance of a tool that meets their expectations using statements like Essential (I will use this tool daily), Worthwhile (I will use this tool), Unimportant (I will not use this tool), or Unwise (This tool will hinder my test code writing productivity). Additionally, they are queried about their main motivation for using or not using such tools.

(5) Practitioners' Expectations: This section investigates surveyed practitioners' expectations regarding the usage scenarios (i.e., generating test inputs, oracles, or cases), test levels and types (e.g., unit testing, system testing, acceptance testing, and performance testing), and network environments (i.e., public network, internal network, and offline) for utilizing automated test generation tools. Then, the section explores the important factors in determining their acceptance of using the tools, such as passing rate (the proportion of generated test code that is syntactically correct, compliable, and runnable), coverage rate, correct rate, and bug detection rate. Finally, the section explores the minimum adoption criteria of such tools in terms of effectiveness (the required thresholds of passing rate, coverage rate, correct rate, and bug detection rate), efficiency (the time required to generate a single test code), scalability (the capacity of tools to handle a specified number of lines of code), and conciseness (the length of generated test code).

At the end of the survey, we allow surveyed practitioners to provide free-text comments about automated test generation and our survey. In particular, we inquire about their perspectives on the potential opportunities and challenges that LLMs like ChatGPT could present for automated test generation. Surveyed practitioners may or may not provide any final comments. Before launching the survey, we conduct a pilot survey with four industrial experts and two academics specializing in software testing research. These individuals are not part of the interviewees or surveyed practitioners. We gather feedback on the survey length and the clarity and understandability of terms. Based on this feedback, we make minor adjustments to the draft survey and create a finalized version. To accommodate an international audience, we provided an English version of the survey on Google Forms. In addition, to support practitioners from China, we translate the survey into Chinese and made it available on the popular survey platform Wenjuanxing.

Table 1: The roles and working experiences of practitioners.

Role	Population	<1y	1-3y	3-5y	5-10y	>10y
Development	218	17	52	35	70	44
Testing	42	3	9	10	15	5
Algorithm Design	27	4	10	11	2	0
Architect	19	0	1	5	5	8
Project Manager	18	0	2	5	7	4
Others	15	1	4	1	4	5
	339	25	78	67	103	66

Note: Other roles related to software development, testing, and management include chief technology officer, software testing researchers, software security analysts, software operations developers, and system architecture and cloud solution analysts.

Respondent Recruitment: To obtain an adequate number of surveyed practitioners from diverse backgrounds, we first reach out to professionals within our social and professional network who work in IT companies and ask for their assistance in disseminating our survey. Specifically, we send invitations to our contacts at Google, Microsoft, Huawei, Tencent, ByteDance, Alibaba, and other companies, encouraging them to share our survey with their colleagues. Then, we collect the public email addresses of contributors from GitHub repositories. Our focus is on repositories hosting popular open-source projects, determined by the number of stars they receive. We send emails containing a link to our survey to 6,554 potential developers. From these emails, we receive 415 automatic replies indicating the absence of the recipients. Finally, we receive 363 survey responses, out of which two incomplete surveys and 15 responses with completion times of less than three minutes are excluded. In addition, we exclude seven responses from individuals with roles such as product manager and teaching professional, and one with an unspecified role.

The data reported herein is based on the remaining 339 valid responses. These surveyed practitioners represent 46 countries across five continents. China, the United States, and Germany are the top three countries with the most surveyed practitioners, comprising 63%, 8%, and 4% of the total respondents, respectively. Table 1 shows the distribution of the surveyed practitioners in terms of roles and work experience. Most surveyed practitioners are actively involved in software development and testing and have more than three years of experience.

Data Analysis: We analyze the survey results based on the types of questions. For both single-choice and multiple-choice questions, we provide the percentages of each option selected. We perform a qualitative analysis of open-ended questions by carefully examining the responses. To identify trends in the Likert-scale questions, we create bar charts (many of which are presented throughout this paper). We exclude "I don't know" ratings as they constitute a small minority (approximately 1%) of all ratings. To facilitate the replication of this study, the interview guide and questionnaire are available in the online link [1]. Due to the sensitive nature of the project-related information provided by the practitioners, we are unable to publicly release the results of our survey.

2.3 Stage 3: Literature Review

Research papers on automated test generation are typically published in software engineering, system security, and artificial intelligence. Therefore, we go through full research papers published in ICSE, PLDI, ESEC/FSE, ASE, ISSTA, TSE, TOSEM, ASPLOS, USENIX ATC, SP, CCS, USENIX Security, ACL, IJCAI, ICLR, NIPS, and AAAI spanning from 2022 to 2024. These papers are renowned for their significant contributions and represent the forefront of advancements in automated test generation capabilities. We first use DBLP to examine the titles for keywords like "generate", "construct", "build", "test case", "test oracle", "test input", or "fuzz", which aid in identifying papers related to automated test generation. "Fuzz" is included as a keyword because such papers may utilize fuzzing tests to generate input data for bug detection. Subsequently, we apply the snowballing method to ensure a thorough examination of the relevant literature. From our initial search, we identify 123 papers. After excluding 40 papers due to their irrelevance or their empirical focus, which did not introduce new automated test generation techniques, we finalize a selection of 83 papers, with 18 from ICSE, 7 from ESEC/FSE, 12 from ASE, 7 from ISSTA, 18 from TSE, 7 from TOSEM, 2 from ASPLOS, 4 from CCS, 3 from SP, 4 from USENIX Security, and 1 from AAAI. For each paper, the first two authors read its content and analyze the capabilities of the proposed technique [2]. For example, Zhao et al. [86] proposed the Avgust tool for generating usage-based tests for mobile applications. Their experiment demonstrated that 35 out of the generated 51 tests (68.6%) successfully fulfilled the intended usage. We categorize this technique based on various factors: the aspect considered when writing test code (requirement descriptions), the usage scenario (test input generation), the testing level (system testing and function testing), the access to service (available offline), and the passing rate (68.6%). The first two authors discuss and verify differences in capability analysis, and confirm the final results through further review of the papers. The Cohen's Kappa value of 0.83 between the two authors indicates a high level of consensus.

3 Results

3.1 RQ1: Practice of Developing Test Code

We investigate whether surveyed practitioners write test code in the development process. If they do, we examine the aspects (requirement descriptions or code under test) that they mainly consider when writing test code. As depicted in Figure 2, most surveyed practitioners write their test code primarily based on the requirement descriptions, with 14% completely relying on it and 43% mostly relying on it. They commonly cite three reasons for this preference:

✍ *Because the goal of the tests is to make sure that the code is doing what it was designed to do, and this is based on the requirements.*

✍ *Our project has excellent documentation, which allows to thoroughly study all features of the product being developed. This speeds up the speed of writing tests, allows you to describe the steps of test cases in as much detail as possible, and reduces the likelihood of errors.*

✍ *When tests are directly linked to requirements, it becomes easier to track testing progress and maintain them as the requirements evolve. If requirements change, it's clear which tests might need adjustment or reevaluation.*

22% of surveyed practitioners primarily write their test code based on the code under test, with 4% solely relying on it. This approach is generally driven by two factors: the lack of specific requirement documentation and the difficulty for requirement documents to cover all testing scenarios. Some note:

✍ *I know that TDD (Test Driven Development) is best, but I am working on a startup research project, and there is no one who perfectly defines requirements in such a form that it is easy to test against.*

✍ *Unit tests based on requirements are often difficult to cover all cases and are prone to changes with the evolution of requirements.*

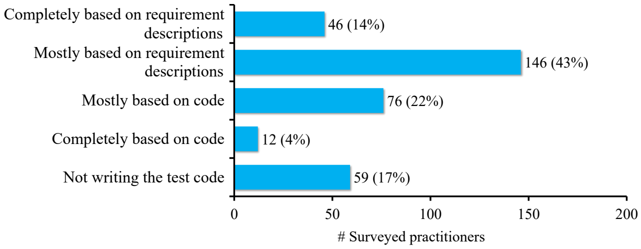

Figure 2: The aspects considered when writing test code.

Conversely, 17% of surveyed practitioners indicate that they do not write test code during the software development process. The reasons cited by these practitioners primarily include tight project delivery deadlines, lack of requirement for test code by the company or project, or the division of roles within the development team, where developers focus solely on writing code while dedicated testers handle the test code.

> **Finding 1.** In practical software development, most surveyed practitioners write their test code primarily based on requirement descriptions, while a minority write test code primarily based on the code under test.

[1] https://figshare.com/s/c44a70a2af62120c66aa

[2] Our review does not aim to compare the performance of different types of automated test generation techniques. Instead, it explores the optimal outcomes achievable with current techniques to meet practitioners' expectations.

Table 2: The current issues with conventional and LLM-based automated test generation tools.

	Issues	Distribution of conventional tools					Distribution of LLM tools				
		Strong Disagree	Disagree	Neutral	Agree	Strong Agree	Strong Disagree	Disagree	Neutral	Agree	Strong Agree
	The tools are unsatisfying for me		15%	27%	44%	12%		15%	28%	42%	9%
I_1	The installation and use of tools are difficult		14%	31%	38%	10%	15%	40%	23%	17%	
I_2	Limited support for different languages and products				64%	26%			25%	47%	16%
I_3	Struggle to handle complex or specific scenarios			10%	67%	24%			11%	28%	54%
I_4	Require lots of time to generate test cases		12%	26%	48%	10%	10%	25%	25%	27%	12%
I_5	The generated test code cannot find bugs effectively		15%	20%	42%	18%	10%	23%	23%	37%	
I_6	Lots of incorrectly generated test cases (oracle)	10%		41%	33%			18%	25%	37%	16%
I_7	The generated test code have limited coverage	12%	12%	24%	43%	10%		15%	35%	33%	13%
I_8	Poor readability and maintainability of generated code		17%	24%	40%	14%	17%	22%	19%	30%	13%

■ Strong Disagree ■ Disagree ■ Neutral ■ Agree ■ Strong Agree

3.2 RQ2: Practices and Issues of Automated Test Generation Tools

Practices of Automated Test Generation Tools: The survey results reveal that of 339 responses, 96 (28%) surveyed practitioners have used automated test generation tools. LLM tools like ChatGPT are the most commonly used tools among them, being used by 56% of surveyed practitioners who have utilized automated test generation tools. Additionally, some surveyed practitioners mention using internally developed LLM tools within their companies for automated test generation. Following LLM tools, Testful, Pynguin, and EvoSuite rank as the second, third, and fourth most commonly used tools, with adoption rates of 22%, 21%, and 16% among surveyed practitioners, respectively. Among the 339 responses, 243 (72%) surveyed practitioners have not used automated test generation tools. The survey findings shed light on the primary reasons for this lack of adoption, including uncertainty about the effectiveness and reliability of automated test generation tools (47%), a lack of technical expertise (37%), unawareness of their existence (34%), compatibility issues with project programming languages or technologies (22%), security and compliance concerns (20%), a requirement for payment (14%), and installation and learning concerns (9%). Notably, the percentages do not add up to 100% due to the multiple-choice nature of the question.

> 💡 **Finding 2.** 28% of surveyed practitioners report using automated test generation tools, with LLM tools like ChatGPT being the most commonly used. The primary reason other surveyed practitioners refrain from using such tools is uncertainty about their effectiveness and reliability.

Issues of Automated Test Generation Tools: Table 2 displays surveyed practitioners' assessments of issues encountered when using conventional automated test generation tools and LLM tools, respectively. Among practitioners employing conventional tools, 56% express dissatisfaction, while 51% of those using LLM tools find them unsatisfactory. Both groups identify "Struggling to handle complex or specific scenarios" as the primary issue, with 91% and 82% agreement, respectively. Some practitioners share:

📝 *They can carry the heavy load of writing boring tests, but you still need a human that brings the creativity of writing extreme edge cases.*

📝 *When using a tool like Evosuite, I went through a configuration and debugging phase of about an hour to generate unit tests for simple methods. However, to use it for more complex generation tasks, it requires additional time and effort to learn and debug. Moreover, the development of large-scale projects rich in dependencies may not be flexible enough. The cost of using it and the return on investment makes us hesitant to utilize it.*

📝 *I did try using ChatGPT to generate a test before. It sometimes works if the code is simple. But it usually gives strange and bizarre results when given a more complex code to test. It usually fixes the test code manually, which is probably slower than just writing it ourselves.*

The second most significant challenge for both sets of practitioners is "Limited support for different programming languages and products." Conventional tools indeed target specific languages like Pynguin [47] for Python and EvoSuite [23] for Java, contributing to this limitation. Additionally, some practitioners report LLMs generating inaccurate test cases for less common products and producing test code inconsistent with the specific product's coding style:

📝 *Copilot generated incorrect test cases for less popular frameworks, necessitating manual adjustments to guide it towards generating comprehensive test cases.*

📝 *LLM sometimes gives inconsistent test code (e.g., different test framework or code style). And we have to change it to match our codebase code style.*

Surveyed practitioners widely agree on several other issues. However, regarding the difficulty of installing and using automated test generation tools, only 23% find LLM tools challenging, while 48% struggle with conventional tools. As one respondent points out:

📝 *I think GPT-based testing tools will dominate the testing market (almost for sure). Conventional testing tools often have a learning curve and may be hard to use. Compared to such old-and-complicated tools, Copilot wins for sure.*

> 💡 **Finding 3.** Over half of the surveyed practitioners who used automated test generation tools express dissatisfaction with the current tools and consider "Struggling to handle complex or specific scenarios" and "Limited support for different programming languages and products" as the most significant issues.

3.3 RQ3: Tool Importance

Table 3 displays the ratings for the importance of automated test generation tools and practitioners' primary motivations for using these tools. About 95% of surveyed practitioners consider automated test generation tools either "Essential" or "Worthwhile". Specifically, 27% view them as "Essential" and intend to use them daily. The main motivations for tool adoption are "Efficiency and time savings" and "Assisting in regression testing", with 85% and 81% of practitioners

Table 3: The importance of the tool and the primary motivations for using automated test generation tools.

Motivations	Distribution
Importance	4% · 68% · 27%
M_1 Efficiency and Time Savings	13% · 49% · 36%
M_2 Bug detection	23% · 46% · 27%
M_3 Test coverage improvement	17% · 50% · 29%
M_4 Oracle correctness verification	25% · 48% · 24%
M_5 Regression test	17% · 46% · 35%

🟩 Unwise 🟩 Unimportant 🟥 Worthwhile 🟥 Essential
🟦 Strong Disagree 🟦 Disagree ⬜ Neutral 🟧 Agree 🟧 Strong Agree

expressing agreement, respectively. Some practitioners share their perspectives:

📝 *Tests occupy, on average, four times the amount of code as production code. Automating this process would save a significant amount of time, but the tool must be trustworthy and verifiable, even more so than manually written tests.*

📝 *The automated test generation tool may generate test cases not considered in manual testing scenarios. It can also save time for test engineers, complementing manual testing.*

📝 *In some scenarios, the test cases generated by automated test generation tools may not help us detect bugs. In most cases, what is needed is regression testing and smoke testing, which assist in verifying whether existing functionality has been affected and improve the efficiency of regression and smoke testing.*

Though most practitioners value such tools, a minority (5%) consider them unimportant due to doubts about their effectiveness.

📝 *The tool may not be simple enough, and developers may not be willing to learn how to use it. In complex business scenarios, it may not be able to generate useful test cases.*

📝 *My job involves building UI on the web. While I can see an automated test generator could create some unit tests for logic-heavy code, I can't imagine how it could produce integration tests given the intricacy of user needs and behaviors on the web.*

> **Finding 4.** 95% of surveyed practitioners recognize the importance of automated test generation tools in software development. Most believe these tools can improve test code writing efficiency and assist in regression testing.

3.4 RQ4: Practitioners' Expectations

Usage Scenarios: Figure 3 reveals that 86% and 53% of surveyed practitioners anticipate using automated test generation tools for generating test cases and oracles, respectively. Furthermore, 68% of respondents expect these tools to handle input generation. One respondent emphasizes the significance of test input generation, stating, "*I primarily use test data generation since many cases in our practice demand a substantial amount of data to be useful.*"

Test Levels and Types: Unit testing emerges as the most anticipated application of automated test generation tools, with 87% of surveyed practitioners indicating interest. This is closely followed by functional, integration, and regression testing, each garnering interest from over half of the surveyed practitioners (55%, 51%, and 51%, respectively).

Access to Service: The survey results show that 45% and 40% of surveyed practitioners anticipate using tools within an internal network or offline, respectively. In contrast, reliance on public networks is less favored (15%). This preference is primarily driven by the need to maintain the confidentiality of production code, as highlighted by surveyed practitioners: "*Due to the sensitivity of the business code, security takes a higher priority over efficiency. Unless there is an internal networked LLM, the testing tools for accessing LLMs may only be used for small projects.*" and "*The main problem is the security of the production code when using such tools.*"

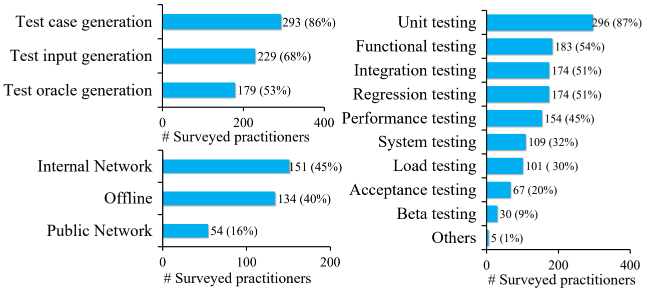

Figure 3: The usage scenarios, test levels and types, and access to services that surveyed practitioners expect.

Table 4: The factors that affect practitioners' acceptance of using automated test generation tools.

Factors	Distribution
F_1 Coverage	14% · 46% · 34%
F_2 Correct rate	40% · 50%
F_3 Bug detection	13% · 46% · 38%
F_4 Mutation score	22% · 43% · 27%
F_5 Passing rate	13% · 40% · 43%
F_6 Understandability/maintainability	11% · 43% · 41%
F_7 Similarity	9% · 12% · 19% · 39% · 21%
F_8 Easy to use	12% · 38% · 40%

🟦 Not Important 🟦 Somewhat Important ⬜ Moderately Important 🟧 Important 🟧 Very Important

Table 5: Practitioners' satisfaction rate with various capability ranges in terms of the effectiveness factors. (We exclude the mutation score and similarity because they are the least preferred by practitioners.)

Effectiveness Metrics	Distribution
Minimum passing rate	11% · 33% · 49%
Minimum correct rate	39% · 53%
Minimum bug detection rate	41% · 51%
Minimum code lines coverage rate	21% · 38% · 31%
Minimum branch coverage rate	21% · 37% · 32%
Minimum requirement coverage rate	15% · 38% · 40%

🟦 5%-20% 🟦 20%-40% ⬜ 40%-60% 🟧 60%-80% 🟧 80%-100%

Evaluation Metrics: Table 4 shows the factors (i.e., evaluation metrics) in determining their acceptance of using the tools. The most preferred evaluation metric is the correct rate. As some practitioners note: "*A false positive result would lead to an increase in our workload rather than improving efficiency.*" and "*I'd also add - the bar for acceptability is very high. I won't use a tool that's not completely accurate. Otherwise, it becomes a headache.*" The second most preferred metric is bug detection rate. As one practitioner notes, "*I think I would have a higher tolerance for unpredictable and random outputs of LLMs if the bugs it does find are novel.*"

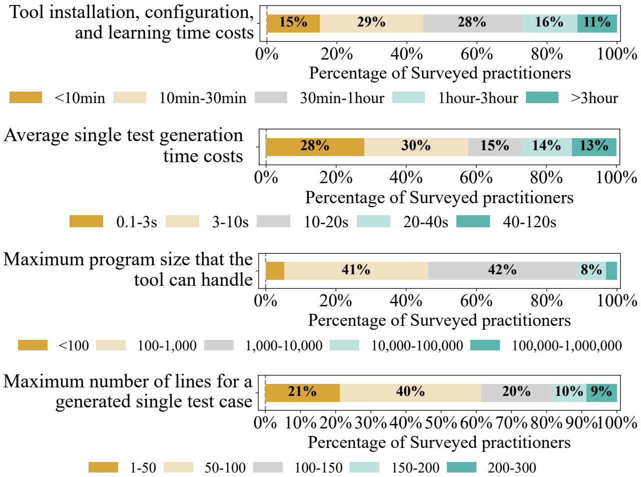

Figure 4: Practitioners' satisfaction rate with various capability ranges in terms of efficiency, scalability, and conciseness.

Effectiveness: Table 5 shows the surveyed practitioners' satisfaction rate with various capability ranges of automated test generation tools in terms of effectiveness factors. Satisfaction rates for each metric are distributed across five capability ranges, from less than 20% to 80-100%. The surveyed practitioners prioritize the correct rate of test cases or oracles, with 53% expressing the desire for a correct rate of 80-100% before considering the use of automated test generation tools. The second-highest requirement among surveyed practitioners is for the bug detection rate, with 51% hoping for a bug detection rate of 80%-100%.

> **Finding 5.** Correct rate and bug detection capability are the most critical factors influencing surveyed practitioners' acceptance of automated test generation tools. Over half of the surveyed practitioners expect the correct rate and bug detection rate to exceed 80%.

Efficiency: For tool installation, configuration, and learning time cost (shown in Figure 4), surveyed practitioners are most satisfied when these activities take 10-30 minutes, with 29% of surveyed practitioners favoring this range. Satisfaction decreases as time cost increases, with 28% satisfied with 30 minutes to an hour, 16% for 1-3 hours, and the least satisfaction at 11% for more than 3 hours. Regarding the average single test generation time, 28% of surveyed practitioners are satisfied when a test can be generated in 0.1-3 seconds. Satisfaction is highest (30%) for test generation times of 3-10 seconds, and it gradually decreases for longer times: 15% are satisfied with 10-20 seconds and 14% with 20-40 seconds.

Scalability [3] ***and Conciseness:*** Only 5% of surveyed practitioners are satisfied with tools that handle programs with less than 100 lines of code. Satisfaction substantially increases to 41% for tools handling program sizes between 100-1,000 lines, and it peaks at 42% for sizes between 1,000-10,000 lines. The majority of surveyed practitioners (40%) are satisfied with test cases comprising 50-100 lines of code. Satisfaction is lower, at 21%, for test cases with 1-50 lines, and it decreases further for test cases with more lines, with 20% satisfaction for 100-150 lines, 10% for 150-200 lines, and 9% for 200-300 lines, because overly long test cases might lead to poor readability and maintainability of test code [26].

> **Finding 6.** More than half of the surveyed practitioners expect automated test generation tools to handle at least 10,000 lines of code, with installation, configuration, and learning time of less than 1 hour, single test code generation time under 10 seconds, and generated test code length not exceeding 100 lines of code.

3.5 RQ5: Difference Between the Current Research and Practitioners' Need

After our literature review process, we identify a total of 83 papers. Due to space constraints, we provide a detailed overview of the current automated test generation techniques in the online link.

Aspects considered when writing test code: In the 83 papers, only a small portion of studies (25 papers, 30%) generate tests primarily based on requirement descriptions, while most studies (58 papers, 70%) generate tests primarily based on the code under test.

Usage Scenarios: There are 40 papers (48%) focused on test case generation, 37 papers (45%) on test input generation, and 6 papers (7%) on test oracle generation.

Test levels and types: Across different testing levels and types, there are 64 (77%), 64 (77%), 22 (27%), 2 (2%), 1 (1%), 1 (1%), 1 (1%) papers respectively focusing on test generation for system testing, functional testing, unit testing, regression testing, performance testing, acceptance testing, and integration testing. Surprisingly, no papers specifically address load testing and beta testing.

Access to service: Among the 83 papers, the majority (72 papers, 87%) focus on developing test generation tools that can be accessed offline as standalone software, tool libraries, or IDE plugins. Seven papers (8%) concentrate on internal network services and use open-source LLMs that can be trained locally, e.g., Nie et al. [54] introduced the CodeT5 model, while Mastropaolo et al. [50] utilized the T5 model. The remaining five papers (6%) [15, 16, 40, 45, 58] leverage public LLM services by employing OpenAI's Codex or GPT 3.5 models.

> **Finding 7.** The majority (70%) of the studies generate tests primarily based on the code under test, which contradicts the surveyed practitioners' preference of generating tests based on requirement descriptions. No or few studies in recent three years have proposed automated test generation techniques for regression testing, acceptance testing, integration testing, load testing, and beta testing, which are areas of concern for the surveyed practitioners.

Evaluation metrics: Among the surveyed 37 papers on test input generation, bug detection (23 papers, 62%), coverage (21 papers, 56%), and passing rate (8 papers, 21%) are the primary metrics employed. Similarly, among the surveyed 46 papers on test case or oracle generation, bug detection (25 papers, 54%), coverage (20 papers, 43%), and correct rate (12 papers, 26%) are the primary metrics utilized. While coverage remains the commonly used metric, it does not align with the preferences of surveyed practitioners. The preference ranking shown in Table 4 underscores that surveyed practitioners prioritize quality attributes such as correct rate, passing rate, and understandability and maintainability of the generated test code, in addition to bug detection capability.

[3] The values of program sizes refer to Kochhar et al. [37], which explored the expectation of the minimum detected lines of code of automated fault localization tools can handle.

Table 6: Practitioners' capability expectations and the capabilities of current research. *Cap. Range* represents the capability range. *Sat. Rate* represents the satisfaction rate of surveyed practitioners' choices. (We exclude the mutation score and similarity because they are the least preferred by practitioners.)

	Description	Cap. Range	Sat. Rate	Papers
Effectiveness	Passing rate	80-100%	100%	[25, 42, 45, 61, 62, 87]
		60-80%	51%	[4, 33, 86]
		40-60%	18%	-
		<40%	7%	[54, 76]
	Correct rate	80-100%	100%	[5, 8, 35, 59, 81, 83]
		60-80%	47%	[17, 50]
		40-60%	8%	[58, 82]
		<40%	3%	[18, 46]
	Line coverage rate	80-100%	100%	[12, 30, 48, 53, 69]
		60-80%	69%	[58, 84, 88]
		40-60%	31%	[16, 27, 32, 81]
		<40%	10%	[15, 41]
	Branch coverage rate	80-100%	100%	[12, 53, 77, 85]
		60-80%	68%	[48]
		40-60%	30%	[58, 78, 88]
		<40%	9%	[34, 38, 41, 44, 81]
	Requirement coverage rate	80-100%	100%	[4]
		60-80%	60%	-
		40-60%	22%	[81]
		<40%	7%	[63]
	Bug detection rate	80-100%	100%	[14, 38, 59]
		60-80%	37%	[30, 51]
		40-60%	5%	[5, 28]
		<40%	1%	[18, 46]
Efficiency	Average single test generation time costs	0.1-3s	100%	[74, 80, 81, 85]
		3-10s	72%	[65]
		10-40s	42%	[13, 28, 41, 62]
		40-120s	13%	[29, 35, 68]
Scalability	Program size that tools can handle	100,000-1,000,000	100%	[10, 52, 56, 67, 77, 84]
		10,000-100,000	96%	[78, 80]
		1,000-10,000	88%	[54, 58]
		<1,000	46%	[15, 40, 59, 85]
Conciseness	Number of lines for a single generated test case	1-50	100%	[54, 62, 82, 86]
		50-100	79%	-
		100-150	39%	-
		150-300	19%	[85]

> **Finding 8.** While coverage is widely used in literature, there is a clear preference among surveyed practitioners for quality attribute metrics like correct rate, passing rate, and understandability and maintainability of generated test code.

Effectiveness: As shown in Table 6, regarding the **passing rate**, an automated test generation technique supporting capabilities in the ranges of 80-100%, 40-80%, 40-60%, and <40% satisfies at least 100%, 51%, 18%, and 7% of the surveyed practitioners, respectively. 11 papers utilize the passing rate metric, with six papers [25, 42, 45, 61, 62, 87] achieving capabilitiy between 80%-100%, meeting the requirements of 100% of practitioners. These studies primarily focus on generating test inputs based on parameter specifications, employing fuzzing techniques with some domain information, or utilizing LLMs based on GUI context to generate input. Consequently, they have achieved a high passing rate. For instance, Sun et al. [62] introduced JUnitTestGen, capable of effectively handling the proper initialization of API caller instances and their parameters, ensuring that test cases are not only syntactically correct but also set up the execution environment correctly before conducting API tests. Liu et al. [45] utilized LLMs to generate text inputs within the UI page, combined with context-aware input prompt generation and prompt-based data tuning methods, effectively integrating them with automated GUI testing tools. However, two papers [54, 76] achieved very low passing rates. Nie et al. [54] utilized deep learning techniques to generate complete test case code (rather than just the test input) for 1,270 open-source Java projects, generating 76.22% compilable code and 28.63% runnable code. Xie et al. [76] analyzed API documentation to extract specific input constraints for deep learning API functions. The ratio of passing inputs was low (33.4%), because the API documents are often incomplete [76].

Regarding the **correct rate**, an automated test generation technique supporting capabilities in the ranges of 80-100%, 60-80%, 40-60%, and <40% satisfies at least 100%, 51%, 18%, and 7% of the surveyed practitioners, respectively. Among the surveyed 83 papers, 12 (14%) papers report the correct rate. Of these, 6 papers (50%) [5, 8, 35, 59, 81, 83] achieve a correct rate of 80%-100%, which can satisfy 100% of our surveyed practitioners. However, their test case or oracle generation scenarios or tasks are relatively simple, not requiring complex analysis or inference of the functionality of the code under test. For example, Alonso et al. [5] analyzed previous API requests and their corresponding responses to generate invariants as test oracles. For instance, in a Java function that receives an array and returns the same array with an additional element, a generated invariant could specify that the returned array always has a greater size than the array provided as input, i.e., $size(return.array[]) > size(input.array[])$. This oracle generation scenario is relatively simple, as it mainly involves observing and analyzing existing data to generate invariants as test oracles rather than generating functional test oracles for API. Zamprogno et al. [83] directly executed test inputs and recorded the variable values after execution to generate assert statements. Kim et al. [35] generated driving scenarios for autonomous driving systems, and the oracles for these test scenarios only include three types: collisions, infractions, and immobility. Su et al. [59] generated test cases for smart contracts, and their oracles are six specific types of smart contract vulnerabilities (e.g., ether leaking).

Additionally, 2 (17%), 2 (17%), and 2 (17%) papers meet the requirements of at least 47%, 8%, and 3% of participants, respectively. The lower correct rates observed in some studies are primarily due to the complexity of the generation scenarios. For example, Liu et al. [46] trained neural networks to generate functional test oracles given the test input generated by EvoSuite [23]. Their results on the Defects4J containing 17 real-world Java projects showed the correct rate of generated oracles was less than 40%. Schäfer et al. [58] introduced TESTPILOT, which utilized GPT 3.5 to generate unit tests for methods in the API of a given project. They evaluated TESTPILOT on 25 npm packages encompassing a total of 1,684 API functions. The generated tests, due to a large number of timeout errors, assertion errors, and correctness errors, resulted in a median correct rate of only 48%.

For the **line coverage rate**, an automated test generation technique supporting capabilities in the ranges of 80-100%, 60-80%, 40-60%, and < 40% coverage satisfies 100%, 69%, 31%, and 10% of surveyed practitioners, respectively. For the **branch coverage rate**, those numbers are slightly adjusted to 100%, 68%, 30%, and 9%. When it comes to the **requirement coverage rate**, the satisfaction levels change to 100%, 60%, 22%, and 7% for the same coverage ranges, respectively. While certain automated test generation methods achieve relatively high line, branch, and requirement coverage rates, it is because the methods primarily focus on optimizing for improved coverage rates. For example, Alonso et al. [4] analyzed API parameter specifications and extracted real test data from knowledge bases like DBpedia for Web APIs. This method, relying on extensive knowledge base searches, ensures that the generated test inputs cover a wide range of scenarios, thus achieving high coverage. However, coverage rates still face challenges in more complex projects. Specifically, 43% of the papers (6 out of 14), 62% of the papers (8 out of 13), and 67% of the papers (2 out of 3) showed performance below 60% in line coverage, branch coverage, and requirement coverage, respectively, with at most 31% of surveyed practitioners being satisfied. For example, Deng et al. [15] utilized LLMs as zero-shot fuzzers to generate test inputs for deep learning libraries. Despite its effectiveness in producing human-like code snippets and enhancing test coverage through the automatic mutation of diverse deep learning program inputs, it faced challenges with mainstream libraries like TensorFlow and PyTorch, resulting in a line coverage rate below 40%. Additionally, Kukucka et al. [38] proposed to combine concolic execution and taint tracking with fuzzing to automatically generate defect-revealing inputs. Their experiment indicated that across five real large-scale projects (Apache Ant, Maven, BCEL, Google Closure, and Mozilla Rhino) with varying branch counts (ranging from 5,858 to 49,602 branches), the proposed method achieved branch coverage rates of 4% - 23%.

Regarding the **bug detection rate**, 3, 2, 2, and 2 automated test generation technique with capabilities in the ranges of 80-100%, 60-80%, 40-60%, and <40% satisfies at least 100%, 37%, 5%, and 1% of the surveyed practitioners, respectively. We find that the 48 papers that explore the bug detection capability of the generated test code employ different evaluation metrics. For instance, Liu et al. [46] measured the number of test cases required to examine to find the first bug, Li et al. [41] counted the total number of detected bugs in deep learning libraries, and Davis et al. [14] used the bug detection rate. We have chosen the bug detection rate as the metric in our survey, as it is a relative indicator that is easier to evaluate the effectiveness of automated test generation tools compared to absolute indicators such as those used in [14, 41]. 9 papers utilize the bug detection rate as the evaluation metric. Three studies [14, 38, 59] achieve a bug detection rate of 80-100%. In the study by Davis et al. [14], the six programs under test contain only 3-57 lines of code, and the types of bugs included are relatively simple, such as infinite loops, NaN outputs, and divide by zero errors. Su et al. [59] also pointed out in their paper that some vulnerability types in their experimental dataset are relatively easier to detect, as they require simpler transaction sequences to trigger. Kukucka et al. [38] achieved a bug detection rate of 100% on the Apache Ant project (the version they used had only one bug), but only 29% on a more complex Google Closure project. This suggests that when generating test code for relatively simple software projects, the bug detection rate of the test code is relatively higher. In contrast, the remaining 6 (83%) papers [5, 18, 28, 30, 46, 51] report bug detection rates below 80% on more realistic and complex programs, such as the autonomous vehicle software system [28], and the Defects4J dataset containing 835 bugs from 17 real-world Java projects [18, 46]. This indicates the bug detection capability of the proposed test generation techniques on real-world and complex scenarios has not met the needs of practitioners.

> **Finding 9.** When confronted with complex real-world scenarios, the majority of proposed automated test generation techniques achieve a correct rate and bug detection rate of less than 80%, satisfying at most 47% of practitioners' expectations. This indicates a pressing need to enhance their capabilities to meet the high thresholds for adoption (where over half of the practitioners expect the correct rate and bug detection rate to exceed 80%).

Efficiency: Regarding the installation, configuration, and learning time costs, our survey results reveal that an automated test generation technique with capabilities in the ranges of <10min, 10-30min, 30min-1hour, and >1hour satisfies at least 100%, 84%, 55%, and 27% of the surveyed practitioners, respectively. As one surveyed practitioner mentioned, " *I think ease of use and installation is a critical factor. I'm not going to bother with new technology if it is too complex to quickly figure out.*". However, none of the reviewed papers utilize this metric to evaluate tool efficiency. Regarding the time taken to generate a single test, an automated test generation technique with capabilities in the ranges of 0.1-3s, 3-10s, 10-40s, and 40-120s satisfies at least 100%, 72%, 42%, and 13% of the surveyed practitioners, respectively. 12 papers report the test generation time costs for evaluation. Of these, 4 (33%) papers can satisfy 100% of our surveyed practitioners, while 1 (8%), 4 (33%), and 3 (25%) papers can satisfy at least 72%, 42%, and 13% of our surveyed practitioners.
Scalability: When considering the size of programs that can be handled, an automated test generation technique with capabilities in the ranges of 100,000-1,000,000 lines, 10,000-100,000 lines, 1,000-10,000 lines, and <1,000 lines satisfies at least 100%, 96%, 88%, and 46% of the surveyed practitioners, respectively. We determine the program size a tool can handle based on the maximum number of lines of software under test or experimental input settings mentioned in the papers. For instance, Yandrapally et al. [78] showcased web applications used in their evaluation, with the largest application having 60K lines of code, falling into the 10,000-100,000 lines range. When employing LLMs, Deng et al. [15] set the *max_tokens* = 256, which we classify as the <1,000 lines range. 14 papers mention the program size their tools can handle. Among them, 10 papers (71%) can meet the needs of at least 88% of the surveyed practitioners, while the remaining 4 papers (29%) can process programs with less than 1,000 lines due to input capacity limitations based on LLMs.
Conciseness: In terms of the number of lines per generated test case, an automated test generation technique with capabilities in the ranges of 1-50, 50-100, 100-150, and 150-300 lines satisfies at least 100%, 79%, 39%, and 19% of the surveyed practitioners, respectively. Among the 83 papers collected, only 5 papers specify the line of the code for a single test case generated by the test generation tool,

with 4 papers (80%) producing test cases under 50 lines and 1 paper (20%) between 150 and 300 lines. These findings indicate that the tools' output sizes generally meet practitioners' expectations.

> **Finding 10.** The majority of the automated test generation techniques proposed in the surveyed papers can satisfy most practitioners' needs regarding test generation time, program size handling, and test case generation size. However, LLMs may fall short in meeting the demands of surveyed practitioners for processing large-scale projects.

4 Discussion

4.1 Implications

Our results highlight some implications for research communities:

(1) Generating test code based on requirement descriptions rather than code under test: While most surveyed practitioners prefer writing test code based on requirement descriptions, existing research mainly focuses on generating test code from the code under test. This indicates a need for building more automated test generation tools that can generate test code from specified requirement descriptions, ensuring effective validation of software functionality. Some surveyed practitioners noted: "*It's unclear to me how generating test cases based on the code under test is beneficial: if the code does not implement the requirements correctly, testing that this incorrect interpretation of the requirements is followed by the code is of no benefit.*", " *I would like to see if LLM or copilot can help engineer on writing good tests especially in TDD practice where engineer can use the tool to write down test as specifications even before writing the code/implementation.* ".

(2) Implication on automated test generation tools: (a) The majority of surveyed practitioners are not using automated test generation tools due to doubts about their effectiveness and reliability. Moreover, those who have employed such tools express dissatisfaction, highlighting "Struggling to handle complex or specific scenarios" and "Limited support for different programming languages and products" as the most prominent issues. As one surveyed practitioner pointed out, "*The most important thing is that the generated test code is consistent and reliable. My problem with such LLM models is that they produce unsafe or generally ineffective code most of the time.*" In addition, Finding 9 demonstrates that when confronted with more complex real-world testing scenarios, the capabilities of most current automated test generation techniques fall short in meeting practitioners' expectations in terms of correct rate and bug detection rate. Therefore, there is strong anticipation among surveyed practitioners for advanced test generation tools that can support complex scenarios, different programming languages, and products, and effectively assist developers in writing reliable tests in real production environments. (b) Regarding the test generation based on requirement descriptions, some surveyed practitioners emphasize in their feedback: *"There are few tools based on requirements in practice. They usually need strict adherence to detailed specs, are complex to learn, and can only create simple tests. Once used, you likely won't want to use them again."* Existing test generation methods based on requirement descriptions, like Docter [76], face limitations such as incomplete specifications, handling complex constraints, and scalability issues [76]. (c) As revealed in Finding 7, recent studies have overlooked proposing automated test generation techniques for regression testing, acceptance testing, integration testing, load testing, and beta testing, despite high demand from practitioners. This discrepancy underscores the need for future research to prioritize these areas.

Moreover, some surveyed practitioners provide suggestions regarding automated test generation at the end of the survey. Instead of generating entirely new test code, they suggest refactoring existing test code or automatically updating the existing test code based on source code or requirement changes. As some surveyed practitioners mentioned, "*Most automated test code generation tools test only for functionality and not complexity or space requirements or speed of the code under test. Some fields or industries require that the code is fully optimized for speed or fully optimized not to take up space.*", "*I would like an automated test code generation tool that generates test for functionality (most tools do this), complexity (memory space and execution time, I have never seen a good tool that does this) and optimizations (finding ways to produce the same result with fewer steps).*", "*Frequent updates in the logic code make it challenging to ensure that the corresponding test code is also kept up to date.*", "*The current issue is that the majority of requirements are constantly changing, leading to the test code becoming easily outdated. This results in high maintenance costs for automated testing, with returns less than the investment.*"

(3) Suggestions for LLM-based automated test generation: LLM tools are increasingly popular for generating test code [70], but over half of surveyed practitioners express dissatisfaction with their performance. To address these concerns, many surveyed practitioners have voiced their opinions and suggestions at the end of the survey. (a) **Training LLMs with customized and high-quality datasets:** *"I don't believe that ChatGPT would be helpful in writing test cases unless it was trained on our code and understood better the requirements of test cases."*, *"I think the main problem with tools based on LLMs is their training dataset. In most cases, these models are trained on public GitHub repositories, and the issue is that many of those repositories are of poor quality, serving as examples of how not to write code. All this poor-quality code will be recommended by the LLM of your choice, not because the technology behind LLMs is flawed, but because of the dataset it uses for training."* (b) **Addressing code dependency challenges:** *"Using LLMs like ChatGPT to write test code is an interesting idea. But the dependencies in the code are a problem, and using external tools is always too cumbersome and takes you away from the development environment."* (c) **Emphasizing output correctness (including oracle) of LLMs**: *"The ability of large models to generate unit test cases is unquestionable, but when it comes to complex interactive functionality testing, can it generate the expected results after feeding all the code to it? How does it know what the oracle should be?"*, "*It should have mutmut, codecov, or other existing tools in the background to check ChatGPT's output*."

(4) Implication on evaluation metrics: While academic papers prioritize code coverage, surveyed practitioners perfer quality attribute metrics like correct rate, passing rate, understandability, and maintainability in the generated test code, in addition to bug detection capability. As someone stated, "*Code and branch coverage for the sake of coverage stats alone is not useful outside of identifying undefined behavior/crash bugs.*", "*Automated test generation tools first need to accurately understand the logic of the code. The ability to generate compilable test code that can verify the logic of the code is the most basic requirement*". Prioritizing these metrics—correct rate, passing rate, understandability and maintainability, and bug detection capability— will better align research efforts with industry needs and ensure the effectiveness of such tools.

4.2 Threats to Validity

We survey 339 practitioners from 46 countries spanning 5 continents. Our surveyed practitioners include individuals working for various companies, such as Google, Microsoft, Huawei, Tencent, ByteDance, Alibaba, and many others, and those contributing to open-source projects hosted on GitHub in diverse roles. However, our findings may not fully capture the expectations of all software engineers. For instance, our survey does not include practitioners who are not proficient in either English or Chinese. We focus on several factors that may influence the adoption of automated test generation tools. Nevertheless, there could be other factors contributing to adoption that have not been explored in our study. We plan to examine these factors in future research. In addition, our survey can only estimate practitioners' willingness to adopt such tools. The actual adoption process is complex, involving not only individual perceived willingness but also factors such as organizational support (e.g., training and incentives) and social influence (e.g., support from peers/colleagues) [73, 75]. These factors contribute to the overall adoption of such techniques. Nevertheless, individual attitudes play a significant role in influencing actual adoption, and our survey specifically measures this aspect.

5 Related Work

Automated Test Generation. Automated test [6] generation can be categorized into: test case generation [24], test input generation, and test oracle generation [70]. Various test case generation methods have been proposed, including: random-based [14, 72], search-based [11, 21, 40, 43, 60, 79, 88], symbolic execution-based [31], machine/deep learning-based [8, 21, 69, 81, 86], and LLM-based [9, 40, 54]. For example, Blasi et al. [8] utilized natural language processing techniques to extract requirement descriptions or specification information for generating test cases. Nie et al. [54] fine-tuned the CodeT5 model to generate test cases. Test input generation aims to automatically create input data for test execution, particularly for system testing purposes [70]. There is a growing interest in test input generation for various applications such as deep learning libraries/systems [15, 41, 64], autonomous driving software [29, 65, 68], web applications [48], and mobile applications [19, 45]. For example, Deng et al. [15] used a generative LLM (Codex) and an infilling LLM (InCoder) to generate and mutate input deep learning programs for fuzzing deep learning libraries. Huai et al. [29] proposed a search-based test input generation technique called DoppelTest, which revealed software bugs by generating scenarios containing multiple autonomous vehicles that account for traffic control elements (e.g., traffic lights and stop signs). Liu et al. [45] developed the QTypist method based on LLM, which generated suitable text inputs for mobile GUI testing by comprehending the contextual information of the application. Test case generation methods often encounter the oracle problem [7] and cannot generate test cases that effectively expose functional bugs [46]. To address the issue, researchers [18, 46, 50, 82] have proposed to automatically generate test oracles. For example, Dinella et al. [18] proposed a Transformer-based approach named TOGA to infer functional test oracles given the test input generated by EvoSuite. Subsequently, Liu et al. [46] proposed several improvements for TOGA to make the evaluation of TOGA more realistic and applicable.

Studies on Software Testing Practice. Several recent studies investigated software testing practices via survey questionnaires or interviews. For example, Ahmad et al. [1] investigated practitioners' perceptions of test flakiness and how this varies between different industries. They [2] also investigated the root causes of test flakiness and mitigation strategies to understand how practitioners perceive it. Eck et al. [20] examined the perceptions of software developers about the nature, relevance, and challenges of flaky tests and developers' fixing strategies. Kochhar et al. [36] and Tran et al. [66] investigated the issue of test case quality, including how practitioners define test case quality and which aspects of test cases are important for quality assessment. In exploring the gap between industry and academia, Santos et al. [57] investigated the differences in interests between academic researchers and practitioners in software testing. Martins et al. [49] investigated the software testing practitioners' perspectives to evaluate the acquisition of knowledge about software testing in undergraduate courses. Fischbach et al. [22] explored quality factors of test artifacts that have a positive or negative impact on the activities of agile testers. Alégroth et al. [3] explored the experts' knowledge, best practices, and experiences with model-based testing. However, there have been no previous studies that delve into the practices, issues, and expectations of practitioners regarding automated test generation techniques.

6 Conclusion and Future Work

We interview 13 professionals and survey 339 practitioners to explore their testing practices and their views on automated test generation tools. Practitioners express dissatisfaction with existing tools due to difficulties in handling complex scenarios and limited support for different programming languages and products. The correct rate and bug detection capability emerge as critical factors influencing their acceptance of these tools. We also compare current research with practitioners' expectations, identifying areas for improvement to better meet practitioners' needs. Future studies could prioritize generating test code based on requirement descriptions to align with practitioners' preferences for writing test code. Additionally, studies could focus on refactoring existing test code or automatically updating it based on source code or requirement changes. LLM-based tools should utilize customized and high-quality datasets for training and address code dependency challenges to enhance their effectiveness.

Acknowledgments

We thank the interviewees and survey participants for their invaluable insights. This research is supported by the Ningbo Natural Science Foundation (No. 2023J292) and the National Research Foundation, under its Investigatorship Grant (NRF-NRFI08-2022-0002). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

References

[1] Azeem Ahmad, Ola Leifler, and Kristian Sandahl. 2019. Empirical analysis of factors and their effect on test flakiness-practitioners' perceptions. *arXiv preprint arXiv:1906.00673* (2019).

[2] Azeem Ahmad, Ola Leifler, and Kristian Sandahl. 2021. Empirical analysis of practitioners' perceptions of test flakiness factors. *Software Testing, Verification and Reliability* 31, 8 (2021), e1791.

[3] Emil Alégroth, Kristian Karl, Helena Rosshagen, Tomas Helmfridsson, and Nils Olsson. 2022. Practitioners' best practices to Adopt, Use or Abandon Model-based Testing with Graphical models for Software-intensive Systems. *Empirical Software Engineering* 27, 5 (2022), 103.

[4] Juan C Alonso, Alberto Martin-Lopez, Sergio Segura, Jose Maria Garcia, and Antonio Ruiz-Cortes. 2023. ARTE: Automated Generation of Realistic Test Inputs for Web APIs. *IEEE Transactions on Software Engineering* 49, 1 (2023), 348–363.

[5] Juan C Alonso, Sergio Segura, and Antonio Ruiz-Cortés. 2023. AGORA: Automated Generation of Test Oracles for REST APIs. In *the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1018–1030.

[6] Maurício Aniche, Christoph Treude, and Andy Zaidman. 2021. How developers engineer test cases: An observational study. *IEEE Transactions on Software Engineering* 48, 12 (2021), 4925–4946.

[7] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* 41, 5 (2014), 507–525.

[8] Arianna Blasi, Alessandra Gorla, Michael D Ernst, and Mauro Pezzè. 2022. Call Me Maybe: Using NLP to Automatically Generate Unit Test Cases Respecting Temporal Constraints. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–11.

[9] Carolin Brandt, Marco Castelluccio, Christian Holler, Jason Kratzer, Andy Zaidman, and Alberto Bacchelli. 2024. Mind the gap: What working with developers on fuzz tests taught us about coverage gaps. In *46th International Conference on Software Engineering: Software Engineering in Practice*. 157–167.

[10] Sicong Cao, Biao He, Xiaobing Sun, Yu Ouyang, Chao Zhang, Xiaoxue Wu, Ting Su, Lili Bo, Bin Li, Chuanlei Ma, et al. 2023. Oddfuzz: Discovering java deserialization vulnerabilities via structure-aware directed greybox fuzzing. In *2023 IEEE Symposium on Security and Privacy*. IEEE, 2726–2743.

[11] Junjie Chen, Chenyao Suo, Jiajun Jiang, Peiqi Chen, and Xingjian Li. 2023. Compiler test-program generation via memoized configuration search. In *45th IEEE/ACM International Conference on Software Engineering*. IEEE, 2035–2047.

[12] Peng Chen, Yuxuan Xie, Yunlong Lyu, Yuxiao Wang, and Hao Chen. 2023. Hopper: Interpretative fuzzing for libraries. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1600–1614.

[13] Garrett Christian, Trey Woodlief, and Sebastian Elbaum. 2023. Generating Realistic and Diverse Tests for LiDAR-Based Perception Systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering*. IEEE, 2604–2616.

[14] Matthew Davis, Sangheon Choi, Sam Estep, Brad Myers, and Joshua Sunshine. 2023. NaNofuzz: A Usable Tool for Automatic Test Generation. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1114–1126.

[15] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*. 423–435.

[16] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In *46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[17] Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. 2022. Fuzzing deep-learning libraries via automated relational api inference. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 44–56.

[18] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K Lahiri. 2022. Toga: A neural method for test oracle generation. In *Proceedings of the 44th International Conference on Software Engineering*. 2130–2141.

[19] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel testing of android apps. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 481–492.

[20] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding flaky tests: The developer's perspective. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 830–840.

[21] Patric Feldmeier and Gordon Fraser. 2022. Neuroevolution-based generation of tests and oracles for games. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 72:1–72:13.

[22] Jannik Fischbach, Henning Femmer, Daniel Mendez, Davide Fucci, and Andreas Vogelsang. 2020. What Makes Agile Test Artifacts Useful? An Activity-Based Quality Model from a Practitioners' Perspective. In *the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–10.

[23] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[24] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. 2015. Does automated unit test generation really help software testers? a controlled empirical study. *ACM Transactions on Software Engineering and Methodology* 24, 4 (2015), 1–49.

[25] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2022. Griffin: Grammar-free DBMS fuzzing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[26] Giovanni Grano, Simone Scalabrino, Harald C Gall, and Rocco Oliveto. 2018. An empirical investigation on the readability of manual and generated test cases. In *Proceedings of the 26th Conference on Program Comprehension*. 348–351.

[27] Jiazhen Gu, Xuchuan Luo, Yangfan Zhou, and Xin Wang. 2022. Muffin: Testing deep learning libraries via neural architecture fuzzing. In *Proceedings of the 44th International Conference on Software Engineering*. 1418–1430.

[28] Yuqi Huai, Sumaya Almanee, Yuntianyi Chen, Xiafa Wu, Qi Alfred Chen, and Joshua Garcia. 2023. sceno RITA: Generating Diverse, Fully-Mutable, Test Scenarios for Autonomous Vehicle Planning. *IEEE Transactions on Software Engineering* (2023).

[29] Yuqi Huai, Yuntianyi Chen, Sumaya Almanee, Tuan Ngo, Xiang Liao, Ziwen Wan, Qi Alfred Chen, and Joshua Garcia. 2023. Doppelgänger test generation for revealing bugs in autonomous driving software. In *2023 IEEE/ACM 45th International Conference on Software Engineering*. IEEE, 2591–2603.

[30] Ahmad Humayun, Miryung Kim, and Muhammad Ali Gulzar. 2023. Codependence Aware Fuzzing for Dataflow-Based Big Data Analytics. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1050–1061.

[31] Sungjae Hwang, Sungho Lee, Jihoon Kim, and Sukyoung Ryu. 2021. Justgen: effective test generation for unspecified JNI behaviors on JVMs. In *2021 IEEE/ACM 43rd International Conference on Software Engineering*. 1708–1718.

[32] Bokdeuk Jeong, Joonun Jang, Hayoon Yi, Jiin Moon, Junsik Kim, Intae Jeon, Taesoo Kim, WooChul Shim, and Yong Ho Hwang. 2023. Utopia: Automatic generation of fuzz driver using unit tests. In *2023 IEEE Symposium on Security and Privacy*. IEEE, 2676–2692.

[33] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. 2023. {DynSQL}: Stateful Fuzzing for Database Management Systems with Complex and Valid {SQL} Query Generation. In *32nd USENIX Security Symposium*. 4949–4965.

[34] Jiwon Kim, Benjamin E Ujcich, and Dave Jing Tian. 2023. Intender: Fuzzing Intent-Based Networking with Intent-State Transition Guidance. In *32nd USENIX Security Symposium*. 4463–4480.

[35] Seulbae Kim, Major Liu, Junghwan" John" Rhee, Yeuseok Jeon, Yonghwi Kwon, and Chung Hwan Kim. 2022. Drivefuzz: Discovering autonomous driving bugs through driving quality-guided fuzzing. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1753–1767.

[36] Pavneet Singh Kochhar, Xin Xia, and David Lo. 2019. Practitioners' views on good software testing practices. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice*. 61–70.

[37] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th international symposium on software testing and analysis*. 165–176.

[38] James Kukucka, Luís Pina, Paul Ammann, and Jonathan Bell. 2022. Confetti: Amplifying concolic guidance for fuzzers. In *Proceedings of the 44th International Conference on Software Engineering*. 438–450.

[39] Patricia Leavy. 2022. *Research design: Quantitative, qualitative, mixed methods, arts-based, and community-based participatory research approaches*. Guilford Publications.

[40] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. CODAMOSA: Escaping coverage plateaus in test generation with pre-trained large language models. In *International conference on software engineering*.

[41] Meiziniu Li, Jialun Cao, Yongqiang Tian, Tsz On Li, Ming Wen, and Shing-Chi Cheung. 2023. Comet: Coverage-guided model generation for deep learning library testing. *ACM Transactions on Software Engineering and Methodology* 32, 5 (2023), 1–34.

[42] Xiaoting Li, Xiao Liu, Lingwei Chen, Rupesh Prajapati, and Dinghao Wu. 2022. ALPHAPROG: reinforcement generation of valid programs for compiler fuzzing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 12559–12565.

[43] Jun-Wei Lin, Navid Salehnamadi, and Sam Malek. 2023. Route: Roads not taken in ui testing. *ACM Transactions on Software Engineering and Methodology* 32, 3 (2023), 1–25.

[44] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. Nnsmith: Generating diverse and valid test cases for deep learning compilers. In *the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 530–543.

[45] Zhe Liu, Chunyang Chen, Junjie Wang, Xing Che, Yuekai Huang, Jun Hu, and Qing Wang. 2023. Fill in the blank: Context-aware automated text input generation for mobile gui testing. In *2023 IEEE/ACM 45th International Conference on*

Software Engineering. 1355–1367.

[46] Zhongxin Liu, Kui Liu, Xin Xia, and Xiaohu Yang. 2023. Towards more realistic evaluation for neural test oracle generation. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 589–600.

[47] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 168–172.

[48] Bogdan Marculescu, Man Zhang, and Andrea Arcuri. 2022. On the faults found in rest apis by automated test generation. *ACM Transactions on Software Engineering and Methodology* 31, 3 (2022), 1–43.

[49] Luana Martins, Vinicius Brito, Daniela Feitosa, Larissa Rocha, Heitor Costa, and Ivan Machado. 2021. From Blackboard to the Office: A Look Into How Practitioners Perceive Software Testing Education. In *Evaluation and Assessment in Software Engineering*. 211–220.

[50] Antonio Mastropaolo, Nathan Cooper, David Nader Palacio, Simone Scalabrino, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2023. Using transfer learning for code-related tasks. *IEEE Transactions on Software Engineering* 49, 4 (2023), 1580–1598.

[51] Héctor D Menéndez, Michele Boreale, Daniele Gorla, and David Clark. 2020. Output sampling for output diversity in automatic unit test generation. *IEEE Transactions on Software Engineering* 48, 1 (2020), 295–308.

[52] Hector D Menendez and David Clark. 2021. Hashing fuzzing: introducing input diversity to improve crash detection. *IEEE Transactions on Software Engineering* 48, 9 (2021), 3540–3553.

[53] Mathieu Nassif, Alexa Hernandez, Ashvitha Sridharan, and Martin P Robillard. 2021. Generating unit tests for documentation. *IEEE Transactions on Software Engineering* 48, 9 (2021), 3268–3279.

[54] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J Mooney, and Milos Gligoric. 2023. Learning Deep Semantics for Test Completion. In *45th IEEE/ACM International Conference on Software Engineering*. 2111–2123.

[55] Chittoor V Ramamoorthy, S-BF Ho, and WT Chen. 1976. On the automated generation of program test data. *IEEE Transactions on software engineering* 4 (1976), 293–300.

[56] Seemanta Saha, Laboni Sarker, Md Shafiuzzaman, Chaofan Shou, Albert Li, Ganesh Sankaran, and Tevfik Bultan. 2023. Rare path guided fuzzing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1295–1306.

[57] Ronnie ES Santos, Ayşe Bener, Maria Teresa Baldassarre, Cleyton VC Magalhães, Jorge S Correia-Neto, and Fabio QB da Silva. 2019. Mind the gap: are practitioners and researchers in software testing speaking the same language?. In *7th International Workshop on Conducting Empirical Studies in Industry*. 10–17.

[58] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* (2023).

[59] Jianzhong Su, Hong-Ning Dai, Lingjun Zhao, Zibin Zheng, and Xiapu Luo. 2022. Effectively generating vulnerable transaction sequences in smart contracts with reinforcement learning-guided fuzzing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[60] Baicai Sun, Dunwei Gong, Feng Pan, Xiangjuan Yao, and Tian Tian. 2023. Evolutionary generation of test suites for multi-path coverage of MPI programs with non-determinism. *IEEE Transactions on Software Engineering* (2023).

[61] Baicai Sun, Dunwei Gong, Tian Tian, and Xiangjuan Yao. 2020. Integrating an ensemble surrogate model's estimation into test data generation. *IEEE Transactions on Software Engineering* 48, 4 (2020), 1336–1350.

[62] Xiaoyu Sun, Xiao Chen, Yanjie Zhao, Pei Liu, John Grundy, and Li Li. 2022. Mining android api usage to generate unit test cases for pinpointing compatibility issues. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 70:1–70:13.

[63] Yang Sun, Christopher M Poskitt, Jun Sun, Yuqi Chen, and Zijiang Yang. 2022. LawBreaker: An approach for specifying traffic laws and fuzzing autonomous vehicles. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[64] Zohdinasab Tahereh, Vincenzo Riccio, Tonella Paolo, et al. 2023. DeepAtash: Focused Test Generation for Deep Learning Systems. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*. 954–966.

[65] Haoxiang Tian, Guoquan Wu, Jiren Yan, Yan Jiang, Jun Wei, Wei Chen, Shuo Li, and Dan Ye. 2022. Generating Critical Test Scenarios for Autonomous Driving Systems via Influential Behavior Patterns. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 46:1–46:12.

[66] Huynh Khanh Vi Tran, Nauman Bin Ali, Jürgen Börstler, and Michael Unterkalmsteiner. 2019. Test-case quality–understanding practitioners' perspectives. In *20th International Conference on Product-Focused Software Process Improvement*. 37–52.

[67] Vasudev Vikram, Isabella Laybourn, Ao Li, Nicole Nair, Kelton OBrien, Rafaello Sanna, and Rohan Padhye. 2023. Guiding greybox fuzzing with mutation testing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 929–941.

[68] Meriel von Stein, David Shriver, and Sebastian Elbaum. 2023. DeepManeuver: Adversarial Test Generation for Trajectory Manipulation of Autonomous Vehicles. *IEEE Transactions on Software Engineering* (2023).

[69] Chunhui Wang, Fabrizio Pastore, Arda Goknil, and Lionel C Briand. 2022. Automatic generation of acceptance test cases from use case specifications: an nlp-based approach. *IEEE Transactions on Software Engineering* 48, 2 (2022), 585–616.

[70] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2023. Software testing with large language model: Survey, landscape, and vision. *arXiv preprint arXiv:2307.07221* (2023).

[71] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *the ACM/IEEE 42nd International Conference on Software Engineering*. 1398–1409.

[72] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free lunch for testing: Fuzzing deep-learning libraries from open source. In *44th International Conference on Software Engineering*. 995–1007.

[73] Jim Witschey, Olga Zielinska, Allaire Welk, Emerson Murphy-Hill, Chris Mayhorn, and Thomas Zimmermann. 2015. Quantifying developers' adoption of security tools. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 260–271.

[74] Trey Woodlief, Sebastian Elbaum, and Kevin Sullivan. 2022. Semantic image fuzzing of AI perception systems. In *Proceedings of the 44th International Conference on Software Engineering*. 1958–1969.

[75] Shundan Xiao, Jim Witschey, and Emerson Murphy-Hill. 2014. Social influences on secure development tool adoption: why security tools spread. In *17th ACM conference on Computer supported cooperative work & social computing*. 1095–1106.

[76] Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W Godfrey. 2022. Docter: Documentation-guided fuzzing for testing deep learning api functions. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 176–188.

[77] Cong Yan, Suman Nath, and Shan Lu. 2023. Generating Test Databases for Database-Backed Applications. In *2023 IEEE/ACM 45th International Conference on Software Engineering*. IEEE, 2048–2059.

[78] Rahulkrishna Yandrapally, Saurabh Sinha, Rachel Tzoref-Brill, and Ali Mesbah. 2023. Carving UI Tests to Generate API Tests and API Specification. In *2023 IEEE/ACM 45th International Conference on Software Engineering*. 1971–1982.

[79] Rahul Krishna Yandrapally and Ali Mesbah. 2022. Fragment-based test generation for web apps. *IEEE Transactions on Software Engineering* 49, 3 (2022), 1086–1101.

[80] Xiangjuan Yao, Gongjie Zhang, Feng Pan, Dunwei Gong, and Changqing Wei. 2020. Orderly generation of test data via sorting mutant branches based on their dominance degrees for weak mutation testing. *IEEE Transactions on Software Engineering* 48, 4 (2020), 1169–1184.

[81] Guixin Ye, Tianmin Hu, Zhanyong Tang, Zhenye Fan, Shin Hwei Tan, Bo Zhang, Wenxiang Qian, and Zheng Wang. 2023. A Generative and Mutational Approach for Synthesizing Bug-Exposing Test Cases to Guide Compiler Fuzzing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1127–1139.

[82] Hao Yu, Yiling Lou, Ke Sun, Dezhi Ran, Tao Xie, Dan Hao, Ying Li, Ge Li, and Qianxiang Wang. 2022. Automated assertion generation via information retrieval and its integration with deep learning. In *Proceedings of the 44th International Conference on Software Engineering*. 163–174.

[83] Lucas Zamprogno, Braxton Hall, Reid Holmes, and Joanne M Atlee. 2023. Dynamic Human-in-the-Loop Assertion Generation. *IEEE Transactions on Software Engineering* 49, 4 (2023), 2337–2351.

[84] Man Zhang, Andrea Arcuri, Yonggang Li, Yang Liu, and Kaiming Xue. 2023. White-box fuzzing RPC-based APIs with EvoMaster: An industrial case study. *ACM Transactions on Software Engineering and Methodology* 32, 5 (2023), 1–38.

[85] Qian Zhang, Jiyuan Wang, Guoqing Harry Xu, and Miryung Kim. 2022. HeteroGen: transpiling C to heterogeneous HLS code with automated test generation and program repair. In *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 1017–1029.

[86] Yixue Zhao, Saghar Talebipour, Kesina Baral, Hyojae Park, Leon Yee, Safwat Ali Khan, Yuriy Brun, Nenad Medvidović, and Kevin Moran. 2022. Avgust: automating usage-based test generation from videos of app executions. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 421–433.

[87] Yuanhang Zhou, Fuchen Ma, Yuanliang Chen, Meng Ren, and Yu Jiang. 2023. CLFuzz: Vulnerability Detection of Cryptographic Algorithm Implementation via Semantic-aware Fuzzing. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–28.

[88] Zhichao Zhou, Yuming Zhou, Chunrong Fang, Zhenyu Chen, and Yutian Tang. 2022. Selectively Combining Multiple Coverage Goals in Search-Based Unit Test Generation. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 91:1–91:12.

Received 2024-04-12; accepted 2024-07-03