A Roadmap for Software Testing in Open-Collaborative and AI-Powered Era

QING WANG, JUNJIE WANG, MINGYANG LI, YAWEN WANG, and ZHE LIU, Institute of Software Chinese Academy of Sciences, Beijing, China

Internet technology has given rise to an open-collaborative software development paradigm, necessitating the open-collaborative schema to software testing. It enables diverse and globally distributed contributions, but also presents significant challenges to efficient testing processes, coordination among personnel, and management of testing artifacts. At the same time, advancements in AI have enhanced testing capabilities and enabling automation, while also introducing new testing needs and unique challenges for AI-based systems. In this context, this article explores software testing in the open-collaborative and AI-powered era, focusing on the interrelated dimensions of process, personnel, and technology. Among them, process involves managing testing workflows and artifacts to improve efficiency, personnel emphasizes the role of individuals in ensuring testing quality through collaboration and contributions, while technology refers to AI methods that enhance testing capabilities and address challenges in AI-based systems. Furthermore, we delve into the challenges and opportunities arising from emerging technologies such as Large Language Models (LLMs) and the AI model-centric development paradigm.

CCS Concepts: • **Software and its engineering → Software creation and management**;

Additional Key Words and Phrases: Software Testing, Artificial Intelligence, AI, Large Language Model, LLM, Open Source, Open Collaborative

ACM Reference format:
Qing Wang, Junjie Wang, Mingyang Li, Yawen Wang, and Zhe Liu. 2025. A Roadmap for Software Testing in Open-Collaborative and AI-Powered Era. *ACM Trans. Softw. Eng. Methodol.* 34, 5, Article 148 (May 2025), 17 pages.
https://doi.org/10.1145/3709355

All authors are also with State Key Laboratory of Intelligent Game, Beijing, China and University of Chinese Academy of Sciences, Beijing, China.
This work was supported by the National Natural Science Foundation of China Grant Nos. 62232016, 62072442, 62402484, 62402483, Youth Innovation Promotion Association CAS, Basic Research Program of ISCAS Grant No. ISCAS-JCZD-202304, Major Program of ISCAS Grant Nos. ISCAS-ZD-202302 and ISCAS-ZD-202401, Innovation Team 2024 ISCAS (No. 2024-66).
Authors' Contact Information: Qing Wang (corresponding author), Institute of Software Chinese Academy of Sciences, Beijing, China; e-mail: wq@iscas.ac.cn; Junjie Wang (corresponding author), Institute of Software Chinese Academy of Sciences, Beijing, China; e-mail: junjie@iscas.ac.cn; Mingyang Li, Institute of Software Chinese Academy of Sciences, Beijing, China; e-mail: mingyang2017@iscas.ac.cn; Yawen Wang, Institute of Software Chinese Academy of Sciences, Beijing, China; e-mail: yawen2018@iscas.ac.cn; Zhe Liu, Institute of Software Chinese Academy of Sciences, Beijing, China; e-mails: liuzhe181@mails.ucas.edu.cn, liuzhe2020@iscas.ac.cn.

ACM 1557-7392/2025/5-ART148
https://doi.org/10.1145/3709355

1 Introduction

Open collaboration is a hallmark of the Internet era in the past 10 to 20 years, where geographical distances cease to be a barrier. Platforms like social media networks and online forums have emerged as global hubs for communication, enabling individuals worldwide to exchange ideas, share experiences, and engage in discussions [19]. Similarly, in the realm of software development, this has led to the rise of the open-collaborative development paradigm [10]. Platforms like GitHub have flourished, providing developers from diverse backgrounds with the opportunity to contribute code, discuss technical matters, and address software issues, regardless of their geographic location [6, 46]. In such development environments, software testing and quality assurance faces both new challenges and opportunities. Unlike the closed-environment testing paradigm with fixed contributors and pre-defined setups, open-environment testing faces the challenges of distributed collaboration, diverse contributors, and rapid iterations, often resulting in uneven outcomes. This variability necessitates not only a focus on testing techniques but also on the testing process itself, such as improving coordination of testing activities to handle variability and streamlining the management of testing artifacts to achieve better cost-effectiveness. Additionally, attention must also be given to the personnel involved, providing them with the support needed to enhance their efficiency.

Another significant shift is the rise of AI technologies. From the early days of machine learning and **Deep Learning (DL)** to the advent of pre-trained **Large Language Models (LLMs)**, these advancements offer effective solutions to address the challenges inherent in open-collaborative testing.

There exist notable roadmaps that have significantly contributed to the understanding and advancement of software testing practices. One such landmark roadmap is "Software Testing: A Research Travelogue (2000–2014)," spearheaded by Professor Alessandro Orso and Gregg Rothermel [43]. This seminal work primarily explores software testing techniques and methodologies, offering valuable insights into the evolution of testing technologies over the past decade and a half.

However, a decade has passed since then, and with the constant emergence of new technologies, there is a pressing need for a new roadmap to summarize and project future research in software testing. Moreover, we aim to adopt a broader perspective, recognizing that the factors influencing software testing extend well beyond the techniques alone. Therefore, a comprehensive overview and roadmap for software testing in the open-collaborative and AI-powered era requires a holistic understanding that encompasses not only testing methodologies but also broader contextual factors.

Taken in this sense, our exploration of software testing in the open-collaborative and AI-powered era is guided by the recognition that three key dimensions—process, personnel, and technology—form a triad of fundamental factors influencing software testing practices. These dimensions represent orthogonal facets that collectively shape the landscape of software testing, as visually demonstrated in Figure 1. Among them, the *process* dimension focuses on methodologies to manage dynamic, distributed contributions efficiently and ensure timely, comprehensive testing coverage. The *personnel* dimension highlights the critical role of human contributions in identifying and resolving issues to maintain testing quality. The *technology* dimension emphasizes the impact of AI advancements in enhancing testing capabilities and addressing the unique challenges of AI-driven systems. By examining these three dimensions—process, personnel, and technology—we gain a comprehensive understanding of the intricate landscape of software testing within open-collaborative environments.

Additionally, we also outline future trends. As emerging technologies like LLMs continue to evolve, the landscape of software testing research is expected to expand to encompass various aspects. This may include leveraging LLMs for enhancing testing practices, exploring testing

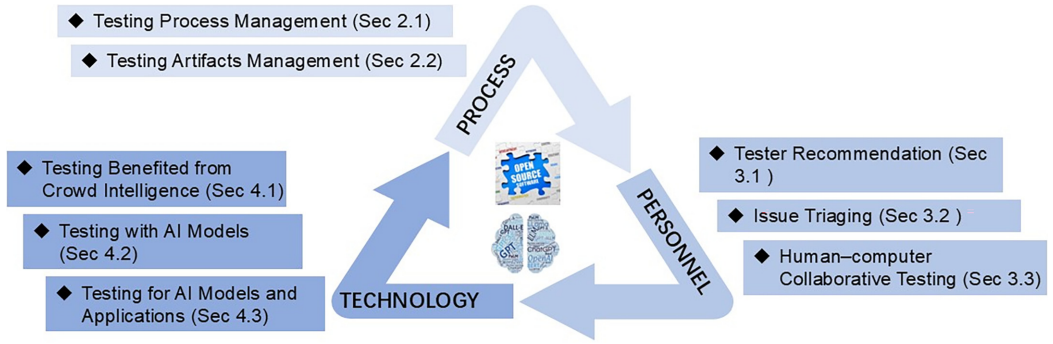

Fig. 1. Overview about software testing in open-collaborative and AI-powered era.

methodologies tailored for LLMs and related applications, ensuring quality assurance for auto-generated code produced by AI models, and refining collaboration strategies within AI model-centric paradigms. These areas represent promising avenues for future research and innovation in the field of software testing under open-collaborative and AI-powered era.

To curate the relevant works, we adopt an informal methodology informed by the authors' expertise in software testing and AI. The selection prioritized survey papers and high-impact studies from top-ranked venues in software engineering as well as AI related venues. This allowed us to quickly incorporate emerging insights into our discussion while focusing on the core challenges and opportunities related to software testing in the AI era.

The article is structured as follows: Sections 2 to 4 provide an overview of software testing researches from the perspectives of process, personnel, and technology, respectively. Section 5 explores the opportunities and challenges in this field, while Section 6 concludes the article.

2 Process Related

In open-collaborative environments, the process dimension of software testing involves practices and methodologies that enable efficient and effective testing in dynamic, distributed development settings, alongside the streamlined management of testing artifacts.

2.1 Testing Process Management

In modern software development, **Continuous Integration (CI)** has become a cornerstone practice for maintaining code quality and fostering collaboration. By frequently integrating code into a shared version control repository, CI ensures that each change is automatically validated through automated build and testing pipelines. A critical aspect of this process is regression testing, which serves as a key manifestation of the testing process by verifying that new changes do not introduce defects into existing functionality. While regression testing is invaluable for maintaining software quality, executing a full suite of tests can be highly resource-intensive and time-consuming, often requiring hours or even days to complete. This delay can hinder the CI cycle and prevent timely feedback for developers [67]. To mitigate these challenges, techniques such as **Test Case Prioritization (TCP)** and test case selection have been developed to optimize the testing process, by selecting and prioritizing test cases in order to provide early feedback to developers. We introduce the TCP techniques which share similarities with selection techniques.

Earlier attempts start with heuristics-based techniques, e.g., Elbaum et al. [16] prioritized tests based on whether they have not been executed for long or have failed in the recent commits, and Haghighatkhah et al. [20] scheduled tests based on the combination of their previous execution

results and similarity. Later, other researchers harness the power of machine learning by using a large amount of historical data in CI, and propose numerous machine learning-based TCP techniques which have been demonstrated to be promising. They build neural models to predict the optimal sequence of tests instead of human-defined strategies. In particular, these machine learning-based techniques can be categorized into supervised learning-based [5, 7, 47, 67], and reinforcement learning-based techniques [4, 28, 50]. The latter ones would continuously adjust its prioritization strategy, i.e., it is first tested (i.e., prioritizing the test suite) and then trained based on the prioritization feedback.

Apart from that, in open-collaborative environments, crowdsourced testing (also typically abbreviated as crowdtesting) has become a key strategy for enhancing software quality by leveraging contributions from a large, diverse group of external testers. Crowdtesting is hard to manage in nature. Given the unpredictability of distributed crowdtesting processes, it is difficult to estimate (a) remaining number of bugs yet to be detected or (b) required cost to find those bugs. Experience-based decisions may result in ineffective crowdtesting processes, e.g., there is an average of 32% wasteful spending in current crowdtesting practices. Wang et al. [58] explored automated decision support to effectively manage crowdtesting processes.

2.2 Testing Artifacts Management

In open-collaborative environments, the testers—often from different backgrounds and geographical locations—can provide valuable feedback by identifying issues that might not be caught through traditional testing methods. However, the volume of contributions from such a wide range of sources presents unique challenges. The influx of issue reports, bug findings, and feedback (all of them can be treated as testing artifacts) can lead to information overload, where valuable insights are buried under redundant or conflicting contributions. This makes it difficult to prioritize and filter out irrelevant data, which can undermine the efficiency of the testing process. Effective information filtering techniques are therefore necessary to improve the signal-to-noise ratio and streamline the testing process. Duplicate detection is the most-commonly employed technique, which aims at identifying and eliminating redundant or duplicate information, such as duplicate issue reports or discussions on similar topics. For example, Nguyen et al. [41] applied information retrieval techniques for duplicate detection by computing the textual similarity between two reports. Sun et al. [51] designed a set of features for measuring the reports' similarity in terms of textual descriptions and attributes, and employed machine learning techniques for duplicate detection. Yang et al. [64] modeled the semantic similarity of reports using DL techniques for duplicate detection. In addition, Huang et al. [23] and Zhang et al. [73] respectively conducted experimental evaluations of the commonly used approaches for duplicate detection.

3 Personnel Related

In open-collaborative software testing, the involvement of individuals plays a crucial role in the success of the testing process. Human factors influence the quality of testing, as contributions and collaborations from testers, developers, and stakeholders are vital for identifying critical issues, quickly fix the issues, and ensuring comprehensive coverage.

3.1 Tester Recommendation

Unlike typical software development projects, testing tasks typically require a group of testers, ideally with diverse backgrounds, to achieve the goal of diversified testing and covering different areas of the software. Therefore, different from the worker recommendation for general crowdsourcing tasks, the tester recommendation for crowdtesting tasks tends to take into account the diversity of the recommended workers. For example, Wang et al. [57] propose a multi-objective

crowd tester recommendation approach, which aims at recommending crowd tester by maximizing the bug detection probability of testers, the relevance with the test task, the diversity of testers, and minimizing the test cost. Additionally, previous studies on worker recommendation mainly focus on one-time recommendations with respect to the initial context at the beginning of a new task. However, for crowdtesting, a typical task can last from 3 days to 2 weeks, during which crowd testers can freely conduct the testing and submit reports. Taking this into account, Wang et al. [59] point out the need for accelerating crowdtesting by recommending appropriate testers in a dynamic manner, and they propose a context-aware in-process crowd testers recommendation approach, to detect more bugs earlier and potentially shorten the testing period.

3.2 Issue Triaging

Various issues appear during software testing, and issue fixing is a time-consuming and costly task. Once an issue report is received, assigning it to a suitable developer within a short time interval can reduce the time and cost of the issue fixing process. This assignment process is known as issue triaging. Issue triaging is a time-consuming process since often a large number of developers are involved in software testing. To aid in finding appropriate developers, earlier practice adopted techniques as machine learning, graph analysis, fuzzy set, and topic modeling. For example, Anvik et al. [2] utilized machine learning methods to solve it. Jeong et al. [24] proposed to use a bug tossing graph to improve issue triaging prediction accuracy. Tamrawi et al. [54] proposed a method called Bugzie, which uses a fuzzy set and cache-based approach to increase the accuracy of issue triaging. Xia et al. [63] proposed a specialized topic modeling algorithm named multi-feature topic model for issue triaging. Later studies used DL for bug triaging, e.g., Lee et al. [27] applied word embedding to train a CNN-based classifier. Dipongkor et al. [14] conducted the experimental evaluation about fine-tuning the transformer-based language models for this task.

3.3 Human–Computer Collaborative Testing

There are studies that analyze incorporating automation technologies to assist manual testing, exemplifying human–computer interaction in the software testing process. For instance, Liu et al. automatically trace testers' actions and use explicit visual annotations to guide or remind them of unexplored areas, helping testers avoid missing functionalities or repeating steps [34]. Similarly, Chen et al. investigate human collaboration in crowd testing scenarios. They utilize interactive event-flow graphs to track and aggregate each tester's interactions into a single directed graph, which visualizes the test cases already explored. Crowd testers can interact with these graphs to discover new navigation paths and improve test coverage [9].

4 Technology Related

Advancements in AI technology have significantly enhanced testing capabilities, introducing intelligent automation and improving efficiency in open-collaborative environments. At the same time, the rise of AI-driven applications and systems has created new demands for testing, necessitating specialized techniques to validate the functionality, robustness, and fairness of AI models and applications. These dual developments highlight the evolving role of technology in both strengthening traditional testing processes and addressing the unique challenges posed by AI-centric systems.

4.1 Testing Benefited from Crowd Intelligence

In the context of open-collaborative software development, a vast amount of data contributed by developers with diverse backgrounds is aggregated. These data encapsulate rich knowledge about software quality assurance, and can be harnessed to empower the testing techniques. For example, Mao et al. [39] extracted the test scripts from crowdbased testing to automatically infer the reusable

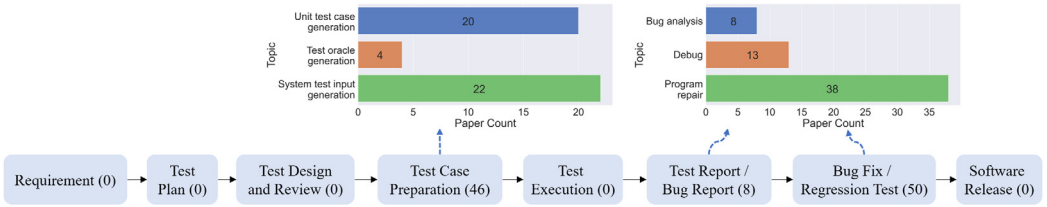

Fig. 2. Distribution of testing tasks with LLMs [56].

high-level event sequences for enhancing the automated mobile testing. Liu et al. [33] collected a large number of GUI screenshots with UI display issues from the crowdsourced testing platform, and used them to train a visual understanding model for automatically detecting the GUIs with display issues and locating the buggy region. Wei et al. [61] mined the code/models from open source to obtain the code snippets from the DL library documentation, library developer tests, and DL models in the wild, then leveraged this information to perform fuzz testing for DL libraries.

4.2 Testing with AI Models

Since the introduction of the concept of training "Deep Neural Network" by Geoffrey Hinton in 2006, DL has been increasingly adopted to develop cutting-edge tools for software testing research, thanks to its ability to enhance performance. A comprehensive survey [65] explores how DL is applied in testing areas such as test case generation, bug localization, and application testing.

In recent years, the pervasive advancements in LLMs have profoundly impacted various domains, including software testing. These models have been increasingly harnessed to bolster testing capabilities across different facets, ranging from enhancing test coverage in unit testing to diversifying test case generation in integration testing. There is a relevant literature review titled "Software Testing with Large Language Models: Survey, Landscape, and Vision" [56], which provides a comprehensive overview of the utilization of LLMs in software testing. It analyzes 102 relevant studies that have employed LLMs for software testing, examining them from both the software testing and LLMs perspectives. As demonstrated in Figure 2, LLMs are commonly used for test case preparation (including unit test case generation, test oracle generation, and system test input generation), program debugging, and bug repair. However, there is currently no practices for applying LLMs in the tasks of early testing lifecycle (such as test requirement, test plan, etc.).

For unit test case generation, a majority of the earlier published studies adopt the pre-training or fine-tuning schema. For example, Alagarsamy et al. [1] first pre-trained the LLM with the focal method and asserted statements to enable the LLM to have a stronger foundation knowledge of assertions. They then fine-tuned the LLM for the test case generation task where the objective is to learn the relationship between the focal method and the corresponding test case. By comparison, most later studies typically focus on how to design the prompt, to make the LLM better understanding the context of the task. Yuan et al. [68] performed an empirical study to evaluate ChatGPT's capability of unit test generation with both a quantitative analysis and a user study in terms of correctness, sufficiency, readability, and usability. And results show that the generated tests still suffer from correctness issues, including diverse compilation errors and execution failures. To address this, they proposed an approach where ChatGPT was used to improve the quality of its own generated tests, using an initial test generator and an iterative test refiner. The iterative refiner followed a validate-and-fix approach, correcting compilation errors by prompting the LLM based on error messages and additional code context.

For system test input generation, it varies for specific types of software being tested. For mobile applications, test input generation requires a wide range of text inputs or operation combinations (e.g., clicking a button or long-pressing a list) to test the application's functionality and user interface [31, 32]. In contrast, for DL libraries, the test input consists of programs that cover diverse DL APIs [12, 13]. For example, Liu et al. [32] formulate the test input generation of mobile GUI testing problem as a Q&A task, which asks LLM to chat with the mobile apps by passing the GUI page information to the LLM. The LLM then generates testing scripts (i.e., GUI operations) and executes them while receiving app feedback, iterating the process. The proposed GPTDroid also introduces a functionality-aware memory prompting mechanism that equips the LLM with the ability to retain testing knowledge of the whole process and conduct long-term functionality-based reasoning to guide exploration. Deng et al. [13] used both generative and infilling LLMs to generate and mutate valid/diverse input DL programs for fuzzing DL libraries. The process starts with a generative LLM (CodeX) to generate a set of seed programs using target DL APIs. Next, part of the seed program is replaced with masked tokens, and an infilling LLM (InCoder) is used to fill in the masked tokens and generate new code.

4.3 Testing for AI Models and Applications

The rise of AI applications raises concerns about trustworthiness, particularly in safety-critical domains such as self-driving systems and medical treatments. Software testing plays a crucial role in detecting and addressing discrepancies between expected and actual behaviors in these applications. However, testing AI systems presents unique challenges due to their statistical nature, evolving behavior, and the oracle problem.

The notable systematic review titled "Machine Learning Testing: Survey, Landscapes and Horizons" [71] presents an extensive examination of methodologies for assessing machine learning (including DL) systems. It encompasses 144 papers that explore various aspects of testing properties (such as correctness, robustness, and fairness), testing components (including data, learning programs, and frameworks), workflows (encompassing test generation and evaluation), and application scenarios (such as autonomous driving and machine translation), as shown in Figure 3.

Adversarial inputs represent a critical concept within AI testing, as they play a significant role in assessing the robustness of AI models. These inputs are deliberately perturbed based on the original inputs, often deviating from the typical data distribution encountered in real-world scenarios. By subjecting models to these carefully crafted inputs, testers can identify potential weaknesses and shortcomings of AI models. For example, Zhou et al. [74] proposed DeepBillboard to generate real-world adversarial billboards that can trigger potential steering errors of autonomous driving systems. Sun et al. [53] automatically generate test inputs via mutating the words in translation inputs for testing machine translation systems. In order to generate translation pairs that ought to yield consistent translations, their approach conducts word replacement based on word embedding similarities.

The test oracle problem is challenging, because many machine learning algorithms are probabilistic programs. Metamorphic relations are widely studied to tackle the oracle problem, and they are based on transformations of training or test data that are expected to yield unchanged or certain expected changes in the predictive output. For example, Dwarakanath et al. [15] applied metamorphic relations to image classifications with SVM and DL systems. The changes on the data include changing the feature or instance orders, linear scaling of the test features, normalization or scaling up the test data, or changing the convolution operation order of the data. Tian et al. [55] and Zhang et al. [72] stated that the autonomous vehicle steering angle should not change significantly or stay the same for the transformed images under different weather conditions.

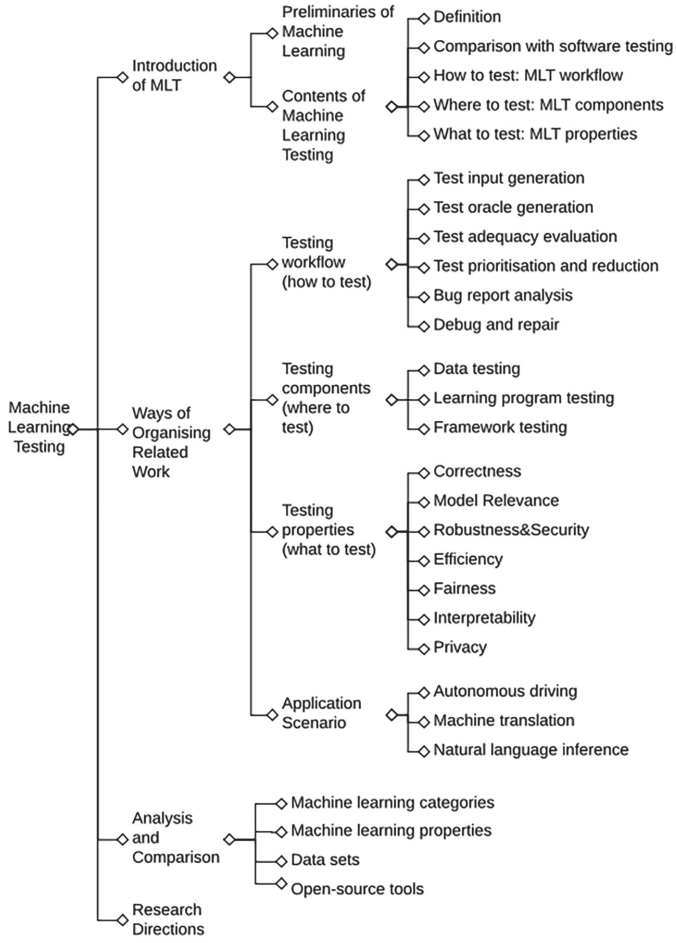

Fig. 3. Details of machine learning testing [71].

Test adequacy evaluation aims to discover whether the existing tests have a good fault-revealing ability. It provides an objective confidence measurement on testing activities. In traditional software testing, code coverage measures the degree to which the source code of a program is executed by a test suite [70]. Unlike traditional software, code coverage is seldom a demanding criterion for AI testing, since the decision logic of an AI model is not written manually but rather it is learned from training data. Pei et al. [45] proposed the first coverage criterion, neuron coverage, particularly designed for DL testing. Neuron coverage is calculated as the ratio of the number of unique neurons activated by all test inputs and the total number of neurons in a DNN. In particular, a neuron is activated if its output value is larger than a user-specified threshold. Following that, there are more fine-grained criteria, like k-multi-section neuron coverage, neuron boundary coverage, and strong neuron activation coverage, and so on.

5 Challenges and Opportunities

Despite the advancements and breakthroughs discussed in the preceding sections, open-collaborative development environments and the rapid evolution of AI technologies/ecosystems continue to

present numerous challenges, highlighting ongoing research opportunities and emerging trends in the coming years.

5.1 Leveraging LLMs for Enhancing Testing

Although software testing with LLMs has undergone significant growth in the past 3 years, it is still in its early stages of development, and numerous challenges and open questions need to be addressed.

Test Case Generation. Exploring diverse software behaviors while minimizing costs remains a critical challenge in software testing. Test case generation, in particular, poses significant obstacles in traditional automated testing due to the difficulty of producing test inputs with correct grammar and semantics, especially for complex or composite inputs. As a result, automated testing often struggles to achieve comprehensive coverage. LLMs offer promising potential with their exceptional ability to understand software context, yet current research has barely scratched the surface of their capabilities. For example, for unit test case generation, in SF110 dataset, the line coverage is merely 2% [49]. For system test input generation, in terms of fuzzing DL libraries, the API coverage for TensorFlow is reported to be 66% (2,215/3,316) [13].

As the initial wave of enthusiasm for LLMs has subsided, people have become increasingly aware of the limitations of LLMs and the strengths of program analysis and traditional testing techniques. Consequently, the integration of LLMs with traditional approaches has emerged as an important direction for future exploration. One direction may utilize mutation testing together with the LLMs to generate more diversified outputs. For example, when fuzzing a DL library, instead of directly generating the code snippet with LLM, Deng et al. [13] replace parts of the selected seed (code generated by LLM) with masked tokens using different mutation operators to produce masked inputs. They then leverage the LLM to perform code infilling to generate new code that replaces the masked tokens, which can significantly increase the diversity of the generated tests. Automatically generate test cases based on metamorphic relations to cover a wide range of inputs is also a promise avenue. Combine LLM with other traditional technique is also a promising direction, e.g., Jiang et al. [25] conduct a systematic study on the LLMs and constraint-based tools for test input generation, and find that there are limitations for LLMs in specific scenarios such as sequential calculation, where constraint-based tools are in a position of strength. By combining them together, there can be 1.4× to 2.3× improvement than the baselines.

Other potential research direction could involve utilizing testing-specific data to train or fine-tune a specialized LLM that is specifically designed to understand the nature of testing. By doing so, the LLM can inherently acknowledge the requirements of testing and autonomously generate diverse outputs.

Test Oracle Problem. The oracle problem has been a longstanding challenge in various testing applications, e.g., testing machine learning systems [71] and testing DL libraries [13]. To alleviate the oracle problem to the overall testing activities, a common practice is to transform it into a more easily derived form, often by utilizing differential testing [52] or focusing on only identifying crash bugs [32].

Exploring the use of LLMs to derive other types of test oracles represents an interesting and valuable research direction. Specifically, metamorphic testing is also widely used in software testing practices to help mitigate the oracle problem, yet in most cases, defining metamorphic relations relies on human ingenuity. Recent preliminary attempts have demonstrated the potential of leveraging LLMs to automatically discover and define these relations, offering a promising avenue for further investigation and application [38]. Another promising avenue is exploring the capability of LLMs to automatically generate test cases based on metamorphic relations, covering a wide range of inputs. Apart from that, the advancement of multi-model LLMs like GPT-4 may open up possibilities for exploring their ability to detect bugs in software user interfaces and assist in deriving test

oracles. By leveraging the image understanding and reasoning capabilities of these models, one can investigate their potential to automatically identify inconsistencies, errors, or usability issues in user interfaces, e.g., VisionDroid [35] makes the first attempt towards this direction.

Real-World Application of LLMs in Software Testing. Due to concerns regarding data privacy, when considering real-world practice, most software organizations tend to avoid using commercial LLMs and would prefer to adopt open source ones with training or fine-tuning using organization-specific data. Furthermore, some companies also consider the current limitations in terms of computational power or pay close attention to energy consumption, they tend to fine-tune medium-sized models. It might be quite challenging for these models to achieve similar performance to what existing papers have reported. Recent research has highlighted the importance of high-quality training data in improving the performance of models for code-related tasks. A notable example in this area is the StarCoder2 model [36], where the StarCoder2-15B significantly outperforms other models of similar size (e.g., CodeLlama-13B) and even matches or surpasses the performance of CodeLlama-34B. The superior performance of StarCoder2 is reported as attributing to the curation of high-quality open data sources, such as GitHub issues, pull requests, Kaggle datasets, Jupyter notebooks, and code documentation. Additionally, rigorous data pre-processing steps, including deduplication and the application of filters to eliminate low-quality code, have played a crucial role in enhancing the model's effectiveness. However, building high-quality, organization-specific datasets for training or fine-tuning is a time-consuming and labor-intensive process. To address this challenge, automated techniques from the field of mining software repositories [21]—an area that has made significant strides over the past decade—can be employed. These techniques enable efficient extraction and analysis of key information, streamlining the dataset creation process.

In addition, exploring methodologies for better pre-training or fine-tuning LLMs with software-specific data is a promising direction. In recent times, the approach to LLM pre-training and fine-tuning has largely followed the "more data, better results" philosophy, with an emphasis on utilizing as much data as possible. However, the availability of open data is limited, and this path has nearly reached its limits. On the other hand, software-specific data differ from natural language data in that it contains more structural information, such as data flow and control flow. Therefore, enabling LLMs to better learn these software-specific structural elements may become a key focus in future research. Previous work on code representations has highlighted the benefits of incorporating data flow information, as demonstrated by Guo et al. [2021] in their GraphCodeBERT model.

5.2 Testing for LLMs and LLM-Centric Applications

Since the emergence of LLMs, AI-enabled software applications have rapidly advanced. The quality of these applications now relies not only on functional correctness but also on the performance of the embedded or associated AI models. This shift has introduced unprecedented challenges to testing.

Testing Methodology Specifically Designed for LLMs. There have been numerous research efforts on testing machine learning and DL models; however, in the context of general artificial intelligence, i.e., LLMs, there has been relatively less exploration in software-related conferences and journals. At AI-related conferences, much work has been done on benchmarking LLMs, such as evaluating their performance in task automation [60], instruction tuning [48], and judge assistants [8]. These benchmarking efforts have significantly advanced the field, with some works already having hundreds of citations. From the perspective of software engineering, a systematic testing methodology specifically designed for LLMs is urgently needed. It's essential to extend testing beyond basic benchmarks and include functional, non-functional, and safety testing, especially considering the complex and unpredictable behaviors of LLMs when applied in real-world software applications. Furthermore, with the emergence of multi-modal LLMs, there is a need for more research and attention on testing such models.

Traditional software testing techniques face several challenges due to the non-deterministic nature, complex input structures, and the lack of transparency in LLMs' decision-making processes. While techniques like fuzz testing and metamorphic testing may be adapted and extended to these scenarios, they still do not fully address the core complexities of LLMs. These methods may help detect specific issues but often fail to tackle the underlying dynamics of model behavior, such as how models handle ambiguous or unseen inputs. Therefore, there is room for more novel approaches and ideas in testing LLMs, potentially requiring entirely new paradigms that go beyond traditional methods to better evaluate their performance and reliability in diverse, real-world tasks.

Testing LLM-as-Agent Systems. Whether for classification tasks or generation task, LLMs remain far from mature. Yet, LLM-powered applications have already proliferated, with the LLM-as-agent paradigm [30, 62] being a prominent example. In this scenario, LLMs handle tasks such as cognitive understanding and decision-making to support specific applications. This paradigm extends beyond the LLMs themselves, incorporating external components like environments, memory modules for storing interaction history, external knowledge bases for retrieving up-to-date information, and tools or services for task execution. Testing the performance of LLM-as-agent systems requires evaluating both traditional software metrics (e.g., response speed, fault tolerance, and reliability) and AI-specific aspects (e.g., compliance, robustness, generalization, trustworthiness, and fairness).

Apart from that, with recent advancements, such as Claude 3.5's computer use and AutoGLM, LLM agents are demonstrating greater autonomy, more sophisticated tool usage, and emerging visual capabilities, evolving toward LLM-centered operating systems. Testing in this context involves verifying the model's ability to effectively utilize external resources, adapt to dynamic environments, and maintain coherence across complex tasks. Additionally, testing strategies must account for challenging scenarios, including handling unexpected inputs, adapting to novel situations, and recovering gracefully from errors or disruptions. By adopting a holistic testing approach, developers can ensure the functionality and reliability of LLM-as-agent systems in diverse real-world scenarios.

Quality Assurance of LLM Store (E.g., GPT Store). The first two items in this subsection primarily discuss testing from the perspectives of LLMs and LLM agents. Here, the discussion will be expanded from the perspective of the LLM ecosystem. GPTs, as a new form of service based on LLMs, will make the GPT Store a new channel for people to access applications, similar to the Google Play Store or the Apple App Store in the era of mobile applications. They are custom versions of GPT tailored for specific purposes, allowing users to create personalized iterations of GPT to better suit their needs [42]. These customized GPTs can be designed for various tasks, such as teaching children math, providing assistance in board games, or generating stickers. In the context of the mobile app market, tasks related to ensuring the quality and reliability of apps, such as malware detection, privacy violation detection, and sensitive data leaks detection, remain critical for GPTs. However, due to the differences between LLMs and mobile apps, new testing techniques are urgently needed to address these challenges. Moreover, discrepancies between the descriptions of GPT capabilities and their actual performance may arise, necessitating robust testing procedures to ensure consistency and accuracy. Additionally, leveraging user-contributed feedback and reviews on the GPT Store platform could serve as a valuable resource for identifying defects and improving GPT performance, which can take inspiration from previous researches on mining mobile app reviews [11, 17].

5.3 Testing and Quality Assurance for Auto-Generated Code

With the advancement of LLMs and their remarkable performance in code generation tasks, developers are increasingly relying on these models for various coding and debugging tasks. Recently, there have been reports of groundbreaking developments, such as Microsoft's creation of an AI programmer named AutoDev [40], capable of mastering full-stack skills, which can not only write code and debug but also train models and even bid for projects on the largest freelancing platform, Upwork.

Automatic code generation is rapidly gaining traction, yet testing practices in this area lag behind. Currently, most approaches utilize code-focused LLMs to assist engineers in writing code, functioning similarly to pair programming in agile methodologies. In this collaborative dynamic, LLMs generate code while engineers review it, relying primarily on traditional testing methods. However, this mismatch between human-driven workflows and AI-driven automation hinders efficiency. Furthermore, challenges such as LLM hallucinations, data poisoning, and other vulnerabilities introduce significant and often hidden risks to code quality.

Functional Concerns of Auto-Generated Code. The integration of AI-generated code into open source projects raises concerns regarding code reliability. While AI models demonstrate impressive capabilities in code generation, there remain uncertainties about the robustness and correctness of the generated code [44]. Liu et al. explore and evaluate the hallucinations in LLM-powered code generation, and categories them into intent conflicting (e.g., local semantic conflicting), context deviation (e.g., generate repetitive statements), and knowledge conflicting (e.g., using un-imported library) [29]. It is challenging for LLMs to detect and correct hallucinations through prompting. Therefore, it is crucial to develop specialized techniques for detecting and mitigating these issues. There have been some attempts targeted at specific tasks [37], but they have yet to achieve satisfactory results.

Non-Functional Concerns of Auto-Generated Code. In addition to functional correctness, the non-functional requirements of automatically generated code are also critically important. For example, Zhang et al. proposed EffiBench, a benchmark for assessing the efficiency of automatically generated code [22]. Similarly, maintainability and security concerns, as highlighted by Asare et al. [3], are essential aspects of software quality. Developing robust tools and benchmarks to address these non-functional requirements is crucial for ensuring the reliability and practicality of AI-generated code in real-world applications.

In addition to the inherent weaknesses of LLMs that may result in insecure code, external attackers can also introduce security risks into automatically generated code. For example, model publishers might embed malicious backdoors, causing the model to perform normally on standard inputs but generate harmful outputs (i.e., insecure code) when exposed to specific triggered inputs [66]. These malicious code could lead to data theft, unauthorized software behavior, or other security breaches. Therefore, testing for automatically generated code must also account for such scenarios to ensure robustness against these threats.

Testing in Terms of System-Level Auto-Generated Code. The lack of human oversight in the code generation process may lead to the introduction of low-quality code that could compromise the integrity of software systems. The sheer volume of AI-generated contributions necessitates scalable testing processes to ensure compliance with coding standards, adheres to best practices, and complies with project-specific requirements. To meet these needs, it is crucial to develop testing frameworks specifically tailored to address the nuances of auto-generated code. These frameworks should integrate static analysis tools, automated test generation, and dynamic testing techniques to identify and resolve potential issues effectively. Moreover, fostering collaboration between AI developers, software engineers, and open source maintainers is essential to facilitate the seamless integration of AI-generated contributions while upholding the quality and reliability of open source software projects.

5.4 Collaborative Testing between Human and AI

Re-Define Communication and Collaboration between Humans and AI Systems. As described in Section 3, previous studies on human and AI collaboration mainly focused on using AI-related technologies to provide intelligent services for humans, such as recommending suitable issue reports. However, with the advancement of technologies like LLMs, AI's capabilities have become

more prominent, leading to the emergence of many AI-powered techniques and even AI-powered testers. Therefore, we need to re-define communication and collaboration between humans and AI systems during the software testing and quality assurance process. A widely accepted notion is the human-in-the-loop methodology.

In fact, there are already relevant implementations in the field of software testing. For instance, Zamprogno et al. applied it to test assertion generation, where the developer selects the variables they want assertions for, the tool generates assertions only for these variables, and the developer evaluates the relatively small number of generated assertions, ensuring that only useful assertions are persisted in their test cases [69]. Similarly, Geethal et al. applied it to program repair, using it in conjunction with active learning techniques to present test cases with a higher probability of being labeled as failing to the human [18]. These methods have achieved better performance compared to purely automated techniques.

Meanwhile, the launch of LangGraph, a library to help developers build multi-actor, multi-step, stateful LLM applications, has already supported two "human-in-the-loop" features in OpenGPTs: Interrupt and Authorize [26]. The first mode, Interrupt, is the simplest form of control—users monitor the streaming output of the application and manually interrupt it when they deem necessary. The second control mode is Authorize, where users pre-define that the application should hand off control to them whenever a particular actor is about to be called. This underscores the significance of human involvement, and these two modes can inspire the creation of other interaction patterns between humans and AI systems for software testing applications.

6 Conclusion

The open-collaborative software development paradigm, empowered by Internet technology, has significantly transformed the landscape of software testing. This article explores the interconnected dimensions of process, personnel, and technology in the context of modern software testing. It also examines the challenges and opportunities presented by the emergence of LLMs.

In fact, since the release of ChatGPT on 30 November 2022, the AI field has undergone profound changes. What seemed like forward-thinking ideas just a month ago may now appear commonplace due to the rapid emergence of new technologies. In this era of accelerated productivity, it is difficult to predict what will happen in the next 5 to 10 years. Returning to the field of software engineering and software testing, the development of AI is rapidly transforming related research tasks, and we have witnessed how some research areas that were highly relevant just 3 years ago have now become less frequently discussed. This roadmap is an attempt to reflect on and analyze the past from the vantage point of the present, offering insights into the challenges and opportunities ahead based on our experience. It is clear that software testing will make significant strides in the future, but no one can predict with certainty what those advancements will look like, which is precisely what makes this era so exciting.

References

[1] Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. 2023. A3Test: Assertion-augmented automated test case generation. arXiv:2302.10352. Retrieved from https://arxiv.org/abs/2302.10352

[2] John Anvik, Lyndon Hiew, and Gail C. Murphy. 2006. Who should fix this bug? In *28th International Conference on Software Engineering*, 361–370.

[3] Owura Asare, Meiyappan Nagappan, and N. Asokan. 2023. Is GitHub's copilot as bad as humans at introducing vulnerabilities in code? *Empirical Software Engineering* 28, 6 (2023), 129.

[4] Mojtaba Bagherzadeh, Nafiseh Kahani, and Lionel C. Briand. 2022. Reinforcement learning for test case prioritization. *IEEE Transactions on Software Engineering* 48, 8 (2022), 2836–2856. DOI: https://doi.org/10.1109/TSE.2021.3070549

[5] Antonia Bertolino, Antonio Guerriero, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. 2020. Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration. In *42nd International Conference*

on Software Engineering (ICSE '20). Gregg Rothermel and Doo-Hwan Bae (Eds.), ACM, 1–12. DOI: https://doi.org/10.1145/3377811.3380369

[6] Grady Booch and Alan W. Brown. 2003. Collaborative development environments. *Advanced Computing* 59, 1 (2003), 1–27.

[7] Benjamin Busjaeger and Tao Xie. 2016. Learning for test prioritization: An industrial case study. In *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '16)*. Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.), ACM, 975–980. DOI: https://doi.org/10.1145/2950290.2983954

[8] Dongping Chen, Ruoxi Chen, Shilin Zhang, Yinuo Liu, Yaochen Wang, Huichi Zhou, Qihui Zhang, Yao Wan, Pan Zhou, and Lichao Sun. 2024. MLLM-as-a-judge: Assessing multimodal LLM-as-a-judge with vision-language benchmark. arXiv:2402.04788. Retrieved from https://arxiv.org/abs/2402.04788

[9] Yan Chen, Maulishree Pandey, Jean Y. Song, Walter S. Lasecki, and Steve Oney. 2020. Improving crowd-supported GUI testing with structural guidance. In *2020 CHI Conference on Human Factors in Computing Systems*, 1–13.

[10] Kevin Crowston, Kangning Wei, James Howison, and Andrea Wiggins. 2008. Free/Libre open-source software development: What we know and what we do not know. *ACM Computing Surveys* 44, 2 (2008), 1–35.

[11] Jacek Dąbrowski, Emmanuel Letier, Anna Perini, and Angelo Susi. 2022. Analysing app reviews for software engineering: A systematic literature review. *Empirical Software Engineering* 27, 2 (2022), 43.

[12] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2023. Large language models are edge-case fuzzers: Testing deep learning libraries via FuzzGPT. In *46th IEEE/ACM International Conference on Software Engineering (ICSE '24)*, Article 70 (2023), 1–13.

[13] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2023. Large language models are zero shot fuzzers: Fuzzing deep learning libraries via large language models. In *32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23)*, 423–435.

[14] Atish Kumar Dipongkor and Kevin Moran. 2023. A comparative study of transformer-based neural text representation techniques on bug triaging. In *38th IEEE/ACM International Conference on Automated Software Engineering (ASE '23)*. IEEE, 1012–1023. DOI: https://doi.org/10.1109/ASE56229.2023.00217

[15] Anurag Dwarakanath, Manish Ahuja, Samarth Sikand, Raghotham M. Rao, R. P. Jagadeesh Chandra Bose, Neville Dubash, and Sanjay Podder. 2018. Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. In *27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '18)*. Frank Tip and Eric Bodden (Eds.), ACM, 118–128. DOI: https://doi.org/10.1145/3213846.3213858

[16] Sebastian G. Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (FSE '22). Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.), ACM, 235–245. DOI: https://doi.org/10.1145/2635868.2635910

[17] Cuiyun Gao, Jichuan Zeng, Michael R. Lyu, and Irwin King. 2018. Online app review analysis for identifying emerging issues. In *40th International Conference on Software Engineering*, 48–58.

[18] Charaka Geethal, Marcel Böhme, and Van-Thuan Pham. 2023. Human-in-the-loop automatic program repair. *IEEE Transactions on Software Engineering* 49, 10 (2023), 4526–4549.

[19] Anatoliy Gruzd and Caroline Haythornthwaite. 2013. Enabling community through social media. *Journal of Medical Internet Research* 15, 10 (2013), e248.

[20] Alireza Haghighatkhah, Mika Mäntylä, Markku Oivo, and Pasi Kuvaja. 2018. Test prioritization in continuous integration environments. *Journal of Systems and Software* 146 (2018), 80–98. DOI: https://doi.org/10.1016/J.JSS.2018.08.061

[21] Ahmed E. Hassan. 2008. The road ahead for mining software repositories. In *2008 Frontiers of Software Maintenance*. IEEE, 48–57.

[22] Dong Huang, Jie M. Zhang, Yuhao Qing, and Heming Cui. 2024. EffiBench: Benchmarking the efficiency of automatically generated code. arXiv:2402.02037. DOI: https://doi.org/10.48550/ARXIV.2402.02037

[23] Yuekai Huang, Junjie Wang, Song Wang, Zhe Liu, Yuanzhe Hu, and Qing Wang. 2020. Quest for the golden approach: An experimental evaluation of duplicate crowdtesting reports detection. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '20)*. Maria Teresa Baldassarre, Filippo Lanubile, Marcos Kalinowski, and Federica Sarro (Eds.), ACM, 17:1–17:12. DOI: https://doi.org/10.1145/3382494.3410694

[24] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. 2009. Improving bug triage with bug tossing graphs. In *7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 111–120.

[25] Zongze Jiang, Ming Wen, Jialun Cao, Xuanhua Shi, and Hai Jin. 2024. Towards understanding the effectiveness of large language models on directed test input generation. In *39th IEEE/ACM International Conference on Automated Software Engineering*, 1408–1420.

[26] langchain. 2024. Retrieved from https://blog.langchain.dev/human-in-the-loop-with-opengpts-and-langgraph/

[27] Sun-Ro Lee, Min-Jae Heo, Chan-Gun Lee, Milhan Kim, and Gaeul Jeong. 2017. Applying deep learning based automatic bug triager to industrial projects. In *2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '17)*. Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.), ACM, 926–931. DOI: https://doi.org/10.1145/3106237.3117776

[28] Jackson A. Prado Lima and Silvia Regina Vergilio. 2022. A multi-armed bandit approach for test case prioritization in continuous integration environments. *IEEE Transactions on Software Engineering* 48, 2 (2022), 453–465. DOI: https://doi.org/10.1109/TSE.2020.2992428

[29] Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, and Li Zhang. 2024. Exploring and evaluating hallucinations in LLM-powered code generation. arXiv:2404.00971. Retrieved from https://arxiv.org/abs/2404.00971

[30] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. 2023. AgentBench: Evaluating LLMs as agents. arXiv:2308.03688. DOI: https://doi.org/10.48550/ARXIV.2308.03688

[31] 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 *45th International Conference on Software Engineering (ICSE '23)*, 1355–1367.

[32] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2024. Make LLM a testing expert: Bringing human-like interaction to mobile GUI testing via functionality-aware decisions. In *the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*, Article 100 (2024), 1–13.

[33] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2021. Owl eyes: Spotting UI display issues via visual understanding. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20)*. ACM, New York, NY, 398–409. DOI: https://doi.org/10.1145/3324884.3416547

[34] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2022. Guided bug crush: Assist manual GUI testing of Android apps via hint moves. In *2022 CHI Conference on Human Factors in Computing Systems*, 1–14.

[35] Zhe Liu, Cheng Li, Chunyang Chen, Junjie Wang, Boyu Wu, Yawen Wang, Jun Hu, and Qing Wang. 2024. Vision-driven automated mobile GUI testing via multimodal large language model. arXiv:2407.03037. Retrieved from https://arxiv.org/abs/2407.03037

[36] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. StarCoder 2 and the Stack v2: The next generation. arXiv:2402.19173. Retrieved from https://arxiv.org/abs/2402.19173

[37] Junliang Luo, Tianyu Li, Di Wu, Michael Jenkin, Steve Liu, and Gregory Dudek. 2024. Hallucination detection and hallucination mitigation: An investigation. arXiv:2401.08358. Retrieved from https://arxiv.org/abs/2401.08358

[38] Quang-Hung Luu, Huai Liu, and Tsong Yueh Chen. 2023. Can ChatGPT advance software testing intelligence? An experience report on metamorphic testing. arXiv:2310.19204. Retrieved from https://arxiv.org/abs/2310.19204

[39] Ke Mao, Mark Harman, and Yue Jia. 2017. Crowd intelligence enhances automated mobile testing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 16–26. DOI: https://doi.org/10.1109/ASE.2017.8115614

[40] Microsoft. 2024. AutoDev. Retrieved from https://visualstudiomagazine.com/Articles/2024/03/20/autodev.aspx

[41] Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N. Nguyen, David Lo, and Chengnian Sun. 2012. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *IEEE/ACM International Conference on Automated Software Engineering (ASE '12)*. Michael Goedicke, Tim Menzies, and Motoshi Saeki (Eds.), ACM, 70–79. DOI: https://doi.org/10.1145/2351676.2351687

[42] OpenAI. 2024. GPTs. Retrieved from https://openai.com/blog/introducing-gpts

[43] Alessandro Orso and Gregg Rothermel. 2014. Software testing: A research travelogue (2000-2014). In *On Future of Software Engineering (FOSE '14)*. James D. Herbsleb and Matthew B. Dwyer (Eds.), ACM, 117–132. DOI: https://doi.org/10.1145/2593882.2593885

[44] Shuyin Ouyang, Jie M. Zhang, Mark Harman, and Meng Wang. 2023. LLM is like a box of chocolates: The non-determinism of ChatGPT in code generation. arXiv:2308.02828. DOI: https://doi.org/10.48550/ARXIV.2308.02828

[45] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated whitebox testing of deep learning systems. In *26th Symposium on Operating Systems Principles*. ACM, 1–18. DOI: https://doi.org/10.1145/3132747.3132785

[46] Bikram Sengupta, Satish Chandra, and Vibha Sinha. 2006. A research agenda for distributed software development. In *28th International Conference on Software Engineering*, 731–740.

[47] Aizaz Sharif, Dusica Marijan, and Marius Liaaen. 2021. DeepOrder: Deep learning for test case prioritization in continuous integration testing. In *IEEE International Conference on Software Maintenance and Evolution (ICSME '21)*. IEEE, 525–534. DOI: https://doi.org/10.1109/ICSME52107.2021.00053

[48] Yongliang Shen, Kaitao Song, Xu Tan, Wenqi Zhang, Kan Ren, Siyu Yuan, Weiming Lu, Dongsheng Li, and Yueting Zhuang. 2024. TaskBench: Benchmarking large language models for task automation. arXiv:2311.18760. Retrieved

from https://arxiv.org/abs/2311.18760

[49] Mohammed Latif Siddiq, Joanna Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. 2023. Exploring the effectiveness of large language models in generating unit tests. arXiv:2305.00418. Retrieved from https://arxiv.org/abs/2305.00418

[50] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Tevfik Bultan and Koushik Sen (Eds.), ACM, 12–22. DOI: https://doi.org/10.1145/3092703.3092709

[51] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. 2010. A discriminative model approach for accurate duplicate bug report retrieval. In *32nd ACM/IEEE International Conference on Software Engineering (ICSE '10)*. Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel (Eds.), ACM, 45–54. DOI: https://doi.org/10.1145/1806799.1806811

[52] Maolin Sun, Yibiao Yang, Yang Wang, Ming Wen, Haoxiang Jia, and Yuming Zhou. 2023. SMT solver validation empowered by large pre-trained language models. In *38th IEEE/ACM International Conference on Automated Software Engineering (ASE '23)*. IEEE, 1288–1300. DOI: https://doi.org/10.1109/ASE56229.2023.00180

[53] Zeyu Sun, Jie M. Zhang, Mark Harman, Mike Papadakis, and Lu Zhang. 2020. Automatic testing and improvement of machine translation. In *42nd International Conference on Software Engineering (ICSE '20)*. Gregg Rothermel and Doo-Hwan Bae (Eds.), ACM, 974–985. DOI: https://doi.org/10.1145/3377811.3380420

[54] Ahmed Tamrawi, Tung Thanh Nguyen, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2011. Fuzzy set and cache-based approach for bug triaging. In *19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 365–375.

[55] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated testing of deep-neural-network-driven autonomous cars. In *40th International Conference on Software Engineering (ICSE '18)*. Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.), ACM, 303–314. DOI: https://doi.org/10.1145/3180155.3180220

[56] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language model: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* (2024), 911–936.

[57] Junjie Wang, Song Wang, Jianfeng Chen, Tim Menzies, Qiang Cui, Miao Xie, and Qing Wang. 2021. Characterizing crowds to better optimize worker recommendation in crowdsourced testing. *IEEE Transactions on Software Engineering* 47, 6 (2021), 1259–1276. DOI: https://doi.org/10.1109/TSE.2019.2918520

[58] Junjie Wang, Ye Yang, Rahul Krishna, Tim Menzies, and Qing Wang. 2019. ISENSE: Completion-aware crowdtesting management. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 912–923.

[59] Junjie Wang, Ye Yang, Song Wang, Yuanzhe Hu, Dandan Wang, and Qing Wang. 2020. Context-aware in-process crowdworker recommendation. In *42nd International Conference on Software Engineering (ICSE '20)*. Gregg Rothermel and Doo-Hwan Bae (Eds.), ACM, 1535–1546. DOI: https://doi.org/10.1145/3377811.3380380

[60] Yidong Wang, Zhuohao Yu, Zhengran Zeng, Linyi Yang, Cunxiang Wang, Hao Chen, Chaoya Jiang, Rui Xie, Jindong Wang, Xing Xie, et al. 2024. PandaLM: An automatic evaluation benchmark for LLM instruction tuning optimization. arXiv:2306.05087. Retrieved from https://arxiv.org/abs/2306.05087

[61] 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 (ICSE '22)*. ACM, New York, NY, 995–1007. DOI: https://doi.org/10.1145/3510003.3510041

[62] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. 2023. The rise and potential of large language model based agents: A survey. arXiv:2309.07864. Retrieved from https://arxiv.org/abs/2309.07864

[63] Xin Xia, David Lo, Ying Ding, Jafar M. Al-Kofahi, Tien N. Nguyen, and Xinyu Wang. 2017. Improving automated bug triaging with specialized topic model. *IEEE Transactions on Software Engineering* 43, 3 (2017), 272–297. DOI: https://doi.org/10.1109/TSE.2016.2576454

[64] Xinli Yang, David Lo, Xin Xia, Lingfeng Bao, and Jianling Sun. 2016. Combining word embedding with information retrieval to recommend similar bug reports. In *27th IEEE International Symposium on Software Reliability Engineering (ISSRE '16)*. IEEE Computer Society, 127–137. DOI: https://doi.org/10.1109/ISSRE.2016.33

[65] Yanming Yang, Xin Xia, David Lo, and John Grundy. 2022. A survey on deep learning for software engineering. *ACM Computing Surveys* 54, 10s (2022), 1–73.

[66] Zhou Yang, Bowen Xu, Jie M. Zhang, Hong Jin Kang, Jieke Shi, Junda He, and David Lo. 2024. Stealthy backdoor attack for code models. *IEEE Transactions on Software Engineering* 50, 4 (2024), 721–741.

[67] Ahmadreza Saboor Yaraghi, Mojtaba Bagherzadeh, Nafiseh Kahani, and Lionel C. Briand. 2022. Scalable and accurate test case prioritization in continuous integration contexts. *IEEE Transactions on Software Engineering* 49, 4 (2022), 1615–1639.

[68] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No more manual tests? Evaluating and improving ChatGPT for unit test generation. arXiv:2305.04207. Retrieved from

https://arxiv.org/abs/2305.04207

[69] Lucas Zamprogno, Braxton Hall, Reid Holmes, and Joanne M. Atlee. 2022. Dynamic human-in-the-loop assertion generation. *IEEE Transactions on Software Engineering* 49, 4 (2022), 2337–2351.

[70] Jie Zhang, Lingming Zhang, Mark Harman, Dan Hao, Yue Jia, and Lu Zhang. 2019. Predictive mutation testing. *IEEE Transactions on Software Engineering* 45, 9 (2019), 898–918. DOI: https://doi.org/10.1109/TSE.2018.2809496

[71] Jie M. Zhang, Mark Harman, Lei Ma, and Yang Liu. 2022. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering* 48, 2 (2022), 1–36.

[72] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. In *33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*. Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.), ACM, 132–142. DOI: https://doi.org/10.1145/3238147.3238187

[73] Ting Zhang, DongGyun Han, Venkatesh Vinayakarao, Ivana Clairine Irsan, Bowen Xu, Ferdian Thung, David Lo, and Lingxiao Jiang. 2023. Duplicate bug report detection: How far are we? *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 97:1–97:32. DOI: https://doi.org/10.1145/3576042

[74] Husheng Zhou, Wei Li, Zelun Kong, Junfeng Guo, Yuqun Zhang, Bei Yu, Lingming Zhang, and Cong Liu. 2020. DeepBillboard: Systematic physical-world testing of autonomous driving systems. In *42nd International Conference on Software Engineering (ICSE '20)*. Gregg Rothermel and Doo-Hwan Bae (Eds.), ACM, 347–358. DOI: https://doi.org/10.1145/3377811.3380422

Received 29 March 2024; revised 10 December 2024; accepted 11 December 2024