







Software Testing With Large Language Models: Survey, Landscape, and Vision

Junjie Wang , Member, IEEE, Yuchao Huang , Chunyang Chen , Zhe Liu ,
Song Wang , Member, IEEE, and Qing Wang , Member, IEEE

Abstract—Pre-trained large language models (LLMs) have recently emerged as a breakthrough technology in natural language processing and artificial intelligence, with the ability to handle large-scale datasets and exhibit remarkable performance across a wide range of tasks. Meanwhile, software testing is a crucial undertaking that serves as a cornerstone for ensuring the quality and reliability of software products. As the scope and complexity of software systems continue to grow, the need for more effective software testing techniques becomes increasingly urgent, making it an area ripe for innovative approaches such as the use of LLMs. This paper provides a comprehensive review of the utilization of LLMs in software testing. It analyzes 102 relevant studies that have used LLMs for software testing, from both the software testing and LLMs perspectives. The paper presents a detailed discussion of the software testing tasks for which LLMs are commonly used, among which test case preparation and program repair are the most representative. It also analyzes the commonly used LLMs, the types of prompt engineering that are employed, as well as the accompanied techniques with these LLMs. It also summarizes the key challenges and potential opportunities in this direction. This work can serve as a roadmap for future research in this area, highlighting potential avenues for exploration, and identifying gaps in our current understanding of the use of LLMs in software testing.

Index Terms—Pre-trained large language model, software testing, LLM, GPT.

I. INTRODUCTION

SOFTWARE testing is a crucial undertaking that serves as a cornerstone for ensuring the quality and reliability of software products. Without the rigorous process of software testing, software enterprises would be reluctant to release their products into the market, knowing the potential consequences of delivering flawed software to end-users. By conducting thorough and meticulous testing procedures, software enterprises can minimize the occurrence of critical software failures, usability issues, or security breaches that could potentially lead to financial losses or jeopardize user trust. Additionally, software testing helps to reduce maintenance costs by identifying and resolving issues early in the development lifecycle, preventing more significant complications down the line [1], [2].

The significance of software testing has garnered substantial attention within the research and industrial communities. In the field of software engineering, it stands as an immensely popular and vibrant research area. One can observe the undeniable prominence of software testing by simply examining the landscape of conferences and symposiums focused on software engineering. Amongst these events, topics related to software testing consistently dominate the submission numbers and are frequently selected for publication.

While the field of software testing has gained significant popularity, there remain dozens of challenges that have not been effectively addressed. For example, one such challenge is automated unit test case generation. Although various approaches, including search-based [3], [4], constraint-based [5] or random-based [6] techniques to generate a suite of unit tests, the coverage and the meaningfulness of the generated tests are still far from satisfactory [7], [8]. Similarly, when it comes to mobile GUI testing, existing studies with random-/rule-based methods [9], [10], model-based methods [11], [12], and learning-based methods [13] are unable to understand the semantic information of the GUI page and often fall short in achieving comprehensive coverage [14], [15]. Considering these limitations, numerous research efforts are currently underway to explore innovative techniques that can enhance the efficacy of software testing tasks, among which large language models are the most promising ones.

Large language models (LLMs) such as T5 and GPT-3 have revolutionized the field of natural language processing (NLP) and artificial intelligence (AI). These models, initially

Manuscript received 15 July 2023; revised 8 February 2024; accepted 9 February 2024. Date of publication 20 February 2024; date of current version 19 April 2024. This work was supported in part by the National Natural Science Foundation of China under Grant 62232016, Grant 62072442, and Grant 62272445; in part by Youth Innovation Promotion Association Chinese Academy of Sciences, Basic Research Program of ISCAS under Grant ISCAS-JCZD-202304; and in part by Major Program of ISCAS under Grant ISCAS-ZD-202302. Recommended for acceptance by L. Mariani. (Corresponding authors: Junjie Wang; Qing Wang.)

Junjie Wang, Yuchao Huang, Zhe Liu, and Qing Wang are with State Key Laboratory of Intelligent Game, Institute of Software Chinese Academy of Sciences, University of Chinese Academy of Sciences, Beijing 100190, China (e-mail: junjie@iscas.ac.cn; yuchao2019@iscas.ac.cn; liuzhe2020@iscas.ac.cn; wq@iscas.ac.cn).

Chunyang Chen is with Technical University of Munich, D-80333 Munich, Germany (e-mail: chunyang.chen@monash.edu).

Song Wang is with York University, Toronto, ON M3J 1P, Canada (e-mail: wangsong@yorku.ca).

This article has supplementary downloadable material available at <https://doi.org/10.1109/TSE.2024.3368208>, provided by the authors.

Digital Object Identifier 10.1109/TSE.2024.3368208

pre-trained on extensive corpora, have exhibited remarkable performance across a wide range of NLP tasks including question-answering, machine translation, and text generation [16], [17], [18], [19]. In recent years, there has been a significant advancement in LLMs with the emergence of models capable of handling even larger-scale datasets. This expansion in model size has not only led to improved performance but also opened up new possibilities for applying LLMs as Artificial General Intelligence. Among these advanced LLMs, models like ChatGPT¹ and LLaMA² boast billions of parameters. Such models hold tremendous potential for tackling complex practical tasks in domains like code generation and artistic creation. With their expanded capacity and enhanced capabilities, LLMs have become game-changers in NLP and AI, and are driving advancements in other fields like coding and software testing.

LLMs have been used for various coding-related tasks including code generation and code recommendation [20], [21], [22], [23]. On one hand, in software testing, there are many tasks related to code generation, such as unit test generation [7], where the utilization of LLMs is expected to yield good performance. On the other hand, software testing possesses unique characteristics that differentiate it from code generation. For example, code generation primarily focuses on producing a single, correct code snippet, whereas software testing often requires generating diverse test inputs to ensure better coverage of the software under test [1]. The existence of these differences introduces new challenges and opportunities when employing LLMs for software testing. Moreover, people have benefited from the excellent performance of LLMs in generation and inference tasks, leading to the emergence of dozens of new practices that use LLMs for software testing.

This article presents a comprehensive review of the utilization of LLMs in software testing. We collect 102 relevant papers and conduct a thorough analysis from both software testing and LLMs perspectives, as roughly summarized in Fig. 1.

From the viewpoint of software testing, our analysis involves an examination of the specific software testing tasks for which LLMs are employed. Results show that 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, while we do not find the practices for applying LLMs in the tasks of early testing life-cycle (such as test requirement, test plan, etc). For each test task, we would provide detailed illustrations showcasing the utilization of LLMs in addressing the task, highlighting commonly-used practices, tracking technology evolution trends, and summarizing achieved performance, so as to facilitate readers in gaining a thorough overview of how LLMs are employed across various testing tasks.

From the viewpoint of LLMs, our analysis includes the commonly used LLMs in these studies, the types of prompt engineering, the input of the LLMs, as well as the accompanied techniques with these LLMs. Results show that about one-third of the studies utilize the LLMs through pre-training or

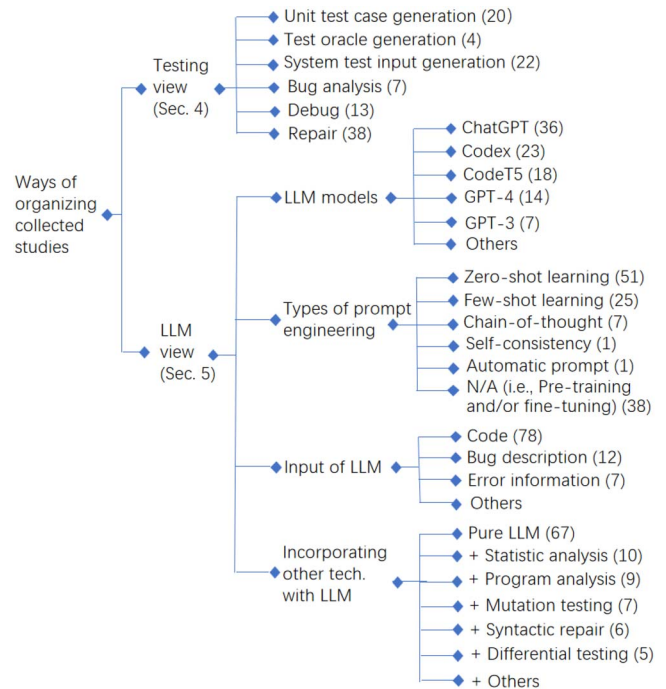


Fig. 1. Structure of the contents in this paper (the numbers in bracket indicates the number of involved papers, and a paper might involve zero or multiple items).

fine-tuning schema, while the others employ prompt engineering to communicate with LLMs to steer their behavior for desired outcomes. For prompt engineering, the zero-shot learning and few-shot learning strategies are most commonly used, while other advances like chain-of-thought promoting and self-consistency are rarely utilized. Results also show that traditional testing techniques like differential testing and mutation testing are usually accompanied by LLMs to help generate more diversified tests.

Furthermore, we summarize the key challenges and potential opportunities in this direction. Although software testing with LLMs has undergone significant growth in the past two years, there are still challenges in achieving high coverage of the testing, test oracle problem, rigorous evaluations, and real-world application of LLMs in software testing. Since it is a new emerging field, there are many research opportunities, including exploring LLMs in an early stage of testing, exploring LLMs for more types of software and non-functional testing, exploring advanced prompt engineering, as well as incorporating LLMs with traditional techniques.

This paper makes the following contributions:

- We thoroughly analyze 102 relevant studies that used LLMs for software testing, regarding publication trends, distribution of publication venues, etc.
- We conduct a comprehensive analysis from the perspective of software testing to understand the distribution of software testing tasks with LLM and present a thorough discussion about how these tasks are solved with LLM.
- We conduct a comprehensive analysis from the perspective of LLMs, and uncover the commonly-used LLMs, the

¹<https://openai.com/blog/chatgpt>

²<https://ai.meta.com/blog/large-language-model-llama-meta-ai/>

types of prompt engineering, input of the LLMs, as well as the accompanied techniques with these LLMs.

- We highlight the challenges in existing studies and present potential opportunities for further studies.
- We maintain a GitHub website <https://github.com/LLM-Testing/LLM4SoftwareTesting> that serves as a platform for sharing and hosting the latest publications about software testing with LLM.

We believe that this work will be valuable to both researchers and practitioners in the field of software engineering, as it provides a comprehensive overview of the current state and future vision of using LLMs for software testing. For researchers, this work can serve as a roadmap for future research in this area, highlighting potential avenues for exploration and identifying gaps in our current understanding of the use of LLMs in software testing. For practitioners, this work can provide insights into the potential benefits and limitations of using LLMs for software testing, as well as practical guidance on how to effectively integrate them into existing testing processes. By providing a detailed landscape of the current state and future vision of using LLMs for software testing, this work can help accelerate the adoption of this technology in the software engineering community and ultimately contribute to improving the quality and reliability of software systems.

II. BACKGROUND

A. Large Language Model (LLM)

Recently, pre-trained language models (PLMs) have been proposed by pretraining Transformer-based models over large-scale corpora, showing strong capabilities in solving various natural language processing (NLP) tasks [16], [17], [18], [19]. Studies have shown that model scaling can lead to improved model capacity, prompting researchers to investigate the scaling effect through further parameter size increases. Interestingly, when the parameter scale exceeds a certain threshold, these larger language models demonstrate not only significant performance improvements but also special abilities such as in-context learning, which are absent in smaller models such as BERT.

To discriminate the language models in different parameter scales, the research community has coined the term large language models (LLM) for the PLMs of significant size. LLMs typically refer to language models that have hundreds of billions (or more) of parameters and are trained on massive text data such as GPT-3, PaLM, Codex, and LLaMA. LLMs are built using the Transformer architecture, which stacks multi-head attention layers in a very deep neural network. Existing LLMs adopt similar model architectures (Transformer) and pre-training objectives (language modeling) as small language models, but largely scale up the model size, pre-training data, and total compute power. This enables LLMs to better understand natural language and generate high-quality text based on given context or prompts.

Note that, in existing literature, there is no formal consensus on the minimum parameter scale for LLMs, since the model capacity is also related to data size and total compute. In a

recent survey of LLMs [17], the authors focus on discussing the language models with a model size larger than 10B. Under their criteria, the first LLM is T5 released by Google in 2019, followed by GPT-3 released by OpenAI in 2020, and there are more than thirty LLMs released between 2021 and 2023 indicating its popularity. In another survey of unifying LLMs and knowledge graphs [24], the authors categorize the LLMs into three types: encoder-only (e.g., BERT), encoder-decoder (e.g., T5), and decoder-only network architecture (e.g., GPT-3). In our review, we take into account the categorization criteria of the two surveys and only consider the encoder-decoder and decoder-only network architecture of pre-training language models, since they can both support generative tasks. We do not consider the encoder-only network architecture because they cannot handle generative tasks, were proposed relatively early (e.g., BERT in 2018), and there are almost no models using this architecture after 2021. In other words, the LLMs discussed in this paper not only include models with parameters of over 10B (as mentioned in [17]) but also include other models that use the encoder-decoder and decoder-only network architecture (as mentioned in [24]), such as BART with 140M parameters and GPT-2 with parameter sizes ranging from 117M to 1.5B. This is also to potentially include more studies to demonstrate the landscape of this topic.

B. Software Testing

Software testing is a crucial process in software development that involves evaluating the quality of a software product. The primary goal of software testing is to identify defects or errors in the software system that could potentially lead to incorrect or unexpected behavior. The whole life cycle of software testing typically includes the following tasks (demonstrated in Fig. 4):

- Requirement Analysis: analyze the software requirements and identify the testing objectives, scope, and criteria.
- Test Plan: develop a test plan that outlines the testing strategy, test objectives, and schedule.
- Test Design and Review: develop and review the test cases and test suites that align with the test plan and the requirements of the software application.
- Test Case Preparation: the actual test cases are prepared based on the designs created in the previous stage.
- Test Execution: execute the tests that were designed in the previous stage. The software system is executed with the test cases and the results are recorded.
- Test Reporting: analyze the results of the tests and generate reports that summarize the testing process and identify any defects or issues that were discovered.
- Bug Fixing and Regression Testing: defects or issues identified during testing are reported to the development team for fixing. Once the defects are fixed, regression testing is performed to ensure that the changes have not introduced new defects or issues.
- Software Release: once the software system has passed all of the testing stages and the defects have been fixed, the software can be released to the customer or end user.



Fig. 2. Overview of the paper collection process.

The testing process is iterative and may involve multiple cycles of the above stages, depending on the complexity of the software system and the testing requirements.

During the testing phase, various types of tests may be performed, including unit tests, integration tests, system tests, and acceptance tests.

- Unit Testing involves testing individual units or components of the software application to ensure that they function correctly.
- Integration Testing involves testing different modules or components of the software application together to ensure that they work correctly as a system.
- System Testing involves testing the entire software system as a whole, including all the integrated components and external dependencies.
- Acceptance Testing involves testing the software application to ensure that it meets the business requirements and is ready for deployment.

In addition, there can be functional testing, performance testing, unit testing, security testing, accessibility testing, etc., which explores various aspects of the software under test [1].

III. PAPER SELECTION AND REVIEW SCHEMA

A. Paper Collection Methodology

Fig. 2 shows our paper search and selection process. To collect as much relevant literature as possible, we use both automatic search (from paper repository database) and manual search (from major software engineering and artificial intelligence venues). We searched papers from Jan. 2019 to Jun. 2023 and further conducted the second round of search to include the papers from Jul. 2023 to Oct. 2023.

1) *Automatic Search*: To ensure that we collect papers from diverse research areas, we conduct an extensive search using four popular scientific databases: ACM digital library, IEEE Xplore digital library, arXiv, and DBLP.

We search for papers whose title contains *keywords related to software testing tasks and testing techniques* (as shown below) in the first three databases. In the case of DBLP, we use additional *keywords related to LLMs* (as shown below) to filter out irrelevant studies, as relying solely on testing-related keywords would result in a large number of candidate studies. While using two sets of keywords for DBLP may result in overlooking certain related studies, we believe it is still a feasible strategy. This is due to the fact that a substantial number of studies present in this database can already be found in the

first three databases, and the fourth database only serves as a supplementary source for collecting additional papers.

- Keywords related with software testing tasks and techniques: test OR bug OR issue OR defect OR fault OR error OR failure OR crash OR debug OR debugger OR repair OR fix OR assert OR verification OR validation OR fuzz OR fuzzer OR mutation.
- Keywords related with LLMs: LLM OR language model OR generative model OR large model OR GPT-3 OR ChatGPT OR GPT-4 OR LLaMA OR PaLM2 OR CodeT5 OR CodeX OR CodeGen OR Bard OR InstructGPT. Note that, we only list the top ten most popular LLMs (based on Google search), since they are the search keywords for matching paper titles, rather than matching the paper content.

The above search strategy based on the paper title can recall a large number of papers, and we further conduct the automatic filtering based on the paper content. Specifically, we filter the paper whose content contains “LLM” or “language model” or “generative model” or “large model” or the name of the LLMs (using the LLMs in [17], [24] except those in our exclusion criteria). This can help eliminate the papers that do not involve the neural models.

2) *Manual Search*: To compensate for the potential omissions that may result from automated searches, we also conduct manual searches. In order to make sure we collect highly relevant papers, we conduct a manual search within the conference proceedings and journal articles from top-tier software engineering venues (listed in Table II).

In addition, given the interdisciplinary nature of this work, we also include the conference proceedings of the artificial intelligence field. We select the top ten venues based on the h5 index from Google Scholar, and exclude three computer vision venues, i.e., CVPR, ICCV, ECCV, as listed in Table II.

3) *Inclusion and Exclusion Criteria*: The search conducted on the databases and venue is, by design, very inclusive. This allows us to collect as many papers as possible in our pool. However, this generous inclusivity results in having papers that are not directly related to the scope of this survey. Accordingly, we define a set of specific inclusion and exclusion criteria and then we apply them to each paper in the pool and remove papers not meeting the criteria. This ensures that each collected paper aligns with our scope and research questions.

Inclusion Criteria. We define the following criteria for including papers:

- The paper proposes or improves an approach, study, or tool/framework that targets testing specific software or systems with LLMs.
- The paper applies LLMs to software testing practice, including all tasks within the software testing lifecycle as demonstrated in Section II-B.
- The paper presents an empirical or experimental study about utilizing LLMs in software testing practice.
- The paper involves specific testing techniques (e.g., fuzz testing) employing LLMs.

If a paper satisfies any of the following criteria, we will include it.

TABLE I
DETAILS OF THE COLLECTED PAPERS

ID	Topic	Paper title	Year	Reference
1	Unit test case generation	Unit Test Case Generation with Transformers and Focal Context	2020	[25]
2	Unit test case generation	Codet: Code Generation with Generated Tests	2022	[26]
3	Unit test case generation	Interactive Code Generation via Test-Driven User-Intent Formalization	2022	[27]
4	Unit test case generation	A3Test: Assertion-Augmented Automated Test Case Generation	2023	[28]
5	Unit test case generation	An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation	2023	[29]
6	Unit test case generation	An Initial Investigation of ChatGPT Unit Test Generation Capability	2023	[30]
7	Unit test case generation	Automated Test Case Generation Using Code Models and Domain Adaptation	2023	[31]
8	Unit test case generation	Automatic Generation of Test Cases based on Bug Reports: a Feasibility Study with Large Language Models	2023	[32]
9	Unit test case generation	Can Large Language Models Write Good Property-Based Tests?	2023	[33]
10	Unit test case generation	CAT-LM Training Language Models on Aligned Code And Tests	2023	[34]
11	Unit test case generation	ChatGPT vs SBST: A Comparative Assessment of Unit Test Suite Generation	2023	[8]
12	Unit test case generation	ChatUnitTest: a ChatGPT-based Automated Unit Test Generation Tool	2023	[35]
13	Unit test case generation	CODAMOSA: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models	2023	[36]
14	Unit test case generation	Effective Test Generation Using Pre-trained Large Language Models and Mutation Testing	2023	[37]
15	Unit test case generation	Exploring the Effectiveness of Large Language Models in Generating Unit Tests	2023	[38]
16	Unit test case generation	How Well does LLM Generate Security Tests?	2023	[39]
17	Unit test case generation	No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation	2023	[7]
18	Unit test case generation	Prompting Code Interpreter to Write Better Unit Tests on Quixbugs Functions	2023	[40]
19	Unit test case generation	Reinforcement Learning from Automatic Feedback for High-Quality Unit Test Generation	2023	[41]
20	Unit test case generation	Unit Test Generation using Generative AI: A Comparative Performance Analysis of Autogeneration Tools	2023	[42]
21	Test oracle generation	Generating Accurate Assert Statements for Unit Test Cases Using Pretrained Transformers	2022	[43]
22	Test oracle generation	Learning Deep Semantics for Test Completion	2023	[44]
23	Test oracle generation; Program repair	Using Transfer Learning for Code-Related Tasks	2023	[45]
24	Test oracle generation; Program repair	Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning	2023	[46]
25	System test input generation	Automated Conformance Testing for JavaScript Engines via Deep Compiler Fuzzing	2021	[47]
26	System test input generation	Fill in the Blank: Context-aware Automated Text Input Generation for Mobile GUI Testing	2022	[48]
27	System test input generation	Large Language Models are Pretty Good Zero-Shot Video Game Bug Detectors	2022	[49]
28	System test input generation	Slgpt: Using Transfer Learning to Directly Generate Simulink Model Files and Find Bugs in the Simulink Toolchain	2021	[50]
29	System test input generation	Augmenting Greybox Fuzzing with Generative AI	2023	[51]
30	System test input generation	Automated Test Case Generation Using T5 and GPT-3	2023	[52]
31	System test input generation	Automating GUI-based Software Testing with GPT-3	2023	[53]
32	System test input generation	AXNav: Replaying Accessibility Tests from Natural Language	2023	[54]
33	System test input generation	Can ChatGPT Advance Software Testing Intelligence? An Experience Report on Metamorphic Testing	2023	[55]
34	System test input generation	Efficient Mutation Testing via Pre-Trained Language Models	2023	[56]
35	System test input generation	Large Language Models are Edge-Case Generators: Crafting Unusual Programs for Fuzzing Deep Learning Libraries	2023	[57]
36	System test input generation	Large Language Models are Zero Shot Fuzzers: Fuzzing Deep Learning Libraries via Large Language Models	2023	[58]
37	System test input generation	Large Language Models for Fuzzing Parsers (Registered Report)	2023	[59]
38	System test input generation	LLM for Test Script Generation and Migration: Challenges, Capabilities, and Opportunities	2023	[60]
39	System test input generation	Make LLM a Testing Expert: Bringing Human-like Interaction to Mobile GUI Testing via Functionality-aware Decisions	2023	[14]
40	System test input generation	PentestGPT: An LLM-empowered Automatic Penetration Testing Tool	2023	[61]
41	System test input generation	SMT Solver Validation Empowered by Large Pre-Trained Language Models	2023	[62]
42	System test input generation	TARGET: Automated Scenario Generation from Traffic Rules for Testing Autonomous Vehicles	2023	[63]
43	System test input generation	Testing the Limits: Unusual Text Inputs Generation for Mobile App Crash Detection with Large Language Model	2023	[64]
44	System test input generation	Understanding Large Language Model Based Fuzz Driver Generation	2023	[65]
45	System test input generation	Universal Fuzzing via Large Language Models	2023	[66]
46	System test input generation	Variable Discovery with Large Language Models for Metamorphic Testing of Scientific Software	2023	[67]
47	System test input generation	White-box Compiler Fuzzing Empowered by Large Language Models	2023	[68]
48	Bug analysis	Itger: an Automatic Issue Title Generation Tool	2022	[69]
49	Bug analysis	CrashTranslator: Automatically Reproducing Mobile Application Crashes Directly from Stack Trace	2023	[70]
50	Bug analysis	Cupid: Leveraging ChatGPT for More Accurate Duplicate Bug Report Detection	2023	[71]
51	Bug analysis	Employing Deep Learning and Structured Information Retrieval to Answer Clarification Questions on Bug Reports	2023	[72]
52	Bug analysis	Explaining Software Bugs Leveraging Code Structures in Neural Machine Translation	2022	[73]
53	Bug analysis	Prompting Is All Your Need: Automated Android Bug Replay with Large Language Models	2023	[74]
54	Bug analysis	Still Confusing for Bug-Component Triaging? Deep Feature Learning and Ensemble Setting to Rescue	2023	[75]
55	Debug	Detect-Localize-Repair: A Unified Framework for Learning to Debug with CodeT5	2022	[76]
56	Debug	Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction	2022	[77]
57	Debug	A Preliminary Evaluation of LLM-Based Fault Localization	2023	[78]
58	Debug	Addressing Compiler Errors: Stack Overflow or Large Language Models?	2023	[79]
59	Debug	Can LLMs Demystify Bug Reports?	2023	[80]
60	Debug	Dcc -help: Generating Context-Aware Compiler Error Explanations with Large Language Models	2023	[81]
61	Debug	Explainable Automated Debugging via Large Language Model-driven Scientific Debugging	2023	[82]
62	Debug	Large Language Models for Test-Free Fault Localization	2023	[83]
63	Debug	Large Language Models in Fault Localisation	2023	[84]
64	Debug	LLM4CBI: Taming LLMs to Generate Effective Test Programs for Compiler Bug Isolation	2023	[85]
65	Debug	Nuances are the Key: Unlocking ChatGPT to Find Failure-Inducing Tests with Differential Prompting	2023	[86]
66	Debug	Teaching Large Language Models to Self-Debug	2023	[87]
67	Debug; Program repair	A study on Prompt Design, Advantages and Limitations of ChatGPT for Deep Learning Program Repair	2023	[88]
68	Program repair	Examining Zero-Shot Vulnerability Repair with Large Language Models	2022	[89]
69	Program repair	Automated Repair of Programs from Large Language Models	2022	[90]
70	Program repair	Fix Bugs with Transformer through a Neural-Symbolic Edit Grammar	2022	[91]
71	Program repair	Practical Program Repair in the Era of Large Pre-trained Language Models	2022	[92]
72	Program repair	Repairing Bugs in Python Assignments Using Large Language Models	2022	[93]
73	Program repair	Towards JavaScript Program Repair with Generative Pre-trained Transformer (GPT-2)	2022	[94]
74	Program repair	An Analysis of the Automatic Bug Fixing Performance of ChatGPT	2023	[95]
75	Program repair	An Empirical Study on Fine-Tuning Large Language Models of Code for Automated Program Repair	2023	[96]
76	Program repair	An Evaluation of the Effectiveness of OpenAI's ChatGPT for Automated Python Program Bug Fixing using QuixBugs	2023	[97]
77	Program repair	An Extensive Study on Model Architecture and Program Representation in the Domain of Learning-based Automated Program Repair	2023	[98]
78	Program repair	Can OpenAI's Codex Fix Bugs? An Evaluation on QuixBugs	2022	[99]
79	Program repair	CIRCLE: Continual Repair Across Programming Languages	2022	[100]
80	Program repair	Coffee: Boost Your Code LLMs by Fixing Bugs with Feedback	2023	[101]
81	Program repair	Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair	2023	[102]
82	Program repair	Domain Knowledge Matters: Improving Prompts with Fix Templates for Repairing Python Type Errors	2023	[103]
83	Program repair	Enhancing Genetic Improvement Mutations Using Large Language Models	2023	[104]
84	Program repair	FixEval: Execution-based Evaluation of Program Fixes for Programming Problems	2023	[105]
85	Program repair	Fixing Hardware Security Bugs with Large Language Models	2023	[106]
86	Program repair	Fixing Rust Compilation Errors using LLMs	2023	[107]
87	Program repair	Framing Program Repair as Code Completion	2022	[108]
88	Program repair	Frustrated with Code Quality Issues? LLMs can Help!	2023	[109]
89	Program repair	GPT-3-Powered Type Error Debugging: Investigating the Use of Large Language Models for Code Repair	2023	[110]
90	Program repair	How Effective Are Neural Networks for Fixing Security Vulnerabilities	2023	[111]
91	Program repair	Impact of Code Language Models on Automated Program Repair	2023	[112]
92	Program repair	Interfix: End-to-end Program Repair with LLMs	2023	[113]
93	Program repair	Keep the Conversation Going: Fixing 162 out of 337 bugs for \$0.42 each using ChatGPT	2023	[114]
94	Program repair	Neural Program Repair with Program Dependence Analysis and Effective Filter Mechanism	2023	[115]
95	Program repair	Out of Context: How important is Local Context in Neural Program Repair?	2023	[116]
96	Program repair	Pre-trained Model-based Automated Software Vulnerability Repair: How Far are We?	2023	[117]
97	Program repair	RAPGen: An Approach for Fixing Code Inefficiencies in Zero-Shot	2023	[118]
98	Program repair	RAP-Gen: Retrieval-Augmented Patch Generation with CodeT5 for Automatic Program Repair	2023	[119]
99	Program repair	STEAM: Simulating the InTeractive BEhavior of ProgrAMmers for Automatic Bug Fixing	2023	[120]
100	Program repair	Towards Generating Functionally Correct Code Edits from Natural Language Issue Descriptions	2023	[121]
101	Program repair	VulRepair: a T5-based Automated Software Vulnerability Repair	2022	[122]
102	Program repair	What Makes Good In-Context Demonstrations for Code Intelligence Tasks with LLMs?	2023	[123]

TABLE II
CONFERENCE PROCEEDINGS AND JOURNALS CONSIDERED FOR
MANUAL SEARCH

Acronym	Venue
SE Conference	ICSE ESEC/FSE
	International Conference on Software Engineering Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering
	ASE
	International Conference on Automated Software Engineering
	ISSTA
	International Symposium on Software Testing and Analysis
	ICST
	International Conference on Software Testing, Verification and Validation
	ESEM
SE Journal	International Symposium on Empirical Software Engineering and Measurement
	MSR
	International Conference on Mining Software Repositories
	QRS
	International Conference on Software Quality, Reliability and Security
	ICSME
	International Conference on Software Maintenance and Evolution
	ISSRE
	International Symposium on Software Reliability Engineering
AI Venues	TSE
	Transactions on Software Engineering
	TOSEM
	Transactions on Software Engineering and Methodology
	EMSE
	Empirical Software Engineering
	ASE
	Automated Software Engineering
	JSS
AI Venues	Journal of Systems and Software
	JSEP
	Journal of Software: Evolution and Process
	STVR
	Software Testing, Verification and Reliability
	IEEE SOFTW.
	IEEE Software
	IET SOFTW.
	IET Software
AI Venues	IST
	Information and Software Technology
	SQJ
	Software Quality Journal
	ICLR
	International Conference on Learning Representations
	NeurIPS
	Conference on Neural Information Processing Systems
	ICML
	International Conference on Machine Learning
AI Venues	AAAI
	AAAI Conference on Artificial Intelligence
	EMNLP
	Conference on Empirical Methods in Natural Language Processing
	ACL
	Annual Meeting of the Association for Computational Linguistics
	IJCAI
	International Joint Conference on Artificial Intelligence

Exclusion Criteria. The following studies would be excluded during study selection:

- The paper does not involve software testing tasks, e.g., code comment generation.
- The paper does not utilize LLMs, e.g., using recurrent neural networks.
- The paper mentions LLMs only in future work or discussions rather than using LLMs in the approach.
- The paper utilizes language models with encoder-only architecture, e.g., BERT, which can not directly be utilized for generation tasks (as demonstrated in Section II-A).
- The paper focuses on testing the performance of LLMs, such as fairness, stability, security, etc. [124], [125], [126].
- The paper focuses on evaluating the performance of LLM-enabled tools, e.g., evaluating the code quality of the code generation tool Copilot [127], [128], [129].

For the papers collected through automatic search and manual search, we conduct a manual inspection to check whether they satisfy our inclusion criteria and filter those following our exclusion criteria. Specifically, the first two authors read each paper to carefully determine whether it should be included based on the inclusion criteria and exclusion criteria, and any paper with different decisions will be handed over to the third author to make the final decision.

4) *Quality Assessment:* In addition, we establish quality assessment criteria to exclude low-quality studies as shown below. For each question, the study's quality is rated as "yes", "partial" or "no" which are assigned values of 1, 0.5, and 0, respectively. Papers with a score of less than eight will be excluded from our study.

- Is there a clearly stated research goal related to software testing?
- Is there a defined and repeatable technique?
- Is there any explicit contribution to software testing?

- Is there an explicit description of which LLMs are utilized?
- Is there an explicit explanation about how the LLMs are utilized?
- Is there a clear methodology for validating the technique?
- Are the subject projects selected for validation suitable for the research goals?
- Are there control techniques or baselines to demonstrate the effectiveness of the proposed technique?
- Are the evaluation metrics relevant (e.g., evaluate the effectiveness of the proposed technique) to the research objectives?
- Do the results presented in the study align with the research objectives and are they presented in a clear and relevant manner?

5) *Snowballing:* At the end of searching database repositories and conference proceedings and journals, and applying inclusion/exclusion criteria and quality assessment, we obtain the initial set of papers. Next, to mitigate the risk of omitting relevant literature from this survey, we also perform backward snowballing [130] by inspecting the references cited by the collected papers so far. Note that, this procedure did not include new studies, which might because the surveyed topic is quite new and the reference studies tend to published previously, and we already include a relatively comprehensive automatic and manual search.

B. Collection Results

As shown in Fig. 2, the collection process started with a total of 14,623 papers retrieved from four academic databases employing keyword searching. Then after automated filtering, manual search, applying inclusion/exclusion criteria, and quality assessment, we finally collected a total of 102 papers involving software testing with LLMs. Table I shows the details of the collected papers. Besides, we provide a more comprehensive overview of these papers regarding the specific characteristics (will be illustrated in Section IV and Section V) in the online appendix of the paper.

Note that, there are two studies which are respectively the extension of a previously published paper by the same authors ([46] and [131], [68] and [132]), and we only keep the extended version to avoid duplicate.

C. General Overview of Collected Paper

Among the papers, 47% papers are published in software engineering venues, among which 19 papers are from ICSE, 5 papers are from FSE, 5 papers are from ASE, and 3 papers are from ISSTA. 2% papers are published in artificial intelligence venues such as EMNLP and ICLR, and 5% papers are published in program analysis or security venues like PLDI and S & P. Besides, 46% of the papers have not yet been published via peer-reviewed venues, i.e., they are disclosed on arXiv. This is understandable because this field is emerging and many works are just completed and in the process of submission. Although these papers did not undergo peer review, we have a quality

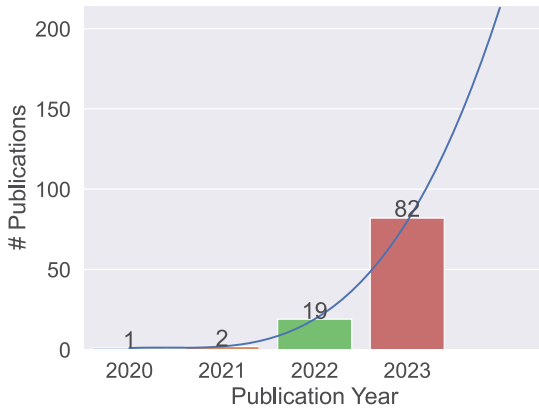


Fig. 3. Trend in the number of papers with year.

assessment process that eliminates papers with low quality, which potentially ensures the quality of this survey.

Fig. 3 demonstrates the trend of our collected papers per year. We can see that as the years go by, the number of papers in this field is growing almost exponentially. In 2020 and 2021, there were only 1 and 2 papers, respectively. In 2022, there were 19 papers, and in 2023, there have been 82 papers. It is conceivable that there will be even more papers in the future, which indicates the popularity and attention that this field is receiving.

IV. ANALYSIS FROM SOFTWARE TESTING PERSPECTIVE

This section presents our analysis from the viewpoint of software testing and organizes the collected studies in terms of testing tasks. Fig. 4 lists the distribution of each involved testing task, aligned with the software testing life cycle. We first provide a general overview of the distribution, followed by further analysis for each task. Note that, for each following subsection, the cumulative total of subcategories may not always match the total number of papers since a paper might belong to more than one subcategory.

We can see that LLMs have been effectively used in both the mid to late stages of the software testing life cycle. In the test case preparation phase, LLMs have been utilized for tasks such as generating unit test cases, test oracle generation, and system test input generation. These tasks are crucial in the mid-phase of software testing to help catch issues and prevent further development until issues are resolved. Furthermore, in later phases such as the test report/bug reports and bug fix phase, LLMs have been employed for tasks such as bug analysis, debugging, and repair. These tasks are critical towards the end of the testing phase when software bugs need to be resolved to prepare for the product's release.

A. Unit Test Case Generation

Unit test case generation involves writing unit test cases to check individual units/components of the software independently and ensure that they work correctly. For a method under test (i.e., often called the focal method), its corresponding unit test consists of a test prefix and a test oracle. In particular,

the test prefix is typically a series of method invocation statements or assignment statements, which aims at driving the focal method to a testable state; and then the test oracle serves as the specification to check whether the current behavior of the focal method satisfies the expected one, e.g., the test assertion.

To alleviate manual efforts in writing unit tests, researchers have proposed various techniques to facilitate automated unit test generation. Traditional unit test generation techniques leverage search-based [3], [4], constraint-based [5] or random-based strategies [6] to generate a suite of unit tests with the main goal of maximizing the coverage in the software under test. Nevertheless, the coverage and the meaningfulness of the generated tests are still far from satisfactory.

Since LLMs have demonstrated promising results in tasks such as code generation, and given that both code generation and unit test case generation involve generating source code, recent research has extended the domain of code generation to encompass unit test case generation. Despite initial success, there are nuances that set unit test case generation apart from general code generation, signaling the need for more tailored approaches.

1) *Pre-Training or Fine-Tuning LLMs for Unit Test Case Generation:* Due to the limitations of LLMs in their earlier stages, a majority of the earlier published studies adopt this pre-training or fine-tuning schema. Moreover, in some recent studies, this schema continues to be employed to increase the LLMs' familiarity with domain knowledge. Alagarsamy et al. [28] first pre-trained the LLM with the focal method and asserted statements to enable the LLM to have a stronger foundation knowledge of assertions, 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. Tufano et al. [25] utilized a similar schema by pre-training the LLM on a large unsupervised Java corpus, and supervised fine-tuning a downstream translation task for generating unit tests. Hashtroudi et al. [31] leveraged the existing developer-written tests for each project to generate a project-specific dataset for domain adaptation when fine-tuning the LLM, which can facilitate generating human-readable unit tests. Rao et al. [34] trained a GPT-style language model by utilizing a pre-training signal that explicitly considers the mapping between code and test files. Steenhoek et al. [41] utilizes reinforcement learning to optimize models by providing rewards based on static quality metrics that can be automatically computed for the generated unit test cases.

2) *Designing Effective Prompts for Unit Test Case Generation:* The advancement of LLMs has allowed them to excel at targeted tasks without pre-training or fine-tuning. Therefore most later studies typically focus on how to design the prompt, to make the LLM better at understanding the context and nuances of this task. Xie et al. [35] generated unit test cases by parsing the project, extracting essential information, and creating an adaptive focal context that includes a focal method and its dependencies within the pre-defined maximum prompt token limit of the LLM, and incorporating these context into a prompt to query the LLM. Dakhel et al. [37] introduced MuTAP for improving the effectiveness of test cases generated by LLMs

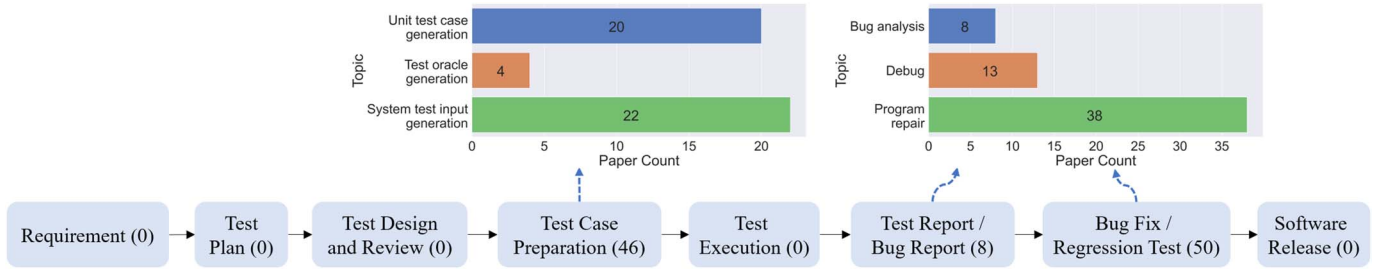


Fig. 4. Distribution of testing tasks with LLMs (aligned with software testing life cycle [1], [133], [134], the number in bracket indicates the number of collected studies per task, and one paper might involve multiple tasks).

TABLE III
PERFORMANCE OF UNIT TEST CASE GENERATION

Dataset	Correctness	Coverage	LLM	Paper
5 Java projects from Defects4J	16.21%	5%-13% (line coverage)	BART	[25]
10 Java projects	40%	89% (line coverage), 90% (branch coverage)	ChatGPT	[35]
CodeSearchNet	41%	N/A	ChatGPT	[7]
HumanEval	78%	87% (line coverage), 92% (branch coverage)	Codex	[38]
SF110	2%	2% (line coverage), 1% (branch coverage)	Codex	[38]

Note that, [39] experiments with Codex, CodeGen, and ChatGPT, and the best performance was achieved by Codex.

in terms of revealing bugs by leveraging mutation testing. They augment prompts with surviving mutants, as those mutants highlight the limitations of test cases in detecting bugs. Zhang et al. [39] generated security tests with vulnerable dependencies with LLMs.

Yuan et al. [7] first 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. They further propose an approach that leveraged the ChatGPT itself to improve the quality of its generated tests with an initial test generator and an iterative test refiner. Specifically, the iterative test refiner iteratively fixed the compilation errors in the tests generated by the initial test generator, which follows a validate-and-fix paradigm to prompt the LLM based on the compilation error messages and additional code context. Guilherme et al. [30] and Li et al. [40] respectively evaluated the quality of the generated unit tests by LLM using different metrics and different prompts.

3) *Test Generation With Additional Documentation:* Vikram et al. [33] went a step further by investigating the potential of using LLMs to generate property-based tests when provided API documentation. They believe that the documentation of an API method can assist the LLM in producing logic to generate random inputs for that method and deriving meaningful properties of the result to check. Instead of generating unit tests from the source code, Plein et al. [32] generated the tests based on user-written bug reports.

4) *LLM and Search-Based Method for Unit Test Generation:* The aforementioned studies utilize LLMs for the whole unit test case generation task, while Lemieux et al. [36] focus on a different direction, i.e., first letting the traditional search-based

software testing techniques (e.g., Pynguin [135]) in generating unit test case until its coverage improvements stall, then asking the LLM to provide the example test cases for under-covered functions. These examples can help the original test generation redirect its search to more useful areas of the search space.

Tang et al. [8] conducts a systematic comparison of test suites generated by the LLM and the state-of-the-art search-based software testing tool EvoSuite, by considering the correctness, readability, code coverage, and bug detection capability. Similarly, Bhatia [42] experimentally investigates the quality of unit tests generated by LLM compared to a commonly-used test generator Pynguin.

5) *Performance of Unit Test Case Generation:* Since the aforementioned studies of unit test case generation are based on different datasets, one can hardly derive a fair comparison and we present the details in Table III to let the readers obtain a general view. We can see that in the SF110 benchmark, all three evaluated LLMs have quite low performance, i.e., 2% coverage [38]. SF110 is an EvoSuite (a search-based unit test case generation technique) benchmark consisting of 111 open-source Java projects retrieved from SourceForge, containing 23,886 classes, over 800,000 bytecode-level branches, and 6.6 million lines of code. The authors did not present detailed reasons for the low performance which can be further explored in the future.

B. Test Oracle Generation

A test oracle is a source of information about whether the output of a software system (or program or function or method) is correct or not [136]. Most of the collected studies in this category target the test assertion generation, which is inside a unit test case. Nevertheless, we opted to treat these studies as separate sections to facilitate a more thorough analysis.

Test assertion, which is to indicate the potential issues in the tested code, is an important aspect that can distinguish the unit test cases from the regular code. This is why some studies specifically focus on the generation of effective test assertions. Actually, before using LLMs, researchers have proposed RNN-based approaches that aim at learning from thousands of unit test methods to generate meaningful assert statements [137], yet only 17% of the generated asserts can exactly match with the ground truth asserts. Subsequently, to improve the performance, several researchers utilized the LLMs for this task.

Mastropaolo et al. [45], [131] pre-trained a T5 model on a dataset composed of natural language English text and source code. Then, it fine-tuned such a model by reusing datasets used in four previous works that used deep learning techniques (such as RNN as mentioned before) including test assertion generation and program repair, etc. Results showed that the extract match rate of the generated test assertion is 57%. Tufano et al. [43] proposed a similar approach which separately pre-trained the LLM with English corpus and code corpus, and then fine-tuned it on the asserts dataset (with test methods, focal methods, and asserts). This further improved the performance to 62% of the exact match rate. Besides the syntax-level data as previous studies, Nie et al. [45] fine-tuned the LLMs with six kinds of code semantics data, including the execution result (e.g., types of the local variables) and execution context (e.g., the last called method in the test method), which enabled LLMs to learn to understand the code execution information. The exact match rate is 17% (note that this paper is based on a different dataset from all other studies mentioned under this topic).

The aforementioned studies utilized the pre-training and fine-tuning schema when using LLMs, and with the increasingly powerful capabilities of LLMs, they can perform well on specific tasks without these specialized pre-training or fine-tuning datasets. Subsequently, Nashid et al. [47] utilized prompt engineering for this task, and proposed a technique for prompt creation that automatically retrieves code demonstrations similar to the task, based on embedding or frequency analysis. They also present evaluations about the few-shot learning with various numbers (e.g., zero-shot, one-shot, or n-shot) and forms (e.g., random vs. systematic, or with vs. without natural language descriptions) of the prompts, to investigate its feasibility on test assertion generation. With only a few relevant code demonstrations, this approach can achieve an accuracy of 76% for exact matches in test assertion generation, which is the state-of-the-art performance for this task.

C. System Test Input Generation

This category encompasses the studies related to creating test input of system testing for enabling the automation of test execution. We employ three subsections to present the analysis from three different orthogonal viewpoints, and each of the collected studies may be analyzed in one or more of these subsections.

The first subsection is *input generation in terms of software types*. The generation of system-level test inputs for software testing varies for specific types of software being tested. For

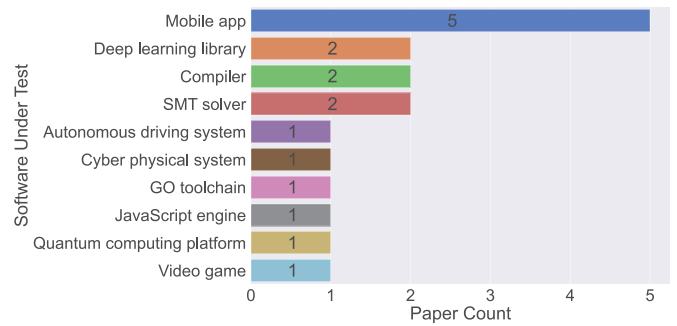


Fig. 5. Distribution of software under test.

example, for mobile applications, the test input generation requires providing a diverse range of text inputs or operation combinations (e.g., click a button, long press a list) [14], [49], which is the key to testing the application's functionality and user interface; while for Deep Learning (DL) libraries, the test input is a program which covers diversified DL APIs [58], [59]. This subsection will demonstrate how the LLMs are utilized to generate inputs for different types of software.

The second subsection *input generation in terms of testing techniques*. We have observed that certain approaches serve as specific types of testing techniques. For example, dozens of our collected studies specifically focus on using LLMs for fuzz testing. Therefore, this subsection would provide an analysis of the collected studies in terms of testing techniques, showcasing how the LLMs are employed to enhance traditional testing techniques.

The third subsection *input generation in terms of input and output*. While most of the collected studies take the source code or the software itself as the input and directly output the software's test input, there are studies that utilize alternative forms of input and output. This subsection would provide an analysis of such studies, highlighting different approaches and their input-output characteristics.

1) *Input Generation in Terms of Software Types*: Fig. 5 demonstrates the types of software under test in our collected studies. It is evident that the most prominent category is mobile apps, with five studies utilizing LLMs for testing, possibly due to their prevalence and importance in today's business and daily life. Additionally, there are respectively two studies focusing on testing deep learning libraries, compilers, and SMT solvers. Moreover, LLM-based testing techniques have also been applied to domains such as cyber-physical systems, quantum computing platforms, and more. This widespread adoption of LLMs demonstrates their effectiveness in handling diverse test inputs and enhancing testing activities across various software domains. A detailed analysis is provided below.

a) *Test input generation for mobile apps*: For mobile app testing, one difficulty is to generate the appropriate text inputs to proceed to the next page, which remains a prominent obstacle for testing coverage. Considering the diversity and semantic requirement of valid inputs (e.g., flight departure, movie name), traditional techniques with heuristic-based or constraint-based techniques [10], [138] are far from generating meaningful text

input. Liu et al. [49] employ the LLM to intelligently generate the semantic input text according to the GUI context. In detail, their proposed QTypist automatically extracts the component information related to the EditText for generating the prompts, and then inputs the prompts into the LLM to generate the input text.

Besides the text input, there are other forms of input for mobile apps, i.e., operations like ‘click a button’ and ‘select a list’. To fully test an app, it is required to cover more GUI pages and conduct more meaningful exploration traces through the GUI operations, yet existing studies with random-/rule-based methods [9], [10], model-based methods [11], [12], and learning-based methods [13] are unable to understand the semantic information of the GUI page thus could not conduct the trace planning effectively. Liu et al. [14] formulates 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 LLM to elicit testing scripts (i.e., GUI operation), and executing them to keep passing the app feedback to LLM, iterating the whole process. The proposed GPTDroid extracts the static context of the GUI page and the dynamic context of the iterative testing process, and designs prompts for inputting this information to LLM which enables the LLM to better understand the GUI page as well as the whole testing process. It 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. Similarly, Zimmermann et al. utilize the LLM to interpret natural language test cases and programmatically navigate through the application under test [54].

Yu et al. [61] investigate the LLM’s capabilities in the mobile app test script generation and migration task, including the scenario-based test generation, and the cross-platform/app test migration.

b) Test input generation for DL libraries: The input for testing DL libraries is DL programs, and the difficulty in generating the diversified input DL programs is that they need to satisfy both the input language (e.g., Python) syntax/semantics and the API input/shape constraints for tensor computations. Traditional techniques with API-level fuzzing [139], [140] or model-level fuzzing [141], [142] suffer from the following limitations: 1) lack of diverse API sequence thus cannot reveal bugs caused by chained API sequences; 2) cannot generate arbitrary code thus cannot explore the huge search space that exists when using the DL libraries. Since LLMs can include numerous code snippets invoking DL library APIs in their training corpora, they can implicitly learn both language syntax/semantics and intricate API constraints for valid DL program generation. Taken in this sense, Deng et al. [59] used both generative and infilling LLMs to generate and mutate valid/diverse input DL programs for fuzzing DL libraries. In detail, it first uses a generative LLM (CodeX) to generate a set of seed programs (i.e., code snippets that use the target DL APIs). Then it replaces part of the seed program with masked tokens using different mutation operators and leverages the ability of infilling LLM (InCoder) to perform code infilling to generate new code that replaces the

masked tokens. Their follow-up study [58] goes a step further to prime LLMs to synthesize unusual programs for the fuzzing DL libraries. It is built on the well-known hypothesis that historical bug-triggering programs may include rare/valuable code ingredients important for bug finding and show improved bug detection performance.

c) Test input generation for other types of software: There are also dozens of studies that address testing tasks in various other domains, due to space limitations, we will present a selection of representative studies in these domains.

Finding bugs in a commercial cyber-physical system (CPS) development tool such as Simulink is even more challenging. Given the complexity of the Simulink language, generating valid Simulink model files for testing is an ambitious task for traditional machine learning or deep learning techniques. Shrestha et al. [51] employs a small set of Simulink-specific training data to fine-tune the LLM for generating Simulink models. Results show that it can create Simulink models quite similar to the open-source models, and can find a super-set of the bugs traditional fuzzing approaches found.

Sun et al. [63] utilize LLM to generate test formulas for fuzzing SMT solvers. It retrains the LLMs on a large corpus of SMT formulas to enable them to acquire SMT-specific domain knowledge. Then it further fine-tunes the LLMs on historical bug-triggering formulas, which are known to involve structures that are more likely to trigger bugs and solver-specific behaviors. The LLM-based compiler fuzzer proposed by Yang et al. [69] adopts a dual-model framework: (1) an analysis LLM examines the low-level optimization source code and produces requirements on the high-level test programs that can trigger the optimization; (2) a generation LLM produces test programs based on the summarized requirements. Ye et al. [48] utilize the LLM for generating the JavaScript programs and then use the well-structured ECMAScript specifications to automatically generate test data along with the test programs, after that they apply differential testing to expose bugs.

2) Input Generation in Terms of Testing Techniques: By utilizing system test inputs generated by LLMs, the collected studies aim to enhance traditional testing techniques and make them more effective. Among these techniques, fuzz testing is the most commonly involved one. Fuzz testing, as a general concept, revolves around generating invalid, unexpected, or random data as inputs to evaluate the behavior of software. LLMs play a crucial role in improving traditional fuzz testing by facilitating the generation of diverse and realistic input data. This enables fuzz testing to uncover potential bugs in the software by subjecting it to a wide range of input scenarios. In addition to fuzz testing, LLMs also contribute to enhancing other testing techniques, which will be discussed in detail later.

a) Universal fuzzing framework: Xia et al. [67] present Fuzz4All that can target many different input languages and many different features of these languages. The key idea behind it is to leverage LLMs as an input generation and mutation engine, which enables the approach to produce diverse and realistic inputs for any practically relevant language. To realize this potential, they present a novel auto-prompting technique, which creates LLM prompts that are well-suited for fuzzing, and a

novel LLM-powered fuzzing loop, which iteratively updates the prompt to create new fuzzing inputs. They experiment with six different languages (C, C++, Go, SMT2, Java and Python) as inputs and demonstrate higher coverage than existing language-specific fuzzers. Hu et al. [52] propose a greybox fuzzer augmented by the LLM, which picks a seed in the fuzzer's seed pool and prompts the LLM to produce the mutated seeds that might trigger a new code region of the software. They experiment with three categories of input formats, i.e., formatted data files (e.g., json, xml), source code in different programming languages (e.g., JS, SQL, C), text with no explicit syntax rules (e.g., HTTP response, md5 checksum). In addition, effective fuzzing relies on the effective fuzz driver, and Zhang et al. [66] utilize LLMs on the fuzz driver generation, in which five query strategies are designed and analyzed from basic to enhanced.

b) Fuzzing techniques for specific software: There are studies that focus on the fuzzing techniques tailored to specific software, e.g., the deep learning library [58], [59], compiler [69], SMT solvers [63], input widget of mobile app [65], cyber-physical system [51], etc. One key focus of these fuzzing techniques is to generate diverse test inputs so as to achieve higher coverage. This is commonly achieved by combining the mutation technique with LLM-based generation, where the former produces various candidates while the latter is responsible for generating the executable test inputs [59], [63]. Another focus of these fuzzing techniques is to generate the risky test inputs that can trigger bugs earlier. To achieve this, a common practice is to collect the historical bug-triggering programs to fine-tune the LLM [63] or treat them as the demonstrations when querying the LLM [58], [65].

c) Other testing techniques: There are studies that utilize LLMs for enhancing GUI testing for generating meaningful text input [49] and functionality-oriented exploration traces [14], which has been introduced in *Test input generation for mobile apps* part of Section IV-C1.

Besides, Deng et al. [62] leverage the LLMs to carry out penetration testing tasks automatically. It involves setting a penetration testing goal for the LLM, soliciting it for the appropriate operation to execute, implementing it in the testing environment, and feeding the test outputs back to the LLM for next-step reasoning.

3) Input Generation in Terms of Input and Output:

a) Output format of test generation: Although most works use LLM to generate test cases directly, there are also some works generating indirect inputs like testing code, test scenarios, metamorphic relations, etc. Liu et al. [65] propose InputBlaster which leverages the LLM to automatically generate unusual text inputs for fuzzing the text input widgets in mobile apps. It formulates the unusual inputs generation problem as a task of producing a set of test generators, each of which can yield a batch of unusual text inputs under the same mutation rule. In detail, InputBlaster leverages LLM to produce the test generators together with the mutation rules serving as the reasoning chain and utilizes the in-context learning schema to demonstrate the LLM with examples for boosting the performance. Deng et al. [64] use LLM to extract key information related to the test scenario from a traffic rule, and represent the extracted information in a test scenario schema, then

synthesize the corresponding scenario scripts to construct the test scenario. Luu et al. [56] examine the effectiveness of LLM in generating metamorphic relations (MRs) for metamorphic testing. Their results show that ChatGPT can be used to advance software testing intelligence by proposing MRs candidates that can be later adapted for implementing tests, but human intelligence should still inevitably be involved to justify and rectify their correctness.

b) Input format of test generation: The aforementioned studies primarily take the source code or the software as the input of LLM, yet there are also studies that take natural language description as the input for test generation. Mathur et al. [53] propose to generate test cases from the natural language described requirements. Ackerman et al. [60] generate the instances from natural language described requirements recursively to serve as the seed examples for a mutation fuzzer.

D. Bug Analysis

This category involves analyzing and categorizing the identified software bugs to enhance understanding of the bug, and facilitate subsequent debug and bug repair. Mukherjee et al. [73] generate relevant answers to follow-up questions for deficient bug reports to facilitate bug triage. Su et al. [76] transform the bug-component triaging into a multi-classification task and a generation task with LLM, then ensemble the prediction results from them to improve the performance of bug-component triaging further. Zhang et al. [72] first leverage the LLM under the zero-shot setting to get essential information on bug reports, then use the essential information as the input to detect duplicate bug reports. Mahbub et al. [74] proposes to explain software bugs with LLM, which generates natural language explanations for software bugs by learning from a large corpus of bug-fix commits. Zhang et al. [70] target to automatically generate the bug title from the descriptions of the bug, which aims to help developers write issue titles and facilitate the bug triaging and follow-up fixing process.

E. Debug

This category refers to the process of identifying and locating the cause of a software problem (i.e., bug). It involves analyzing the code, tracing the execution flow, collecting error information to understand the root cause of the issue, and fixing the issue. Some studies concentrate on the comprehensive debug process, while others delve into specific sub-activities within the process.

1) Overall Debug Framework: Bui et al. [77] proposes a unified Detect-Localize-Repair framework based on the LLM for debugging, which first determines whether a given code snippet is buggy or not, then identifies the buggy lines, and translates the buggy code to its fixed version. Kang et al. [83] proposes automated scientific debugging, a technique that given buggy code and a bug-revealing test, prompts LLMs to automatically generate hypotheses, uses debuggers to actively interact with buggy code, and thus automatically reaches conclusions prior to patch generation. Chen et al. [88] demonstrate that self-debugging can teach the LLM to perform rubber duck

debugging; i.e., without any human feedback on the code correctness or error messages, the model is able to identify its mistakes by investigating the execution results and explaining the generated code in natural language. Cao et al. [89] conducts a study of LLM's debugging ability for deep learning programs, including fault detection, fault localization and program repair.

2) *Bug Localization*: Wu et al. [85] compare the two LLMs (ChatGPT and GPT-4) with the existing fault localization techniques, and investigate the consistency of LLMs in fault localization, as well as how prompt engineering and the length of code context affect the results. Kang et al. [79] propose AutoFL, an automated fault localization technique that only requires a single failing test, and during its fault localization process, it also generates an explanation about why the given test fails. Yang et al. [84] propose LLMAO to overcome the left-to-right nature of LLMs by fine-tuning a small set of bidirectional adapter layers on top of the representations learned by LLMs, which can locate buggy lines of code without any test coverage information. Tu et al. [86] propose LLM4CBI to tame LLMs to generate effective test programs for finding suspicious files.

3) *Bug Reproduction*: There are also studies focusing on a sub-phase of the debugging process. For example, Kang et al. [78] and Plein et al. [81] respectively propose the framework to harness the LLM to reproduce bugs, and suggest bug reproducing test cases to the developer for facilitating debugging. Li et al. [87] focus on a similar aspect of finding the failure-inducing test cases whose test input can trigger the software's fault. It synergistically combines LLM and differential testing to do that.

There are also studies focusing on the bug reproduction of mobile apps to produce the replay script. Feng et al. [75] propose AdbGPT, a new lightweight approach to automatically reproduce the bugs from bug reports through prompt engineering, without any training and hard-coding effort. It leverages few-shot learning and chain-of-thought reasoning to elicit human knowledge and logical reasoning from LLMs to accomplish the bug replay in a manner similar to a developer. Huang et al. [71] propose CrashTranslator to automatically reproduce bugs directly from the stack trace. It accomplishes this by leveraging the LLM to predict the exploration steps for triggering the crash, and designing a reinforcement learning based technique to mitigate the inaccurate prediction and guide the search holistically. Taeb et al. [55] convert the manual accessibility test instructions into replayable, navigable videos by using LLM and UI element detection models, which can also help reveal accessibility issues.

4) *Error Explanation*: Taylor et al. [82] integrates the LLM into the Debugging C Compiler to generate unique, novice-focused explanations tailored to each error. Widjojo et al. [80] study the effectiveness of Stack Overflow and LLMs at explaining compiler errors.

F. Program Repair

This category denotes the task of fixing the identified software bugs. The high frequency of repair-related studies can be

attributed to the close relationship between this task and the source code. With their advanced natural language processing and understanding capabilities, LLM are well-equipped to process and analyze source code, making them an ideal tool for performing code-related tasks such as fixing bugs.

There have been template-based [143], heuristic-based [144], and constraint-based [145], [146] automatic program repair techniques. And with the development of deep learning techniques in the past few years, there have been several studies employing deep learning techniques for program repair. They typically adopt deep learning models to take a buggy software program as input and generate a patched program. Based on the training data, they would build a neural network model that learns the relations between the buggy code and the corresponding fixed code. Nevertheless, these techniques still fail to fix a large portion of bugs, and they typically have to generate hundreds to thousands of candidate patches and take hours to validate these patches to fix enough bugs. Furthermore, the deep learning based program repair models need to be trained with huge amounts of labeled training data (typically pairs of buggy and fixed code), which is time- and effort-consuming to collect the high-quality dataset. Subsequently, with the popularity and demonstrated capability of the LLMs, researchers begin to explore the LLMs for program repair.

1) *Patch Single-Line Bugs*: In the early era of program repair, the focus was mainly on addressing defects related to single-line code errors, which are relatively simple and did not require the repair of complex program logic. Lajkó et al. [95] propose to fine-tune the LLM with JavaScript code snippets to serve as the purpose for the JavaScript program repair. Zhang et al. [116] employs program slicing to extract contextual information directly related to the given buggy statement as repair ingredients from the corresponding program dependence graph, which makes the fine-tuning more focused on the buggy code. Zhang et al. [121] propose a stage-wise framework STEAM for patching single-line bugs, which simulates the interactive behavior of multiple programmers involved in bug management, e.g., bug reporting, bug diagnosis, patch generation, and patch verification.

Since most real-world bugs would involve multiple lines of code, and later studies explore these more complex situations (although some of them can also patch the single-line bugs).

2) *Patch Multiple-Lines Bugs*: The studies in this category would input a buggy function to the LLM, and the goal is to output the patched function, which might involve complex semantic understanding, code hunk modification, as well as program refactoring. Earlier studies typically employ the fine-tuning strategy to enable the LLM to better understand the code semantics. Fu et al. [123] fine-tune the LLM by employing BPE tokenization to handle Out-Of-Vocabulary (OOV) issues which makes the approach generate new tokens that never appear in a training function but are newly introduced in the repair. Wang et al. [120] train the LLM based on both buggy input and retrieved bug-fix examples which are retrieved in terms of the lexical and semantical similarities. The aforementioned studies (including the ones in patching single-line bugs) would predict the fixed programs directly, and Hu et al. [92] utilize

TABLE IV
PERFORMANCE OF PROGRAM REPAIR

Dataset	% Correct patches	LLM	Paper
Defects4J v1.2, Defects4J v2.0, QuixBugs, HumanEval-Java	22/40 Java bugs (QuixBugs dataset, with InCoder-6B, correct code infilling setting)	PLBART, CodeT5, CodeGen, InCoder (each with variant parameters, 10 LLMs in total)	[112]
QuixBugs	23/40 Python bugs, 14/40 Java bugs (complete function generation setting)	Codex-12B	[99]
Defects4J v1.2, Defects4J v2.0, QuixBugs, ManyBugs	39/40 Python bugs, 34/40 Java bugs (QuixBugs dataset, with Codex-12B, correct code infilling setting); 37/40 Python bugs, 32/40 Java bugs (QuixBugs dataset, with Codex-12B, complete function generation setting)	Codex, GPT-Neo, CodeT5, InCoder (each with variant parameters, 9 LLMs in total)	[92]
QuixBugs	31/40 Python bugs (completion function generation setting)	ChatGPT-175B	[95]
DL programs from Stack-Overflow	16/72 Python bugs (complete function generation setting)	ChatGPT-175B	[89]

Note that, for studies with multiple datasets or LLMs, we only present the best performance or in the most commonly utilized dataset.

a different setup that predicts the scripts that can fix the bugs when executed with the delete and insert grammar. For example, it predicts whether an original line of code should be deleted, and what content should be inserted.

Nevertheless, fine-tuning may face limitations in terms of its reliance on abundant high-quality labeled data, significant computational resources, and the possibility of overfitting. To approach the program repair problem more effectively, later studies focus on how to design an effective prompt for program repair. Several studies empirically investigate the effectiveness of prompt variants of the latest LLMs for program repair under different repair settings and commonly-used benchmarks (which will be explored in depth later), while other studies focus on proposing new techniques. Ribeiro et al. [109] take advantage of LLM to conduct the code completion in a buggy line for patch generation, and elaborate on how to circumvent the open-ended nature of code generation to appropriately fit the new code in the original program. Xia et al. [115] propose the conversation-driven program repair approach that interleaves patch generation with instant feedback to perform the repair in a conversational style. They first feed the LLM with relevant test failure information to start with, and then learns from both failures and successes of earlier patching attempts of the same bug for more powerful repair. For earlier patches that failed to pass all tests, they combine the incorrect patches with their corresponding relevant test failure information to construct a new prompt for the LLM to generate the next patch, in order to avoid making the same mistakes. For earlier patches that passed all the tests (i.e., plausible patches), they further ask the LLM to generate alternative variations of the original plausible patches. This can further build on and learn from earlier successes to generate more plausible patches to increase the chance of having correct patches. Zhang et al. [94] propose a similar approach design by leveraging multimodal prompts (e.g., natural language description, error message, input-output-based test cases), iterative querying, test-case-based few-shot selection to produce repairs. Moon et al. [102] propose for bug fixing with feedback. It consists of a critic model to generate feedback, an editor to edit codes based on the feedback, and a feedback selector to choose the best possible feedback from the critic.

Wei et al. [103] propose Repilot to copilot the AI “copilots” (i.e., LLMs) by synthesizing more valid patches during the repair process. Its key insight is that many LLMs produce outputs autoregressively (i.e., token by token), and by resembling human writing programs, the repair can be significantly boosted and guided through a completion engine. Brownlee et al. [105] propose to use the LLM as mutation operators for the search-based techniques of program repair.

3) *Repair With Static Code Analyzer*: Most of the program repair studies would suppose the bug has been detected, while Jin et al. [114] propose a program repair framework paired with a static analyzer to first detect the bugs, and then fix them. In detail, the static analyzer first detects an error (e.g., null pointer dereference) and the context information provided by the static analyzer will be sent into the LLM for querying the patch for this specific error. Wadhwa et al. [110] focus on a similar task, and additionally employ an LLM as the ranker to assess the likelihood of acceptance of generated patches which can effectively catch plausible but incorrect fixes and reduce developer burden.

4) *Repair for Specific Bugs*: The aforementioned studies all consider the buggy code as the input for the automatic program repair, while other studies conduct program repairing in terms of other types of bug descriptions, specific types of bugs, etc. Fakhoury et al. [122] focus on program repair from natural language issue descriptions, i.e., generating the patch with the bug and fix-related information described in the issue reports. Garg et al. [119] aim at repairing performance issues, in which they first retrieve a prompt instruction from a pre-constructed knowledge-base of previous performance bug fixes and then generate a repair prompt using the retrieved instruction. There are studies focusing on the bug fixing of Rust programs [108] or OCaml programs (an industrial-strength programming language) [111].

5) *Empirical Study About Program Repair*: There are several studies related to the empirical or experimental evaluation of the various LLMs on program repair, and we summarize the performance in Table IV. Jiang et al. [113], Xia et al. [93], and Zhang et al. [118] respectively conduct comprehensive experimental evaluations with various LLMs and on different automated program repair benchmarks, while other

researchers [89], [96], [98], [100] focus on a specific LLM and on one dataset, e.g., QuixBugs. In addition, Gao et al. [124] empirically investigate the impact of in-context demonstrations for bug fixing, including the selection, order, and number of demonstration examples. Prenner et al. [117] empirically study how the local context (i.e., code that comes before or after the bug location) affects the repair performance. Horváth et al. [99] empirically study the impact of program representation and model architecture on the repair performance.

There are two commonly-used repair settings when using LLMs to generate patches: 1) complete function generation (i.e., generating the entire patch function), 2) correct code infilling (i.e., filling in a chunk of code given the prefix and suffix), and different studies might utilize different settings which are marked in Table IV. The commonly-used datasets are QuixBugs, Defects4J, etc. These datasets only involve the fundamental functionalities such as sorting algorithms, each program's average number of lines ranging from 13 to 22, implementing one functionality, and involving few dependencies. To tackle this, Cao et al. [89] conducts an empirical study on a more complex dataset with DL programs collected from StackOverflow. Every program contains about 46 lines of code on average, implementing several functionalities including data preprocessing, DL model construction, model training, and evaluation. And the dataset involves more than 6 dependencies for each program, including TensorFlow, Keras, and Pytorch. Their results demonstrate a much lower rate of correct patches than in other datasets, which again reveals the potential difficulty of this task. Similarly, Haque et al. [106] introduce a dataset comprising of buggy code submissions and their corresponding fixes collected from online judge platforms, in which it offers an extensive collection of unit tests to enable the evaluations about the correctness of fixes and further information regarding time, memory constraints, and acceptance based on a verdict.

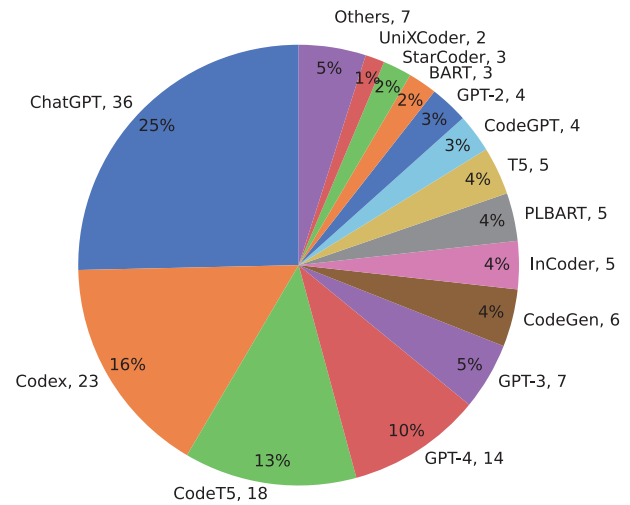
V. ANALYSIS FROM LLM PERSPECTIVE

This section discusses the analysis based on the viewpoints of LLM, specifically, it's unfolded from the viewpoints of utilized LLMs, types of prompt engineering, input of the LLMs, as well as the accompanied techniques when utilizing LLM.

A. LLM Models

As shown in Fig. 6, the most commonly utilized LLM in software testing tasks is ChatGPT, which was released on Nov. 2022 by OpenAI. It is trained on a large corpus of natural language text data, and primarily designed for natural language processing and conversation. ChatGPT is the most widely recognized and popular LLM up until now, known for its exceptional performance across various tasks. Therefore, it comes as no surprise that it ranks in the top position in terms of our collected studies.

Codex, an LLM based on GPT-3, is the second most commonly used LLM in our collected studies. It is trained on a massive code corpus containing examples from many programming languages such as JavaScript, Python, C/C++, and Java. Codex was released on Sep. 2021 by OpenAI and powers



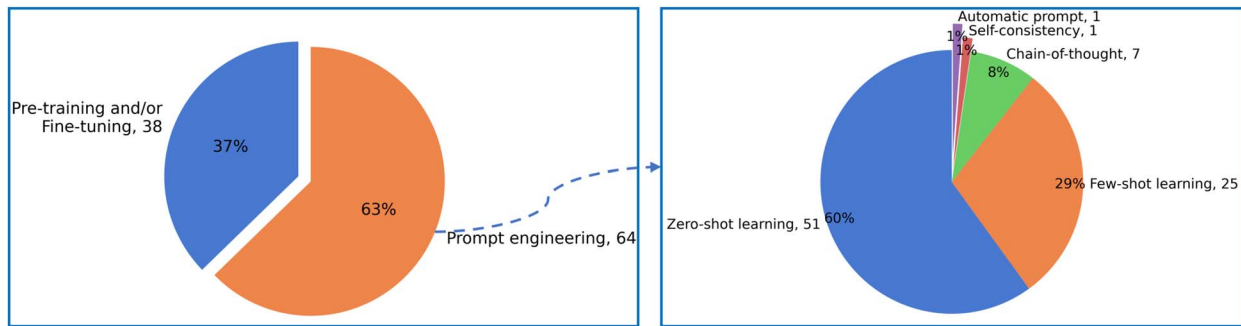


Fig. 7. Distribution about how LLM is used (Note that, a study can involve multiple types of prompt engineering).

(e.g., UI screenshots, programming screencasts) in software testing tasks.

B. Types of Prompt Engineering

As shown in Fig. 7, among our collected studies, 38 studies utilize the LLMs through pre-training or fine-tuning schema, while 64 studies employ the prompt engineering to communicate with LLMs to steer its behavior for desired outcomes without updating the model weights. When using the early LLMs, their performances might not be as impressive, so researchers often use pre-training or fine-tuning techniques to adjust the models for specific domains and tasks in order to improve their performance. Then with the upgrading of LLM technology, especially with the introduction of GPT-3 and later LLMs, the knowledge contained within the models and their understanding/inference capability has increased significantly. Therefore, researchers will typically rely on prompt engineering to consider how to design appropriate prompts to stimulate the model's knowledge.

Among the 64 studies with prompt engineering, 51 studies involve zero-shot learning, and 25 studies involve few-shot learning (a study may involve multiple types). There are also studies involving the chain-of-thought (7 studies), self-consistency (1 study), and automatic prompt (1 study).

Zero-shot learning is to simply feed the task text to the model and ask for results. Many of the collected studies employ the Codex, CodeT5, and CodeGen (as shown in Section V-A), which is already trained on source code. Hence, for the tasks dealing with source code like unit test case generation and program repair as demonstrated in previous sections, directly querying the LLM with prompts is the common practice. There are generally two types of manners of zero-shot learning, i.e., with and without instructions. For example, Xie et al. [36] would provide the LLMs with the instructions as “please help me generate a JUnit test for a specific Java method...” to facilitate the unit test case generation. In contrast, Siddiq et al. [39] only provide the code header of the unit test case (e.g., “class \${className}\${suffix}Test {”), and the LLMs would carry out the unit test case generation automatically. Generally speaking, prompts with clear instructions will yield more accurate results, while prompts without instructions are typically suitable for very specific situations.

Few-shot learning presents a set of high-quality demonstrations, each consisting of both input and desired output, on the target task. As the model first sees the examples, it can better understand human intention and criteria for what kinds of answers are wanted, which is especially important for tasks that are not so straightforward or intuitive to the LLM. For example, when conducting the automatic test generation from general bug reports, Kang et al. [78] provide examples of bug reports (questions) and the corresponding bug reproducing tests (answers) to the LLM, and their results show that two examples can achieve the highest performance than no examples or other number of examples. Another example of test assertion generation, Nashid et al. [47] provide demonstrations of the focal method, the test method containing an `<AssertPlaceholder>`, and the expected assertion, which enables the LLMs to better understand the task.

Chain-of-thought (CoT) prompting generates a sequence of short sentences to describe reasoning logics step by step (also known as reasoning chains or rationales) to the LLMs for generating the final answer. For example, for program repair from the natural language issue descriptions [122], given the buggy code and issue report, the authors first ask the LLM to localize the bug, and then they ask it to explain why the localized lines are buggy, finally, they ask the LLM to fix the bug. Another example is for generating unusual programs for fuzzing deep learning libraries, Deng et al. [58] first generate a possible “bug” (bug description) before generating the actual “bug-triggering” code snippet that invokes the target API. The predicted bug description provides an additional hint to the LLM, indicating that the generated code should try to cover specific potential buggy behavior.

Self-consistency involves evaluating the coherence and consistency of the LLM's responses on the same input in different contexts. There is one study with this prompt type, and it is about debugging. Kang et al. [83] employ a hypothesize-observe-conclude loop, which first generates a hypothesis about what the bug is and constructs an experiment to verify, using an LLM, then decide whether the hypothesis is correct based on the experiment result (with a debugger or code execution) using an LLM, after that, depending on the conclusion, it either starts with a new hypothesis or opts to terminate the debugging process and generate a fix.

Automatic prompt aims to automatically generate and select the appropriate instruction for the LLMs, instead of requiring

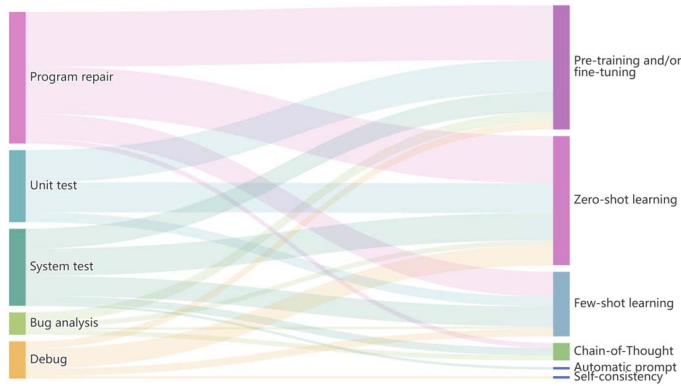


Fig. 8. Mapping between testing tasks and how LLMs are used.

the user to manually engineer a prompt. Xia et al. [67] introduce an auto-prompting step that automatically distills all user-provided inputs into a concise and effective prompt for fuzzing. Specifically, they first generate a list of candidate prompts by incorporating the user inputs and auto prompting instruction while setting the LLM at high temperature, then a small-scale fuzzing experiment is conducted to evaluate each candidate prompt, and the best one is selected.

Note that there are fourteen studies that apply the iterative prompt design when using zero-shot or few-shot learning, in which the approach continuously refines the prompts with the running information of the testing task, e.g., the test failure information. For example, for program repair, Xia et al. [115] interleave patch generation with test validation feedback to prompt future generation iteratively. In detail, they incorporate various information from a failing test including its name, the relevant code line(s) triggering the test failure, and the error message produced in the next round of prompting which can help the model understand the failure reason and provide guidance towards generating the correct fix. Another example is for mobile GUI testing, Liu et al. [14] iteratively query the LLM about the operation (e.g., click a button, enter a text) to be conducted in the mobile app, and at each iteration, they would provide the LLM with current context information like which GUI pages and widgets have just explored.

Mapping between testing tasks and how LLMs are used. Fig. 8 demonstrates the mapping between the testing tasks (mentioned in Section IV) and how LLMs are used (as introduced in this subsection). The unit test case generation and program repair share similar patterns of communicating with the LLMs, since both tasks are closely related to the source code. Typically, researchers utilize pre-training and/or fine-tuning and zero-shot learning methods for these two tasks. Zero-shot learning is suitable because these tasks are relatively straightforward and can be easily understood by LLMs. Moreover, since the training data for these two tasks can be automatically collected from source code repositories, pre-training and/or fine-tuning methods are widely employed for these two tasks, which can enhance LLMs' understanding of domain-specific knowledge.

In comparison, for system test input generation, zero-shot learning and few-shot learning methods are commonly used.

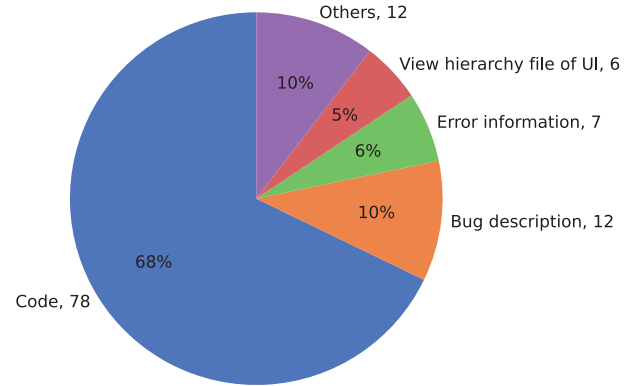


Fig. 9. Input of LLM.

This might be because this task often involves generating specific types of inputs, and demonstrations in few-shot learning can assist the LLMs in better understanding what should be generated. Besides, for this task, the utilization of pre-training and/or fine-tuning methods are not as widespread as in unit test case generation and program repair. This might be attributed to the fact that training data for system testing varies across different software and is relatively challenging to collect automatically.

C. Input of LLM

We also find that different testing tasks or software under test might involve diversified input when querying the LLM, as demonstrated in Fig. 9.

The most commonly utilized input is the **source code** since a large portion of collected studies relate to program repair or unit test case generation whose input are source code. For unit test case generation, typical code-related information would be (i) the complete focal method, including the signature and body; (ii) the name of the focal class (i.e., the class that the focal method belongs to); (iii) the field in the focal class; and (iv) the signatures of all methods defined in the focal class [7], [26]. For program repair, there can be different setups and involve different inputs, including (i) inputting a buggy function with the goal of outputting the patched function, (ii) inputting the buggy location with the goal of generating the correct replacement code (can be a single line change) given the prefix and suffix of the buggy function [93]. Besides, there can be variations for the buggy location input, i.e., (i) does not contain the buggy lines (but the bug location is still known), (ii) give the buggy lines as lines of comments.

There are also 12 studies taking the **bug description** as input for the LLM. For example, Kang et al. [78] take the bug description as input when querying LLM and let the LLM generate the bug-reproducing test cases. Fakhoury et al. [122] input the natural language descriptions of bugs to the LLM, and generate the correct code fixes.

There are 7 studies that would provide the **intermediate error information**, e.g., test failure information, to the LLM, and would conduct the iterative prompt (as described in Section V-B) to enrich the context provided to the LLM. These

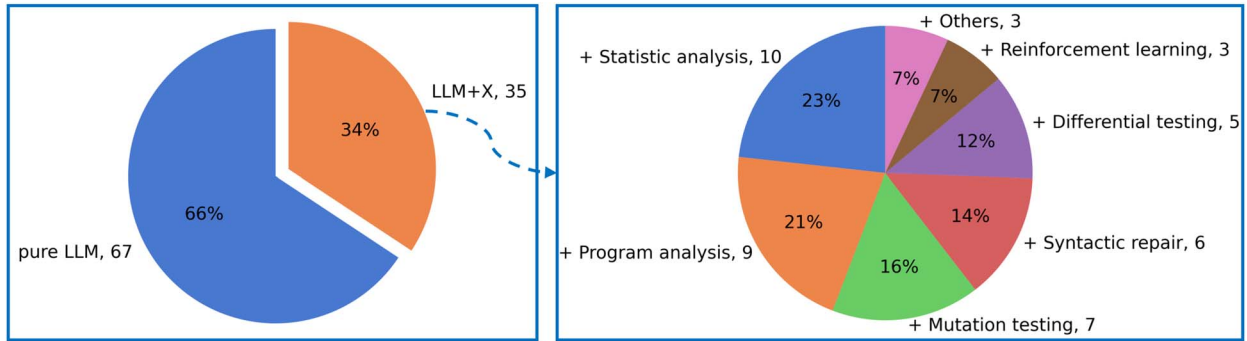


Fig. 10. Distribution about other techniques incorporated with LLMs (Note that, a study can involve multiple types).

studies are related to the unit test case generation and program repair, since in these scenarios, the running information can be acquired easily.

When testing mobile apps, since the utilized LLM could not understand the image of the GUI page, the **view hierarchy file** which represents the details of the GUI page usually acts as the input to LLMs. Nevertheless, with the emergence of GPT-4 which is a multimodal model and accepts both image and text inputs for model input, the GUI screenshots might be directly utilized for LLM's input.

D. Incorporating Other Techniques With LLM

There are divided opinions on whether LLM has reached an all-powerful status that requires no other techniques. As shown in Fig. 10, among our collected studies, 67 of them utilize LLMs to address the entire testing task, while 35 studies incorporate additional techniques. These techniques include mutation testing, differential testing, syntactic checking, program analysis, statistical analysis, etc.

The reason why researchers still choose to combine LLMs with other techniques might be because, despite exhibiting enormous potential in various tasks, LLMs still possess limitations such as comprehending code semantics and handling complex program structures. Therefore, combining LLMs with other techniques optimizes their strengths and weaknesses to achieve better outcomes in specific scenarios. In addition, it is important to note that while LLMs are capable of generating correct code, they may not necessarily produce sufficient test cases to check for edge cases or rare scenarios. This is where mutation and other testing techniques come into play, as they allow for the generation of more diverse and complex code that can better simulate real-world scenarios. Taken in this sense, a testing approach can incorporate a combination of different techniques, including both LLMs and other testing strategies, to ensure comprehensive coverage and effectiveness.

1) *LLM + Statistical Analysis*: As LLMs can often generate a multitude of outputs, manually sifting through and identifying the correct output can be overwhelmingly laborious. As such, researchers have turned to statistical analysis techniques like ranking and clustering [28], [45], [78], [93], [116] to efficiently filter through LLM's outputs and ultimately obtain more accurate results.

2) *LLM + Program Analysis*: When utilizing LLMs to accomplish tasks such as generating unit test cases and repairing software code, it is important to consider that software code inherently possesses structural information, which may not be fully understood by LLMs. Hence, researchers often utilize program analysis techniques, including code abstract syntax trees (ASTs) [74], to represent the structure of code more effectively and increase the LLM's ability to comprehend the code accurately. Researchers also perform the structure-based subsetting of code lines to narrow the focus for LLM [94], or extract additional code context from other code files [7], to enable the models to focus on the most task-relevant information in the codebase and lead to more accurate predictions.

3) *LLM + Mutation Testing*: It is mainly targeting at generating more diversified test inputs. For example, Deng et al. [59] first use LLM to generate the seed programs (e.g., code snippets using a target DL API) for fuzzing deep learning libraries. To enrich the pool of these test programs, they replace parts of the seed program with masked tokens using mutation operators (e.g., replaces the API call arguments with the span token) to produce masked inputs, and again utilize the LLMs to perform code infilling to generate new code that replaces the masked tokens.

4) *LLM + Syntactic Checking*: Although LLMs have shown remarkable performance in various natural language processing tasks, the generated code from these models can sometimes be syntactically incorrect, leading to potential errors and reduced usability. Therefore, researchers have proposed to leverage syntax checking to identify and correct errors in the generated code. For example, in their work for unit test case generation, Alagarsamy et al. [29] additionally introduce a verification method to check and repair the naming consistency (i.e., revising the test method name to be consistent with the focal method name) and the test signatures (i.e., adding missing keywords like public, void, or @test annotations). Xie et al. [36] also validates the generated unit test case and employs rule-based repair to fix syntactic and simple compile errors.

5) *LLM + Differential Testing*: Differential testing is well-suited to find semantic or logic bugs that do not exhibit explicit erroneous behaviors like crashes or assertion failures. In this category of our collected studies, the LLM is mainly responsible

for generating valid and diversified inputs, while the differential testing helps to determine whether there is a triggered bug based on the software's output. For example, Ye et al. [48] first uses LLM to produce random JavaScript programs, and leverages the language specification document to generate test data, then conduct the differential testing on JavaScript engines such as JavaScriptCore, ChakraCore, SpiderMonkey, QuickJS, etc. There are also studies utilizing the LLMs to generate test inputs and then conduct differential testing for fuzzing DL libraries [58], [59] and SAT solvers [63]. Li et al. [87] employs the LLM in finding the failure-inducing test cases. In detail, given a program under test, they first request the LLM to infer the intention of the program, then request the LLM to generate programs that have the same intention, which are alternative implementations of the program, and are likely free of the program's bug. Then they perform the differential testing with the program under test and the generated programs to find the failure-inducing test cases.

VI. CHALLENGES AND OPPORTUNITIES

Based on the above analysis from the viewpoints of software testing and LLM, we summarize the challenges and opportunities when conducting software testing with LLM.

A. Challenges

As indicated by this survey, software testing with LLMs has undergone significant growth in the past two years. However, it is still in its early stages of development, and numerous challenges and open questions need to be addressed.

1) *Challenges for Achieving High Coverage:* Exploring the diverse behaviors of the software under test to achieve high coverage is always a significant concern in software testing. In this context, test generation differs from code generation, as code generation primarily focuses on producing a single, correct code snippet, whereas software testing requires generating diverse test inputs to ensure better coverage of the software. Although setting a high temperature can facilitate the LLMs in generating different outputs, it remains challenging for LLMs to directly achieve the required diversity. For example, for unit test case generation, in SF110 dataset, the line coverage is merely 2% and the branch coverage is merely 1% [39]. For system test input generation, in terms of fuzzing DL libraries, the API coverage for TensorFlow is reported to be 66% (2215/3316) [59].

From our collected studies, we observe that the researchers often 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. [59] 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. Liu et al. [65] leverage LLM to produce the test generators (each of which can yield a batch of unusual text inputs under the same

mutation rule) together with the mutation rules for text-oriented fuzzing, which reduces the human effort required for designing mutation rules.

A 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.

2) *Challenges in Test Oracle Problem:* The oracle problem has been a longstanding challenge in various testing applications, e.g., testing machine learning systems [148] and testing deep learning libraries [59]. To alleviate the oracle problem to the overall testing activities, a common practice in our collected studies is to transform it into a more easily derived form, often by utilizing differential testing [63] or focusing on only identifying crash bugs [14].

There are successful applications of differential testing with LLMs, as shown in Fig. 10. For instance, when testing the SMT solvers, Sun et al. adopt differential testing which involves comparing the results of multiple SMT solvers (i.e., Z3, cvc5, and Bitwuzla) on the same generated test formulas by LLM [63]. However, this approach is limited to systems where counterpart software or running environment can easily be found, potentially restricting its applicability. Moreover, to mitigate the oracle problem, other studies only focus on the crash bugs which are easily observed automatically. This is particularly the case for mobile applications testing, in which the LLMs guide the testing in exploring more diversified pages, conducting more complex operational actions, and covering more meaningful operational sequences [14]. However, this significantly restricts the potential of utilizing the LLMs for uncovering various types of software bugs.

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. Luu et al. [56] have examined the effectiveness of LLM in generating metamorphic relations, yet they only experiment with straightforward prompts by directly querying ChatGPT. Further exploration, potentially incorporating human-computer interaction or domain knowledge, is highly encouraged. Another promising avenue is exploring the capability of LLMs to automatically generate test cases based on metamorphic relations, covering a wide range of inputs.

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.

3) *Challenges for Rigorous Evaluations:* The lack of benchmark datasets and the potential data leakage issues associated with LLM-based techniques present challenges in conducting rigorous evaluations and comprehensive comparisons of proposed methods.

For program repair, there are only two well-known and commonly-used benchmarks, i.e., Defect4J and QuixBugs, as demonstrated in Table IV. Furthermore, these datasets are not specially designed for testing the LLMs. For example, as reported by Xia et al. [93], 39 out of 40 Python bugs in the QuixBugs dataset can be fixed by Codex, yet in real-world practice, the successful fix rate can be nowhere near as high. For unit test case generation, there are no widely recognized benchmarks, and different studies would utilize different datasets for performance evaluation, as demonstrated in Table III. This indicates the need to build more specialized and diversified benchmarks.

Furthermore, the LLMs may have seen the widely-used benchmarks in their pre-training data, i.e., data leakage issues. Jiang et al. [113] check the CodeSearchNet and BigQuery, which are the data sources of common LLMs, and the results show that four repositories used by the Defect4J benchmark are also in CodeSearchNet, and the whole Defects4J repository is included by BigQuery. Therefore, it is very likely that existing program repair benchmarks are seen by the LLMs during pre-training. This data leakage issue has also been investigated in machine learning-related studies. For example, Tu et al. [149] focus on the data leakage in issue tracking data, and results show that information leaked from the “future” makes prediction models misleadingly optimistic. This reminds us that the performance of LLMs on software testing tasks may not be as good as reported in previous studies. It also suggests that we need more specialized datasets that are not seen by LLMs to serve as benchmarks. One way is to collect it from specialized sources, e.g., user-generated content from niche online communities.

4) *Challenges in Real-World Application of LLMs in Software Testing:* As we mentioned in Section V-B, in the early days of using LLMs, pre-training and fine-tuning are commonly used practice, considering the model parameters are relatively few resulting in weaker model capabilities (e.g., T5). As time progressed, the number of model parameters increased significantly, leading to the emergence of models with greater capabilities (e.g., ChatGPT). And in recent studies, prompt engineering has become a common approach. However, 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 is quite challenging for these models to achieve similar performance to what our collected papers have reported. For instance, in the widely-used QuixBugs dataset, it has been reported that 39 out of 40 Python bugs and 34 out of 40 Java bugs can be automatically fixed [93]. However, when it comes to DL programs collected from Stack Overflow, which represent real-world coding practice, only 16 out of 72 Python bugs can be automatically fixed [89].

Recent research has highlighted the importance of high-quality training data in improving the performance of models for code-related tasks [150], yet manually building high-quality

organization-specific datasets for training or fine-tuning is time-consuming and labor-intensive. To address this, one is encouraged to utilize the automated techniques of mining software repositories to build the datasets, for example, techniques like key information extraction techniques from Stack Overflow [151] offer potential solutions for automatically gathering relevant data.

In addition, exploring the methodology for better fine-tuning the LLMs with software-specific data is worth considering because software-specific data differs from natural language data as it contains more structural information, such as data flow and control flow. Previous research on code representations has shown the benefits of incorporating data flow, which captures the semantic-level structure of code and represents the relationship between variables in terms of “whether-value-comes-from” [152]. These insights can provide valuable guidance for effectively fine-tuning LLMs with software-specific data.

B. Opportunities

There are also many research opportunities in software testing with LLMs, which can greatly benefit developers, users, and the research community. While not necessarily challenges, these opportunities contribute to advancements in software testing, benefiting practitioners and the wider research community.

1) *Exploring LLMs in the Early Stage of Testing:* As shown in Fig. 4, LLMs have not been used in the early stage of testing, e.g., test requirements, and test planning. There might be two main reasons behind that. The first is the subjectivity in early-stage testing tasks. Many tasks in the early stages of testing, such as requirements gathering, test plan creation, and design reviews, may involve subjective assessments that require significant input from human experts. This could make it less suitable for LLMs that rely heavily on data-driven approaches. The second might be the lack of open-sourced data in the early stages. Unlike in later stages of testing, there may be limited data available online during early-stage activities. This could mean that LLMs may not have seen much of this type of data, and therefore may not perform well on these tasks.

Adopting a human-computer interaction schema for tackling early-stage testing tasks would harness the domain-specific knowledge of human developers and leverage the general knowledge embedded in LLMs. Additionally, it is highly encouraged for software development companies to record and provide access to early-stage testing data, allowing for improved training and performance of LLMs in these critical testing activities.

2) *Exploring LLMs in Other Testing Phases:* We have analyzed the distribution of testing phases for the collected studies. As shown in Fig 11, we can observe that LLMs are most commonly used in unit testing, followed by system testing. However, there is still no research on the use of LLMs in integration testing and acceptance testing.

For integration testing, it involves testing the interfaces between different software modules. In some software organizations, integration testing might be merged with unit testing, which can be a possible reason why LLM is rarely utilized in

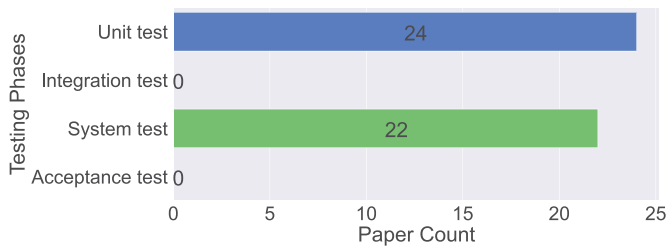


Fig. 11. Distribution of testing phases (note that we omit the studies which do not explicitly specify the testing phases, e.g., program repair).

integration testing. Another reason might be that the size and complexity of the input data in this circumstance may exceed the capacity of the LLM to process and analyze (e.g., the source code of all involved software modules), which can lead to errors or unreliable results. To tackle this, a potential reference can be found in Section IV-A, where Xie et al. [36] design a method to organize the necessary information into the pre-defined maximum prompt token limit of the LLM. Furthermore, integration testing requires diversified data to be generated to sufficiently test the interface among multiple modules. As mentioned in Section IV-C, previous work has demonstrated the LLM's capability in generating diversified test input for system testing, in conjunction with mutation testing techniques [48], [59]. And these can provide insights about generating the diversified interface data for integration testing.

Acceptance testing is usually conducted by business analysts or end-users to validate the system's functionality and usability, which requires more non-technical language and domain-specific knowledge, thus making it challenging to apply LLM effectively. Since acceptance testing involves humans, it is well-suited for the use of human-in-the-loop schema with LLMs. This has been studied in traditional machine learning [153], but has not yet been explored with LLMs. Specifically, the LLMs can be responsible for automatically generating test cases, evaluating test coverage, etc, while human testers are responsible for checking the program's behavior and verifying test oracle.

3) *Exploring LLMs for More Types of Software*: We analyze what types of software have been explored in the collected studies, as shown in Fig. 5. Note that, since a large portion of studies are focused on unit testing or program repair, they are conducted on publicly available datasets and do not involve specific software types.

From the analysis in Section IV-C, the LLM can generate not only the source code for testing DL libraries but also the textual input for testing mobile apps, even the models for testing CPS. Overall, the LLM provides a flexible and powerful framework for generating test inputs for a wide range of applications. Its versatility would make it useful for testing the software in other domains.

From one point of view, some proposed techniques can be applied to other types of software. For example, in the paper proposed for testing deep learning libraries [58], since it proposes techniques for generating diversified, complicated, and human-like DL programs, the authors state that the approach can be easily extended to test software systems from other

application domains, e.g., interpreters, database systems, and other popular libraries. More than that, there are already studies that focus on universal fuzzing techniques [52], [67] which are designed to be adaptable and applicable to different types of test inputs and software.

From another point of view, other types of software can also benefit from the capabilities of LLMs to design the testing techniques that are better suited to their specific domain and characteristics. For instance, the metaverse, with its immersive virtual environments and complex interactions, presents unique challenges for software testing. LLMs can be leveraged to generate diverse and realistic inputs that mimic user behavior and interactions within the metaverse, which are never explored.

4) *Exploring LLMs for Non-Functional Testing*: In our collected studies, LLMs are primarily used for functional testing, and no practice in performance testing, usability testing or others. One possible reason for the prevalence of LLM-based solutions in functional testing is that they can convert functional testing problems into code generation or natural language generation problems [14], [59], which LLMs are particularly adept at solving.

On the other hand, performance testing and usability testing may require more specialized models that are designed to detect and analyze specific types of data, handle complex statistical analyses, or determine the buggy criteria. Moreover, there have been dozens of performance testing tools (e.g., LoadRunner [154]) that can generate a workload that simulates real-world usage scenarios and achieve relatively satisfactory performance.

The potential opportunities might let the LLM integrate the performance testing tools and acts like the LangChain [155], to better simulate different types of workloads based on real user behavior. Furthermore, the LLMs can identify the parameter combinations and values that have the highest potential to trigger performance problems. It is essentially a way to rank and prioritize different parameter settings based on their impact on performance and improve the efficiency of performance testing.

5) *Exploring Advanced Prompt Engineering*: There are a total of 11 commonly used prompt engineering techniques as listed in a popular prompt engineering guide [156], as shown in Fig. 12. Currently, in our collected studies, only the first five techniques are being utilized. The more advanced techniques have not been employed yet, and can be explored in the future for prompt design.

For instance, multimodal chain of thought prompting involves using diverse sensory and cognitive cues to stimulate thinking and creativity in LLMs [157]. By providing images (e.g., GUI screenshots) or audio recordings related to the software under test can help the LLM better understand the software's context and potential issues. Besides, try to prompt the LLM to imagine itself in different roles, such as a developer, user, or quality assurance specialist. This perspective-shifting exercise enables the LLM to approach software testing from multiple viewpoints and uncover different aspects that might require attention or investigation.

Graph prompting [158] involves the representation of information using graphs or visual structures to facilitate understanding and problem-solving. Graph prompting can be a natural

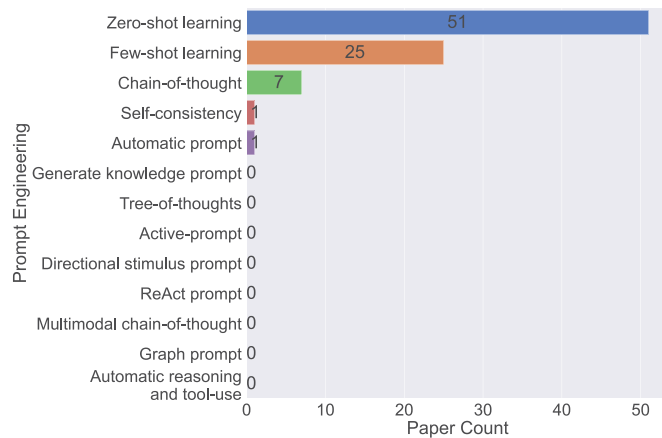


Fig. 12. List of advanced prompt engineering practices and those utilized in the collected papers.

match with software engineering, consider it involves various dependencies, control flow, data flow, state transitions, or other relevant graph structure. Graph prompting can be beneficial in analyzing this structural information, and enabling the LLMs to comprehend the software under test effectively. For instance, testers can use graph prompts to visualize test coverage, identify untested areas or paths, and ensure adequate test execution.

6) *Incorporating LLMs With Traditional Techniques*: There is currently no clear consensus on the extent to which LLMs can solve software testing problems. From the analysis in Section V-D, we have seen some promising results from studies that have combined LLMs with traditional software testing techniques. This implies the LLMs are not the sole silver bullet for software testing. Considering the availability of many mature software testing techniques and tools, and the limited capabilities of LLMs, it is necessary to explore other better ways to combine LLMs with traditional testing or program analysis techniques and tools for better software testing.

Based on the collected studies, the LLMs have been successfully utilized together with various techniques such as differential testing (e.g., [63]), mutation testing (e.g., [59]), program analysis (e.g., [104]), as shown in Fig. 10. From one perspective, future studies can explore improved integration of these traditional techniques with LLMs. Take mutation testing as an example, current practices mainly rely on the human-designed mutation rules to mutate the candidate tests, and let the LLMs re-generate new tests [38], [59], [67], while Liu et al. directly utilize the LLMs for producing the mutation rules alongside the mutated tests [65]. Further explorations in this direction are of great interest.

From another point of view, more traditional techniques can be incorporated in LLMs for software testing. For instance, besides the aforementioned traditional techniques, the LLMs have been combined with formal verification for self-healing software detection in the field of software security [159]. More attempts are encouraged. Moreover, considering the existence of numerous mature software testing tools, one can explore the integration of LLMs with these tools, allowing them to act as a “LangChain” to better explore the potential of these tools.

VII. RELATED WORK

The systematic literature review is a crucial manner for gaining insights into the current trends and future directions within a particular field. It enables us to understand and stay updated on the developments in that domain.

Wang et al. surveyed the machine learning and deep learning techniques for software engineering [160]. Yang et al. and Watson et al. respectively carried out surveys about the use of deep learning in software engineering domain [161], [162]. Bajammal et al. surveyed the utilization of computer vision techniques to improve software engineering tasks [163]. Zhang et al. provided a survey of techniques for testing machine learning systems [150].

With the advancements of artificial intelligence and LLMs, researchers also conduct systematic literature reviews about LLMs, and their applications in various fields (e.g., software engineering). Zhao et al. [17] reviewed recent advances in LLMs by providing an overview of their background, key findings, and mainstream techniques. They focused on four major aspects of LLMs, namely pre-training, adaptation tuning, utilization, and capacity evaluation. Additionally, they summarized the available resources for developing LLMs and discuss the remaining issues for future directions. Hou et al. conducted a systematic literature review on using LLMs for software engineering, with a particular focus on understanding how LLMs can be exploited to optimize processes and outcomes [164]. Fan et al. conducted a survey of LLMs for software engineering, and set out open research challenges for the application of LLMs to technical problems faced by software engineers [165]. Zan et al. conducted a survey of existing LLMs for NL2Code task (i.e., generating code from a natural language description), and reviewed benchmarks and metrics [166].

While these studies either targeted the broader software engineering domain (with a limited focus on software testing tasks) or focused on other software development tasks (excluding software testing), this paper specifically focuses on the use of LLMs for software testing. It surveys related studies, summarizes key challenges and potential opportunities, and serves as a roadmap for future research in this area.

VIII. CONCLUSION

This paper provides a comprehensive review of the use of LLMs in software testing. We have analyzed relevant studies that have utilized LLMs in software testing from both the software testing and LLMs perspectives. This paper also highlights the challenges and potential opportunities in this direction. Results of this review demonstrate that LLMs have been successfully applied in a wide range of testing tasks, including unit test case generation, test oracle generation, system test input generation, program debugging, and program repair. However, challenges still exist in achieving high testing coverage, addressing the test oracle problem, conducting rigorous evaluations, and applying LLMs in real-world scenarios. Additionally, it is observed that LLMs are commonly used in only a subset of the entire testing lifecycle, for example, they are primarily utilized in the middle and later stages of testing, only serving

the unit and system testing phases, and only for functional testing. This highlights the research opportunities for exploring the uncovered areas. Regarding how the LLMs are utilized, we find that various pre-training/fine-tuning and prompt engineering methods have been developed to enhance the capabilities of LLMs in addressing testing tasks. However, more advanced techniques in prompt design have yet to be explored and can be an avenue for future research.

It can serve as a roadmap for future research in this area, identifying gaps in our current understanding of the use of LLMs in software testing and highlighting potential avenues for exploration. We believe that the insights provided in this paper will be valuable to both researchers and practitioners in the field of software engineering, assisting them in leveraging LLMs to improve software testing practices and ultimately enhance the quality and reliability of software systems.

REFERENCES

- [1] G. J. Myers, T. Badgett, T. M. Thomas, and C. Sandler, *The Art of Software Testing*, 2nd ed. Hoboken, NJ, USA: Wiley, 2004.
- [2] M. Pezze and M. Young, *Software Testing and Analysis—Process, Principles and Techniques*. Hoboken, NJ, USA: Wiley, 2007.
- [3] M. Harman and P. McMinn, “A theoretical and empirical study of search-based testing: Local, global, and hybrid search,” *IEEE Trans. Softw. Eng.*, vol. 36, no. 2, pp. 226–247, Mar./Apr. 2010.
- [4] P. Delgado-Pérez, A. Ramírez, K. J. Valle-Gómez, I. Medina-Bulo, and J. R. Romero, “InterEvo-TR: Interactive evolutionary test generation with readability assessment,” *IEEE Trans. Softw. Eng.*, vol. 49, no. 4, pp. 2580–2596, Apr. 2023.
- [5] X. Xiao, S. Li, T. Xie, and N. Tillmann, “Characteristic studies of loop problems for structural test generation via symbolic execution,” in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Silicon Valley, CA, USA, E. Denney, T. Bultan, and A. Zeller, Eds., Piscataway, NJ, USA: IEEE Press, Nov. 2013, pp. 246–256.
- [6] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, “Feedback-directed random test generation,” in *Proc. 29th Int. Conf. Softw. Eng. (ICSE)*, Minneapolis, MN, USA, Los Alamitos, CA, USA: IEEE Comput. Soc. Press, May 2007, pp. 75–84.
- [7] Z. Yuan, et al., “No more manual tests? Evaluating and improving chatGPT for unit test generation,” 2023, *arXiv:2305.04207*.
- [8] Y. Tang, Z. Liu, Z. Zhou, and X. Luo, “ChatGPT vs SBST: A comparative assessment of unit test suite generation,” 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2307.00588>
- [9] Android Developers, “Ui/application exerciser monkey,” 2012. Accessed: Dec. 27, 2023. [Online]. Available: <https://developer.android.google.cn/studio/test/other-testing-tools/monkey>
- [10] Y. Li, Z. Yang, Y. Guo, and X. Chen, “DroidBot: A lightweight UI-guided test input generator for android,” in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. Companion (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2017, pp. 23–26.
- [11] T. Su et al., “Guided, stochastic model-based gui testing of android apps,” in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 245–256.
- [12] Z. Dong, M. Böhme, L. Cojocar, and A. Roychoudhury, “Time-travel testing of android apps,” in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2020, pp. 481–492.
- [13] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, “Reinforcement learning based curiosity-driven testing of android applications,” in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2020, pp. 153–164.
- [14] Z. Liu et al., “Make LLM a testing expert: Bringing human-like interaction to mobile GUI testing via functionality-aware decisions,” 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2310.15780>
- [15] T. Su, J. Wang, and Z. Su, “Benchmarking automated GUI testing for android against real-world bugs,” in *Proc. 29th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, Athens, Greece, New York, NY, USA: ACM, Aug. 2021, pp. 119–130.
- [16] M. Shanahan, “Talking about large language models,” 2022. [Online]. Available: <https://doi.org/10.48550/arXiv.2212.03551>
- [17] W. X. Zhao et al., “A survey of large language models,” 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2303.18223>
- [18] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, “Large language models are zero-shot reasoners,” in *Proc. NeurIPS*, 2022. [Online]. Available: http://papers.nips.cc/paper_files/paper/2022/hash/8bb0d291acd4acf06ef112099c16f326-Abstract-Conference.html
- [19] J. Wei et al., “Chain-of-thought prompting elicits reasoning in large language models,” in *Proc. NeurIPS*, 2022. [Online]. Available: http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html
- [20] J. Li, G. Li, Y. Li, and Z. Jin, “Structured chain-of-thought prompting for code generation,” 2023, *arXiv:2305.06599*.
- [21] J. Li, Y. Li, G. Li, Z. Jin, Y. Hao, and X. Hu, “Skocoder: A sketch-based approach for automatic code generation,” in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE)*, 2023, pp. 2124–2135.
- [22] J. Li, Y. Zhao, Y. Li, G. Li, and Z. Jin, “AceCoder: Utilizing existing code to enhance code generation,” 2023, *arXiv:2303.17780*.
- [23] Y. Dong, X. Jiang, Z. Jin, and G. Li, “Self-collaboration code generation via chatGPT” 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2304.07590>
- [24] S. Pan, L. Luo, Y. Wang, C. Chen, J. Wang, and X. Wu, “Unifying large language models and knowledge graphs: A roadmap,” 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2306.08302>
- [25] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, “Unit test case generation with transformers and focal context,” 2020, *arXiv:2009.05617*.
- [26] B. Chen et al., “Codet: Code generation with generated tests,” 2022, *arXiv:2207.10397*.
- [27] S. K. Lahiri et al., “Interactive code generation via test-driven user-intent formalization,” 2022, *arXiv:2208.05950*.
- [28] S. Alagarsamy, C. Tantithamthavorn, and A. Aleti, “A3test: Assertion-augmented automated test case generation,” 2023, *arXiv:2302.10352*.
- [29] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, “An empirical evaluation of using large language models for automated unit test generation,” *IEEE Trans. Softw. Eng.*, vol. 50, no. 1, pp. 85–105, Jan. 2024.
- [30] V. Guilherme and A. Vincenzi, “An initial investigation of chatGPT unit test generation capability,” in *Proc. 8th Brazilian Symp. Systematic Automated Softw. Testing (SAST)*, Campo Grande, Brazil, A. L. Fontão et al., Eds., New York, NY, USA: ACM, Sep. 2023, pp. 15–24.
- [31] S. Hashtroudi, J. Shin, H. Hemmati, and S. Wang, “Automated test case generation using code models and domain adaptation,” 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2308.08033>
- [32] L. Plein, W. C. Ouédraogo, J. Klein, and T. F. Bissyandé, “Automatic generation of test cases based on bug reports: A feasibility study with large language models,” 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2310.06320>
- [33] V. Vikram, C. Lemieux, and R. Padhye, “Can large language models write good property-based tests?” 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2307.04346>
- [34] N. Rao, K. Jain, U. Alon, C. L. Goues, and V. J. Hellendoorn, “CAT-LM training language models on aligned code and tests,” in *Proc. 38th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Luxembourg, Piscataway, NJ, USA: IEEE Press, Sep. 2023, pp. 409–420, doi: 10.1109/ASE56229.2023.00193.
- [35] Z. Xie, Y. Chen, C. Zhi, S. Deng, and J. Yin, “Chatunitest: A chatGPT-based automated unit test generation tool,” 2023, *arXiv:2305.04764*.
- [36] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, “CodaMosa: Escaping coverage plateaus in test generation with pre-trained large language models,” in *Proc. Int. Conf. Softw. Eng. (ICSE)*, 2023, pp. 919–931.
- [37] A. M. Dakhel, A. Nikanjam, V. Majdinasab, F. Khomh, and M. C. Desmarais, “Effective test generation using pre-trained large language models and mutation testing,” 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2308.16557>
- [38] M. L. Siddiq, J. Santos, R. H. Tanvir, N. Ulfat, F. A. Rifat, and V. C. Lopes, “Exploring the effectiveness of large language models in generating unit tests,” 2023, *arXiv:2305.00418*.
- [39] Y. Zhang, W. Song, Z. Ji, D. Yao, and N. Meng, “How well does LLM generate security tests?” 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2310.00710>
- [40] V. Li and N. Doiron, “Prompting code interpreter to write better unit tests on quixbugs functions,” 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2310.00483>

- [41] B. Steenhoek, M. Tufano, N. Sundaresan, and A. Svyatkovskiy, "Reinforcement learning from automatic feedback for high-quality unit test generation," 2023, *arXiv:2310.02368*.
- [42] S. Bhatia, T. Gandhi, D. Kumar, and P. Jalote, "Unit test generation using generative AI: A comparative performance analysis of autogeneration tools," 2023, *arXiv:2312.10622*.
- [43] M. Tufano, D. Drain, A. Svyatkovskiy, and N. Sundaresan, "Generating accurate assert statements for unit test cases using pretrained transformers," in *Proc. 3rd ACM/IEEE Int. Conf. Automat. Softw. Test*, 2022, pp. 54–64.
- [44] P. Nie, R. Banerjee, J. J. Li, R. J. Mooney, and M. Gligoric, "Learning deep semantics for test completion," 2023, *arXiv:2302.10166*.
- [45] A. Mastropaolo et al., "Using transfer learning for code-related tasks," *IEEE Trans. Softw. Eng.*, vol. 49, no. 4, pp. 1580–1598, Apr. 2023, doi: 10.1109/TSE.2022.3183297.
- [46] N. Nashid, M. Sintaha, and A. Mesbah, "Retrieval-based prompt selection for code-related few-shot learning," in *Proc. 45th Int. Conf. Softw. Eng. (ICSE)*, 2023, pp. 2450–2462.
- [47] G. Ye et al., "Automated conformance testing for javascript engines via deep compiler fuzzing," in *Proc. 42nd ACM SIGPLAN Int. Conf. Program. Lang. Des. Implementation*, 2021, pp. 435–450.
- [48] Z. Liu et al., "Fill in the blank: Context-aware automated text input generation for mobile gui testing," 2022, *arXiv:2212.04732*.
- [49] M. R. Taesiri, F. Macklon, Y. Wang, H. Shen, and C.-P. Bezemer, "Large language models are pretty good zero-shot video game bug detectors," 2022, *arXiv:2210.02506*.
- [50] S. L. Shrestha and C. Csallner, "SIGPT: Using transfer learning to directly generate simulink model files and find bugs in the simulink toolchain," in *Proc. Eval. Assessment Softw. Eng.*, 2021, pp. 260–265.
- [51] J. Hu, Q. Zhang, and H. Yin, "Augmenting greybox fuzzing with generative AI," 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2306.06782>
- [52] A. Mathur, S. Pradhan, P. Soni, D. Patel, and R. Regunathan, "Automated test case generation using t5 and GPT-3," in *Proc. 9th Int. Conf. Adv. Comput. Commun. Syst. (ICACCS)*, vol. 1, 2023, pp. 1986–1992.
- [53] D. Zimmermann and A. Koziolok, "Automating GUI-based software testing with GPT-3," in *Proc. IEEE Int. Conf. Softw. Testing, Verification Validation Workshops (ICSTW)*, 2023, pp. 62–65.
- [54] M. Taeb, A. Swearngin, E. Schoop, R. Cheng, Y. Jiang, and J. Nichols, "Axnav: Replaying accessibility tests from natural language," 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2310.02424>
- [55] Q. Luu, H. Liu, and T. Y. Chen, "Can chatGPT advance software testing intelligence? An experience report on metamorphic testing," 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2310.19204>
- [56] A. Khanfir, R. Degiovanni, M. Papadakis, and Y. L. Traon, "Efficient mutation testing via pre-trained language models," 2023, *arXiv:2301.03543*.
- [57] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzGPT," 2023, *arXiv:2304.02014*.
- [58] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large language models are zero shot fuzzers: Fuzzing deep learning libraries via large language models," 2023, *arXiv:2209.11515*.
- [59] J. Ackerman and G. Cybenko, "Large language models for fuzzing parsers (registered report)," in *Proc. 2nd Int. Fuzzing Workshop (FUZZING)* Seattle, WA, USA, M. Böhme, Y. Noller, B. Ray, and L. Szekeres, Eds., New York, NY, USA: ACM, Jul. 2023, pp. 31–38, doi: 10.1145/3605157.3605173.
- [60] S. Yu, C. Fang, Y. Ling, C. Wu, and Z. Chen, "LLM for test script generation and migration: Challenges, capabilities, and opportunities," 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2309.13574>
- [61] G. Deng et al., "PentestGPT: An llm-empowered automatic penetration testing tool," 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2308.06782>
- [62] M. Sun, Y. Yang, Y. Wang, M. Wen, H. Jia, and Y. Zhou, "SMT solver validation empowered by large pre-trained language models," in *Proc. 38th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Luxembourg, Piscataway, NJ, USA: IEEE Press, 2023, pp. 1288–1300, doi: 10.1109/ASE56229.2023.00180.
- [63] Y. Deng, J. Yao, Z. Tu, X. Zheng, M. Zhang, and T. Zhang, "Target: Automated scenario generation from traffic rules for testing autonomous vehicles," 2023. [Online]. Available: <https://api.semanticscholar.org/CorpusID:258588387>
- [64] Z. Liu et al., "Testing the limits: Unusual text inputs generation for mobile app crash detection with large language model," 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2310.15657>
- [65] C. Zhang et al., "Understanding large language model based fuzz driver generation," 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2307.12469>
- [66] C. Xia, M. Paltenghi, J. Tian, M. Pradel, and L. Zhang, "Universal fuzzing via large language models," 2023. [Online]. Available: <https://api.semanticscholar.org/CorpusID:260735598>
- [67] C. Tsigkanos, P. Rani, S. Müller, and T. Kehrler, "Variable discovery with large language models for metamorphic testing of scientific software," in *Proc. 23rd Int. Conf. Comput. Sci. (ICCS)*, Prague, Czech Republic, J. Mikyska, C. de Mulatier, M. Paszynski, V. V. Krzhizhanovskaya, J. J. Dongarra, and P. M. A. Sloot, Eds., vol. 14073. Springer, Jul. 2023, pp. 321–335, doi: 10.1007/978-3-031-35995-8_23.
- [68] C. Yang et al., "White-box compiler fuzzing empowered by large language models," 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2310.15991>
- [69] T. Zhang, I. C. Irsan, F. Thung, D. Han, D. Lo, and L. Jiang, "iTiger: An automatic issue title generation tool," in *Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2022, pp. 1637–1641.
- [70] Y. Huang et al., "Crashtranslator: Automatically reproducing mobile application crashes directly from stack trace," 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2310.07128>
- [71] T. Zhang, I. C. Irsan, F. Thung, and D. Lo, "Cupid: Leveraging chatGPT for more accurate duplicate bug report detection," 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2308.10022>
- [72] U. Mukherjee and M. M. Rahman, "Employing deep learning and structured information retrieval to answer clarification questions on bug reports," 2023, *arXiv:2304.12494*.
- [73] P. Mahbub, O. Shuvo, and M. M. Rahman, "Explaining software bugs leveraging code structures in neural machine translation," 2022, *arXiv:2212.04584*.
- [74] S. Feng and C. Chen, "Prompting is all your need: Automated android bug replay with large language models," 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2306.01987>
- [75] Y. Su, Z. Han, Z. Gao, Z. Xing, Q. Lu, and X. Xu, "Still confusing for bug-component triaging? Deep feature learning and ensemble setting to rescue," in *Proc. 31st IEEE/ACM Int. Conf. Program Comprehension (ICPC)*, Melbourne, Australia, Piscataway, NJ, USA: IEEE Press, May 2023, pp. 316–327, doi: 10.1109/ICPC58990.2023.00046.
- [76] N. D. Bui, Y. Wang, and S. Hoi, "Detect-localize-repair: A unified framework for learning to debug with codet5," 2022, *arXiv:2211.14875*.
- [77] S. Kang, J. Yoon, and S. Yoo, "Large language models are few-shot testers: Exploring llm-based general bug reproduction," 2022, *arXiv:2209.11515*.
- [78] S. Kang, G. An, and S. Yoo, "A preliminary evaluation of LLM-based fault localization," 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2308.05487>
- [79] P. Widjojo and C. Treude, "Addressing compiler errors: Stack overflow or large language models?" 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2307.10793>
- [80] L. Plein and T. F. Bissyandé, "Can LLMs demystify bug reports?" 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2310.06310>
- [81] A. Taylor, A. Vassar, J. Renzella, and H. A. Pearce, "DCC—Help: Generating context-aware compiler error explanations with large language models," 2023. [Online]. Available: <https://api.semanticscholar.org/CorpusID:261076439>
- [82] S. Kang, B. Chen, S. Yoo, and J.-G. Lou, "Explainable automated debugging via large language model-driven scientific debugging," 2023, *arXiv:2304.02195*.
- [83] A. Z. H. Yang, R. Martins, C. L. Goues, and V. J. Hellendoorn, "Large language models for test-free fault localization," 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2310.01726>
- [84] Y. Wu, Z. Li, J. M. Zhang, M. Papadakis, M. Harman, and Y. Liu, "Large language models in fault localisation," 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2308.15276>
- [85] H. Tu, Z. Zhou, H. Jiang, I. N. B. Yusuf, Y. Li, and L. Jiang, "LLM4CBI: Taming llms to generate effective test programs for compiler bug isolation," 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2307.00593>

- [86] T.-O. Li et al., "Nuances are the key: Unlocking chatGPT to find failure-inducing tests with differential prompting," in *Proc. 38th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2023, pp. 14–26.
- [87] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching large language models to self-debug," 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2304.05128>
- [88] J. Cao, M. Li, M. Wen, and S.-c. Cheung, "A study on prompt design, advantages and limitations of chatGPT for deep learning program repair," 2023, *arXiv:2304.08191*.
- [89] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *Proc. IEEE Symp. Secur. Privacy (SP)*, Los Alamitos, CA, USA: IEEE Comput. Soc., 2022, pp. 1–18.
- [90] Z. Fan, X. Gao, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models," 2022, *arXiv:2205.10583*.
- [91] Y. Hu, X. Shi, Q. Zhou, and L. Pike, "Fix bugs with transformer through a neural-symbolic edit grammar," 2022, *arXiv:2204.06643*.
- [92] C. S. Xia, Y. Wei, and L. Zhang, "Practical program repair in the era of large pre-trained language models," 2022, *arXiv:2210.14179*.
- [93] J. Zhang et al., "Repairing bugs in python assignments using large language models," 2022, *arXiv:2209.14876*.
- [94] M. Lajkó, V. Csuik, and L. Vidács, "Towards javascript program repair with generative pre-trained transformer (GPT-2)," in *Proc. 3rd Int. Workshop Automated Program Repair*, 2022, pp. 61–68.
- [95] D. Sobania, M. Briesch, C. Hanna, and J. Petke, "An analysis of the automatic bug fixing performance of chat," 2023, *arXiv:2301.08653*.
- [96] K. Huang et al., "An empirical study on fine-tuning large language models of code for automated program repair," in *Proc. 38th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Luxembourg, Piscataway, NJ, USA: IEEE Press, Sep. 2023, pp. 1162–1174, doi: 10.1109/ASE56229.2023.00181.
- [97] M. C. Wuisang, M. Kurniawan, K. A. Wira Santosa, A. Agung Santoso Gunawan, and K. E. Saputra, "An evaluation of the effectiveness of openai's chatGPT for automated python program bug fixing using quixbugs," in *Proc. Int. Seminar Appl. Technol. Inf. Commun. (iSemantic)*, 2023, pp. 295–300.
- [98] D. Horváth, V. Csuik, T. Gyimóthy, and L. Vidács, "An extensive study on model architecture and program representation in the domain of learning-based automated program repair," in *Proc. IEEE/ACM Int. Workshop Automated Program Repair (APR@ICSE)*, Melbourne, Australia, Piscataway, NJ, USA: IEEE Press, May 2023, pp. 31–38, doi: 10.1109/APR59189.2023.00013.
- [99] J. A. Prenner, H. Babii, and R. Robbes, "Can openai's codex fix bugs? An evaluation on quixbugs," in *Proc. 3rd Int. Workshop Automated Program Repair*, 2022, pp. 69–75.
- [100] W. Yuan et al., "Circle: Continual repair across programming languages," in *Proc. 31st ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2022, pp. 678–690.
- [101] S. Moon et al., "Coffee: Boost your code llms by fixing bugs with feedback," 2023, *arXiv:2311.07215*.
- [102] Y. Wei, C. S. Xia, and L. Zhang, "Copiloting the copilots: Fusing large language models with completion engines for automated program repair," in *Proc. 31st ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, San Francisco, CA, USA, S. Chandra, K. Blincoe, and P. Tonella, Eds., New York, NY, USA: ACM, Dec. 2023, pp. 172–184, doi: 10.1145/3611643.3616271.
- [103] Y. Peng, S. Gao, C. Gao, Y. Huo, and M. R. Lyu, "Domain knowledge matters: Improving prompts with fix templates for repairing python type errors," 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2306.01394>
- [104] A. E. I. Brownlee et al., "Enhancing genetic improvement mutations using large language models," in *Proc. 15th Int. Symp. Search-Based Softw. Eng. (SSBSE)*, San Francisco, CA, USA, P. Arcaini, T. Yue, and E. M. Fredericks, Eds., vol. 14415. Cham, Switzerland: Springer Nature, Dec. 2023, pp. 153–159, doi: 10.1007/978-3-031-48796-5_13.
- [105] M. M. A. Haque, W. U. Ahmad, I. Lourentzou, and C. Brown, "Fix-eval: Execution-based evaluation of program fixes for programming problems," in *Proc. IEEE/ACM Int. Workshop Automated Program Repair (APR@ICSE)*, Melbourne, Australia, Piscataway, NJ, USA: IEEE Press, May 2023, pp. 11–18, doi: 10.1109/APR59189.2023.00009.
- [106] B. Ahmad, S. Thakur, B. Tan, R. Karri, and H. Pearce, "Fixing hardware security bugs with large language models," 2023, *arXiv:2302.01215*.
- [107] P. Deligiannis, A. Lal, N. Mehrotra, and A. Rastogi, "Fixing rust compilation errors using LLMs," 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2308.05177>
- [108] F. Ribeiro, R. Abreu, and J. Saraiva, "Framing program repair as code completion," in *Proc. 3rd Int. Workshop Automated Program Repair*, 2022, pp. 38–45.
- [109] N. Wadhwa et al., "Frustrated with code quality issues? LLMs can help!" 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2309.12938>
- [110] F. Ribeiro, J. N. C. de Macedo, K. Tsushima, R. Abreu, and J. Saraiva, "GPT-3-powered type error debugging: Investigating the use of large language models for code repair," in *Proc. 16th ACM SIGPLAN Int. Conf. Softw. Lang. Eng. (SLE)*, Cascais, Portugal, J. Saraiva, T. Degueule, and E. Scott, Eds., New York, NY, USA: ACM, Oct. 2023, pp. 111–124, doi: 10.1145/3623476.3623522.
- [111] Y. Wu et al., "How effective are neural networks for fixing security vulnerabilities," 2023, *arXiv:2305.18607*.
- [112] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," 2023, *arXiv:2302.05020*.
- [113] M. Jin et al., "Inferfix: End-to-end program repair with LLMs," 2023, *arXiv:2303.07263*.
- [114] C. S. Xia and L. Zhang, "Keep the conversation going: Fixing 162 out of 337 bugs for \$0.42 each using chatGPT," 2023, *arXiv:2304.00385*.
- [115] Y. Zhang, G. Li, Z. Jin, and Y. Xing, "Neural program repair with program dependence analysis and effective filter mechanism," 2023, *arXiv:2305.09315*.
- [116] J. A. Prenner and R. Robbes, "Out of context: How important is local context in neural program repair?" 2023, *arXiv:2312.04986*.
- [117] Q. Zhang, C. Fang, B. Yu, W. Sun, T. Zhang, and Z. Chen, "Pre-trained model-based automated software vulnerability repair: How far are we?" 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2308.12533>
- [118] S. Garg, R. Z. Moghaddam, and N. Sundaresan, "Rapgen: An approach for fixing code inefficiencies in zero-shot," 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2306.17077>
- [119] W. Wang, Y. Wang, S. Joty, and S. C. H. Hoi, "Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair," in *Proc. 31st ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, San Francisco, CA, USA, S. Chandra, K. Blincoe, and P. Tonella, Eds., New York, NY, USA: ACM, Dec. 2023, pp. 146–158, doi: 10.1145/3611643.3616256.
- [120] Y. Zhang, Z. Jin, Y. Xing, and G. Li, "STEAM: Simulating the interactive behavior of programmers for automatic bug fixing," 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2308.14460>
- [121] S. Fakhoury, S. Chakraborty, M. Musuvathi, and S. K. Lahiri, "Towards generating functionally correct code edits from natural language issue descriptions," 2023, *arXiv:2304.03816*.
- [122] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, "Vul-repair: A t5-based automated software vulnerability repair," in *Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2022, pp. 935–947.
- [123] S. Gao, X. Wen, C. Gao, W. Wang, H. Zhang, and M. R. Lyu, "What makes good in-context demonstrations for code intelligence tasks with LLMs?" in *Proc. 38th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Luxembourg, Piscataway, NJ, USA: IEEE Press, Sep. 2023, pp. 761–773, doi: 10.1109/ASE56229.2023.00109.
- [124] C. Treude and H. Hata, "She elicits requirements and he tests: Software engineering gender bias in large language models," 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2303.10131>
- [125] R. Kocielnik, S. Prabhumoye, V. Zhang, R. M. Alvarez, and A. Anandkumar, "Autobiastest: Controllable sentence generation for automated and open-ended social bias testing in language models," 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2302.07371>
- [126] M. Ciniselli, L. Pascarella, and G. Bavota, "To what extent do deep learning-based code recommenders generate predictions by cloning code from the training set?" in *Proc. 19th IEEE/ACM Int. Conf. Mining Softw. Repositories (MSR)*, Pittsburgh, PA, USA, New York, NY, USA: ACM, 2022, pp. 167–178, doi: 10.1145/3524842.3528440.
- [127] D. Erhabor, S. Udayashankar, M. Nagappan, and S. Al-Kiswani, "Measuring the runtime performance of code produced with GitHub copilot," 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2305.06439>

- [128] R. Wang, R. Cheng, D. Ford, and T. Zimmermann, "Investigating and designing for trust in AI-powered code generation tools," 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2305.11248>
- [129] B. Yetistiren, I. Özsoy, M. Ayerdem, and E. Tüzün, "Evaluating the code quality of AI-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and ChatGPT," 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2304.10778>
- [130] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proc. 18th Int. Conf. Eval. Assessment in Softw. Eng. (EASE)*, London, U.K., M. J. Shepperd, T. Hall, and I. Myrteit, Eds., New York, NY, USA: ACM, May 2014, pp. 38: 1–38:10, doi: 10.1145/2601248.2601268.
- [131] A. Mastropaolo et al., "Studying the usage of text-to-text transfer transformer to support code-related tasks," in *Proc. 43rd IEEE/ACM Int. Conf. Softw. Eng. (ICSE)*, Madrid, Spain, Piscataway, NJ, USA: IEEE Press, May 2021, pp. 336–347.
- [132] C. Tsigkanos, P. Rani, S. Müller, and T. Kehrer, "Large language models: The next frontier for variable discovery within metamorphic testing?" in *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Taipa, Macao, T. Zhang, X. Xia, and N. Novelli, Eds., Piscataway, NJ, USA: IEEE Press, Mar. 2023, pp. 678–682, doi: 10.1109/SANER56733.2023.00070.
- [133] P. Farrell-Vinay, *Manage Software Testing*. New York, NY, USA: Auerbach, 2008.
- [134] A. Mili and F. Tchie, *Software Testing: Concepts and Operations*. Hoboken, NJ, USA: Wiley, 2015.
- [135] S. Lukaszczuk and G. Fraser, "Pynguin: Automated unit test generation for python," in *Proc. 44th IEEE/ACM Int. Conf. Softw. Eng., (ICSE) Companion*, Pittsburgh, PA, USA, ACM/IEEE Press, May 2022, pp. 168–172, doi: 10.1145/3510454.3516829.
- [136] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Trans. Softw. Eng.*, vol. 41, no. 5, pp. 507–525, May 2015.
- [137] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On learning meaningful assert statements for unit test cases," in *Proc. 42nd Int. Conf. Softw. Eng. (ICSE)*, Seoul, South Korea, G. Rothermel and D. Bae, Eds., New York, NY, USA: ACM, Jun./Jul. 2020, pp. 1398–1409.
- [138] Y. He et al., "Textexerciser: Feedback-driven text input exercising for android applications," in *Proc. IEEE Symp. Secur. Privacy (SP)*, San Francisco, CA, USA, Piscataway, NJ, USA: IEEE Press, May 2020, pp. 1071–1087.
- [139] A. Wei, Y. Deng, C. Yang, and L. Zhang, "Free lunch for testing: Fuzzing deep-learning libraries from open source," in *Proc. 44th IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*, Pittsburgh, PA, USA, New York, NY, USA: ACM, May 2022, pp. 995–1007.
- [140] D. Xie et al., "Doctor: Documentation-guided fuzzing for testing deep learning API functions," in *Proc. 31st ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA)*, Virtual Event, South Korea, S. Ryu and Y. Smaragdakis, Eds., New York, NY, USA: ACM, Jul. 2022, pp. 176–188.
- [141] Q. Guo et al., "Audee: Automated testing for deep learning frameworks," in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Melbourne, Australia, Piscataway, NJ, USA: IEEE Press, Sep. 2020, pp. 486–498.
- [142] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang, "Deep learning library testing via effective model generation," in *Proc. 28th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, Virtual Event, USA, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds., New York, NY, USA: ACM, Nov. 2020, pp. 788–799.
- [143] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, New York, NY, USA: ACM, 2018, pp. 298–309, doi: 10.1145/3213846.3213871.
- [144] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proc. 40th Int. Conf. Softw. Eng.*, New York, NY, USA: ACM, 2018, pp. 1–11, doi: 10.1145/3180155.3180233.
- [145] Y. Xiong et al., "Precise condition synthesis for program repair," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. (ICSE)*, 2017, pp. 416–426.
- [146] J. Xuan et al., "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 34–55, Jan. 2017.
- [147] S. Song, X. Li, and S. Li, "How to bridge the gap between modalities: A comprehensive survey on multimodal large language model," 2023, *arXiv:2311.07594*.
- [148] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *IEEE Trans. Softw. Eng.*, vol. 48, no. 2, pp. 1–36, Jan. 2022.
- [149] F. Tu, J. Zhu, Q. Zheng, and M. Zhou, "Be careful of when: An empirical study on time-related misuse of issue tracking data," in *Proc. ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/SIGSOFT FSE)*, Lake Buena Vista, FL, USA, G. T. Leavens, A. Garcia, and C. S. Pasareanu, Eds., New York, NY, USA: ACM, Nov. 2018, pp. 307–318, doi: 10.1145/3236024.3236054.
- [150] Z. Sun, L. Li, Y. Liu, X. Du, and L. Li, "On the importance of building high-quality training datasets for neural code search," in *Proc. 44th IEEE/ACM Int. Conf. Softw. Eng. (ICSE)*, Pittsburgh, PA, USA, Piscataway, NJ, USA: ACM, May 2022, pp. 1609–1620, doi: 10.1145/3510003.3510160.
- [151] L. Shi et al., "ISPY: Automatic issue-solution pair extraction from community live chats," in *Proc. 36th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Melbourne, Australia, Piscataway, NJ, USA: IEEE Press, Nov. 2021, pp. 142–154, doi: 10.1109/ASE51524.2021.967894.
- [152] D. Guo et al., "GraphCodeBERT: Pre-training code representations with data flow," in *Proc. 9th Int. Conf. Learn. Representations (ICLR)*, Virtual Event, Austria, May 2021. [Online]. Available: <https://openreview.net/forum?id=jLoC4ez43PZ>
- [153] F. Yu, A. Seff, Y. Zhang, S. Song, T. Funkhouser, and J. Xiao, "Lsun: Construction of a large-scale image dataset using deep learning with humans in the loop," 2015, *arXiv:1506.03365*.
- [154] "Loadrunner, Inc." Accessed: Dec. 27, 2023. [Online]. Available: microfocus.com
- [155] "Langchain, Inc." Accessed: Dec. 27, 2023. [Online]. Available: <https://docs.langchain.com/docs/>
- [156] Prompt Engineering. "Prompt engineering guide." GitHub. Accessed: Dec. 27, 2023. [Online]. Available: <https://github.com/dair-ai/Prompt-Engineering-Guide>
- [157] Z. Zhang, A. Zhang, M. Li, H. Zhao, G. Karypis, and A. Smola, "Multimodal chain-of-thought reasoning in language models," 2023, *arXiv:2302.00923*.
- [158] Z. Liu, X. Yu, Y. Fang, and X. Zhang, "Graphprompt: Unifying pre-training and downstream tasks for graph neural networks," in *Proc. ACM Web Conf. (WWW)*, Austin, TX, USA, Y. Ding, J. Tang, J. F. Sequeda, L. Aroyo, C. Castillo, and G. Houben, Eds., New York, NY, USA: ACM, Apr./May 2023, pp. 417–428.
- [159] Y. Charalambous, N. Tihanyi, R. Jain, Y. Sun, M. A. Ferrag, and L. C. Cordeiro, "A new era in software security: Towards self-healing software via large language models and formal verification," 2023, *arXiv:2305.14752*.
- [160] S. Wang et al., "Machine/deep learning for software engineering: A systematic literature review," *IEEE Trans. Softw. Eng.*, vol. 49, no. 3, pp. 1188–1231, Mar. 2023, doi: 10.1109/TSE.2022.3173346.
- [161] Y. Yang, X. Xia, D. Lo, and J. C. Grundy, "A survey on deep learning for software engineering," *ACM Comput. Surv.*, vol. 54, no. 10s, pp. 206: 1–206:73, 2022, doi: 10.1145/3505243.
- [162] C. Watson, N. Cooper, D. Nader-Palacio, K. Moran, and D. Poshyvanyk, "A systematic literature review on the use of deep learning in software engineering research," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 2, pp. 32:1–32:58, 2022, doi: 10.1145/3485275.
- [163] M. Bajammal, A. Stocco, D. Mazinanian, and A. Mesbah, "A survey on the use of computer vision to improve software engineering tasks," *IEEE Trans. Softw. Eng.*, vol. 48, no. 5, pp. 1722–1742, May 2022, doi: 10.1109/TSE.2020.3032986.
- [164] X. Hou et al., "Large language models for software engineering: A systematic literature review," 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2308.10620>
- [165] A. Fan et al., "Large language models for software engineering: Survey and open problems," 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2310.03533>
- [166] D. Zan et al., "Large language models meet NL2Code: A survey," in *Proc. 61st Annu. Meeting Assoc. Comput. Linguistics (Vol. 1, Long Papers) (ACL)*, Toronto, ON, Canada, A. Rogers, J. L. Boyd-Graber, and N. Okazaki, Eds., Association for Computational Linguistics, Jul. 2023, pp. 7443–7464, doi: 10.18653/v1/2023.acl-long.411.

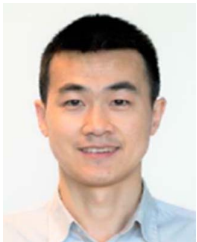


Junjie Wang (Member, IEEE) received the Ph.D. degree from ISCAS, in 2015. She is a Research Professor with the Institute of Software, Chinese Academy of Sciences (ISCAS). She was a Visiting Scholar with North Carolina State University from September 2017 to September 2018 and worked with Prof. Tim Menzies. Her research interests include AI for software engineering, software testing, and software analytics. She has more than 50 high-quality publications including IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, ICSE, TOSEM, FSE,

and ASE, and five of them has received the Distinguished/Best Paper Award, respectively at ICSE 2019, ICSE 2020, and ICPC 2022. She is currently serving as an Associate Editor of IEEE TRANSACTIONS ON SOFTWARE ENGINEERING. For more information, see <https://peopleucas.edu.cn/0058217?language=en>



Yuchao Huang is a Doctoral Student with the Institute of Software Research, Chinese Academy of Sciences (ISCAS). His research interests include software engineering, mobile testing, and deep learning. He has published four papers at top-tier international software engineering conferences, including ICSE and FSE. Specifically, he applies AI and LLM technology in the AI(LLM)-assisted automated mobile GUI bug replay.



Chunyang Chen received the bachelor's degree from BUPT, China and the Ph.D. degree from NTU, Singapore. He is a Full Professor with the School of Computation, Information and Technology, Technical University of Munich, Germany. His main research interest includes automated software engineering, especially data-driven mobile app development. Besides, he is also interested in human-computer interaction and software security. His research has won awards including ACM SIGSOFT Early Career Researcher Award, Facebook Research

Award, four ACM SIGSOFT Distinguished Paper Awards (ICSE'23/21/20, ASE'18), and multiple best paper/demo awards. For more information, see <https://chunyang-chen.github.io/>



Zhe Liu received the Ph.D. degree from the University of Chinese Academy of Sciences, in 2023. He is an Assistant Researcher with the Institute of Software Chinese Academy of Sciences. His research interests include software engineering, mobile testing, deep learning, and human-computer interaction. He has published 15 papers at top-tier international software engineering conferences/journals, including IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, ICSE, CHI, and ASE. Specifically, he applies AI and light-weight program

analysis technology in the following directions: AI(LLM)-assisted automated mobile GUI testing, usability, and bug replay, human machine collaborative testing including testing guide for testers, AI-empowered mining software repository including issue report mining. He received the ACM Student Research Competition (SRC) 2023 Grand Finals Winners, 1st Place, Graduate Category. For more information, see <https://zheliu6.github.io/>



Song Wang (Member, IEEE) received the dual B.E. degrees from Sichuan University, the master's degree from the Institute of Software Chinese Academy of Sciences, and the Ph.D. degree from the University of Waterloo. He is an Assistant Professor with York University, Canada. He worked at the intersection of software engineering and artificial intelligence. He has more than 50 high-quality publications including IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, ICSE, TOSEM, FSE, and ASE, and is the recipient of four Distinguished/Best Paper Awards. He is currently serving as an Associate Editor of ACM Transactions on Software Engineering (TOSEM). For more information, see <https://www.eecs.yorku.ca/wangsong/index.html>



Qing Wang (Member, IEEE) is a Research Professor with the Institute of Software Chinese Academy of Sciences (ISCAS). She is also the Deputy Chief Engineer of ISCAS, and the Director of State Key Laboratory of Intelligent Game of ISCAS. Her research lies in the area of software process, software quality assurance, and artificial intelligence for software engineering. She currently serves as the member of the International Software and Systems Processes Association (ISSPA), the member of the International Software Engineering Research Network (ISERN), the Editorial Board of *Information and Software Technology Journal* (IST) and *Journal of Software Evolution and Process* (JSEP), the Deputy Chair of Software Quality and Testing Group in China National Information Technology Standardization (SAC/TC28/SC7/WG1), and the CMMI lead appraisal. She has edited/co-edited five books, and published more than 100 papers in high-level conferences and journals.