Software Testing with Large Language Model: Survey, Landscape, and Vision

Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, Qing Wang

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, and 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 52 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 ones. 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

1 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.

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.

- *J. Wang,Y. Huang, Z. Liu, Q. Wang are with State Key Laboratory of Intelligent Game, Institute of Software Chinese Academy of Sciences, and University of Chinese Academy of Sciences, Beijing, China. E-mail: {junjie, yuchao2019, liuzhe2020, wq}@iscas.ac.cn*
- *C. Chen is with Monash University, Melbourne, Australia E-mail: chunyang.chen@monash.edu*
- *S. Wang is with York University, Toronto, Canada. E-mail: wangsong@yorku.ca*

While the field of software testing has gained significant popularity, there still remain dozens of challenges that have not been effectively addressed. For example, one such challenge is automated unit test cases generation. Although various approaches, including search-based [1], [2], constraint-based [3] or random-based [4] techniques to generate a suite of unit tests, the coverage and the meaningfulness of the generated tests are still far from satisfactory. Similarly, when it comes to mobile GUI testing, existing studies with random-/rule-based methods [5], [6], model-based methods [7], [8], and learning-based methods [9] are unable to understand the semantic information of the GUI page and often fall short in achieving comprehensive coverage. 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, GPT-3 have revolutionized the field of natural language processing (NLP) and artificial intelligence (AI). These models, initially pre-trained on extensive corpora, have exhibited remarkable performance across a wide range of NLP tasks including question answering, machine translation, and text generation. 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 ChatGPT[1] and LLaMA[2]

1. https://openai.com/blog/chatgpt
2. https://ai.meta.com/blog/large-language-model-llama-meta-ai/

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, software testing.

Actually LLMs have been used for various coding related tasks including code generation and code recommendation [10], [11]. However, there have been concerns about the correctness and reliability of the code generated by LLMs, as some studies have shown that the code generated by LLMs may not always be correct, or may not meet the expected software requirements. By comparison, when LLMs are used for software testing tasks, such as generating test cases or validating the correctness of software behavior, the impact of this problem is relatively weaker. This is because the primary goal of software testing is to identify issues or problems in the software system, rather than to generate correct code or meet specific software requirements. At worst, the only consequence is that the corresponding defects are not discovered. Furthermore, in some cases, the seemingly incorrect outputs from LLMs may actually be beneficial for testing corner cases in software, and can help uncover defects. Taken in this sense, we think the LLMs are a natural match with software testing.

This article presents a comprehensive review of the utilization of LLMs in software testing. We collect 52 relevant papers and conduct a thorough analysis from both software testing and LLMs perspectives, as roughly summarized in Figure 1.

From the viewpoints 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 test input generation), program debug and repair, while we do not find the practices for applying LLMs in the tasks of early testing life-cycle (such as test requirements, test plan, etc).

From the viewpoints of LLMs, our analysis includes the commonly used LLMs in these studies, the types of prompt engineering, 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 fine-tuning schema, while the others employ the prompt engineering to communicate with LLMs to steer its behavior for desired outcomes. For prompt engineering, the zero-shot learning and few-shot learning are most commonly used, while other advances ones 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 with the LLMs to help generate more diversified tests.

Furthermore, we summarize the key challenges and potential opportunities in this direction. These encompass leveraging LLMs for a wider array of software testing tasks and stages, extending their applications to a broader range of testing types and software categories, providing more comprehensive benchmark datasets along with rigorous experimental validations, incorporating advanced prompt engineering techniques, and integrating LLMs with other testing techniques, etc.

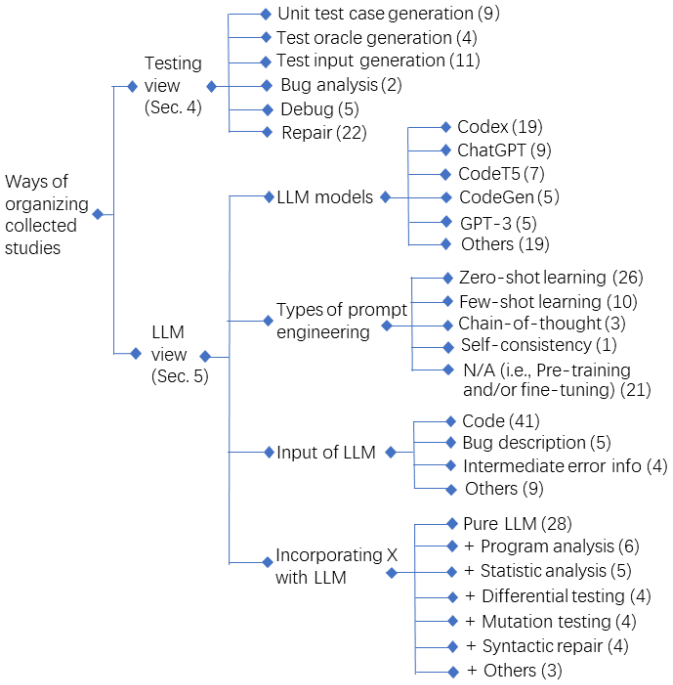

Fig. 1: Structure of the contents in this paper (the numbers in bracket indicates the number of involved papers, and a paper might involve zero or multiple items)

This paper makes the following contributions:

- We thoroughly analyze 52 relevant studies that used LLMs for software testing, regarding publication trends, distribution of publication venues, etc.
- We conduct a comprehensive analysis 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 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 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.

2 BACKGROUND

2.1 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 [12]–[15]. 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 [13], the authors focuses 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 [16], 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 is 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 [13]), but also include other models that use the encoder-decoder and decoder-only network architecture (as mentioned in [16]), 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.

2.2 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 Figure 4):

- Requirements 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 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.

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, and etc, which explores various aspects of the software under test [17].

3 Paper Selection and Review Schema

3.1 Survey Scope

The scope of our paper is software testing with LLMs. We apply the following inclusion criteria when collecting papers. If a paper satisfies any of the following criteria, we will include it.

- 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 software testing lifecyle as demonstrated in Section 2.2.
- The paper presents an empirical or experimental study about utilizing LLMs to software testing practice.
- The paper involves specific testing techniques (e.g., fuzz testing) employing LLMs.

In addition, 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 2.1).
- The paper focuses on testing the performance of LLMs, such as fairness, stability, security, etc. [18]–[20]
- The paper focuses on evaluating the performance of LLM-enabled tool, e.g., evaluating the code quality of code generation tool Copilot [21]–[23].

3.2 Paper Collection Methodology

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 titles contained keywords related to software testing tasks and testing techniques in the first three databases. In the case of DBLP, we use additional keywords related to LLMs to filter out irrelevant studies, as relying solely on testing-related keywords would result in a large number of candidate studies. We use two sets of keywords for DBLP because a significant portion of the studies in this database can be found in the first three databases, thus this fourth database serves as a supplementary source of paper collection.

- Keywords related with software testing: test |bug |issue |defect |terms of testing tasks (e.g., debug, repair) |terms of testing techniques (e.g., fuzz, mutation, metamorphic).
- Keywords related with LLMs: LLM |language model |name of the LLM (e.g., ChatGPT). We use the LLMs in [13] and [16] except those in our exclusion criteria.

To compensate for the potential omissions that may result from automated searches, we also conduct manual searches. Specifically, we manually review the titles and abstracts of papers presented at major conferences in the software engineering field to include additional possible papers.

We conduct both the keywords search and the manual search about the papers from Jan. 2019 to Apr. 2023. We further conduct the second round of paper search to include the papers from Apr. 2023 to Jun. 2023.

To further determine which papers are relevant to this survey, we conduct a three-stage paper filtering. First, we automatically filter the papers whose abstract include words "model". Second, we automatically filter the paper whose content contains the name of the LLMs, using the LLMs in [13], [16] except those in our exclusion criteria. Third, we conduct manual inspection to check whether its satisfies our inclusion criteria. The first filtering stage is mainly to eliminate the papers which donot involve the neural models, and the second filtering stage is mainly to eliminate papers that leverage machine learning, deep learning, etc. In the final filtering stage, 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 decision will be handed over to a third author to make the final decision.

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 total score of less than 8 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 about 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?

3.3 Collection Results

We retrieve a total of 14,623 papers from four databases by keyword searching. After the first filtering stage, we obtain 8,748 papers that may use the neural model. After the second filtering stage, we obtain 922 papers that may use the LLM. After the final filtering and quality assessment stage, we collect a total of 52 papers involving the software testing with LLMs. Table 1 shows the details of the collected papers. Note that, there are two papers which are the extension of previous publications from the same authors. In the

TABLE 1: Details of the collected papers

ID	Topic	Paper title	Year	Reference
1	Unit test generation	Unit Test Case Generation with Transformers and Focal Context	2021	[24]
2	Unit test generation	CodeT: Code Generation with Generated Tests	2022	[25]
3	Unit test generation	Interactive Code Generation via Test Driven User Intent Formalization	2022	[26]
4	Unit test generation	A3Test: Assertion Augmented Automated Test Case Generation	2023	[27]
5	Unit test generation	Adaptive Test Generation Using a Large Language Model	2023	[28]
6	Unit test generation	ChatUniTest: a ChatGPT based automated unit test generation tool	2023	[29]
7	Unit test generation	CodaMOSA: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models	2023	[30]
8	Unit test generation	Exploring the Effectiveness of Large Language Models in Generating Unit Tests	2023	[31]
9	Unit test generation	No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation	2023	[32]
10	Oracle generation	Generating accurate assert statements for unit test cases using pretrained transformers	2020	[33]
11	Oracle generation	Learning Deep Semantics for Test Completion	2023	[34]
12	Oracle generation; Repair	Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning	2023	[35]
13	Oracle generation; Repair	Using Transfer Learning for Code-Related Tasks	2023	[36] [37]
14	Test input generation	Automated Conformance Testing for JavaScript Engines via Deep Compiler Fuzzing	2021	[38]
15	Test input generation	Large Language Models are Pretty Good Zero Shot Video Game Bug Detectors	2022	[39]
16	Test input generation	SLGPT: Using Transfer Learning to Directly Generate Simulink Model Files and Find Bugs in the Simulink Toolchain	2022	[40]
17	Test input generation	Automating GUI-based Software Testing with GPT-3	2023	[41]
18	Test input generation	Chatting with GPT 3 for Zero Shot Human Like Mobile Automated GUI Testing	2023	[42]
19	Test input generation	Efficient Mutation Testing via Pre-Trained Language Models	2023	[43]
20	Test input generation	Fill in the Blank: Context aware Automated Text Input Generation for Mobile GUI Testing	2023	[44]
21	Test input generation	Large Language Models are Edge Case Fuzzers: Testing Deep Learning Libraries via FuzzGPT	2023	[45]
22	Test input generation	Large Language Models are Zero Shot Fuzzers: Fuzzing Deep Learning Libraries via Large Language Models	2023	[46]
23	Test input generation	TARGET: Traffic Rule based Test Generation for Autonomous Driving Systems	2023	[47]
24	Test input generation	Variable Discovery with Large Language Models for Metamorphic Testing of Scientific Software	2023	[48] [49]
25	Bug analysis	iTiger: An Automatic Issue Title Generation Tool	2022	[50]
26	Bug analysis	Explaining Software Bugs Leveraging Code Structures in Neural Machine Translation	2023	[51]
27	Debug	Detect-Localize-Repair: A Unified Framework for Learning to Debug with CodeT5	2022	[52]
28	Debug	Large Language Models are Few shot Testers: Exploring LLM based General Bug Reproduction	2022	[53]
29	Debug	Explainable Automated Debugging via Large Language Model-driven Scientific Debugging	2023	[54]
30	Debug	Finding Failure Inducing Test Cases with ChatGPT	2023	[55]
31	Debug; Repair	A Study on Prompt Design, Advantages and Limitations of ChatGPT for Deep Learning Program Repair	2023	[56]
32	Repair	Examining Zero-Shot Vulnerability Repair with Large Language Models	2021	[43]
33	Repair	Automated Repair of Programs from Large Language Models	2022	[57]
34	Repair	Can OpenAI's Codex Fix Bugs?: An evaluation on QuixBugs	2022	[58]
35	Repair	Fix Bugs with Transformer through a Neural Symbolic Edit Grammar	2022	[59]
36	Repair	Practical Program Repair in the Era of Large Pre-trained Language Models	2022	[60]
37	Repair	Repairing Bugs in Python Assignments Using Large Language Models	2022	[61]
38	Repair	Towards JavaScript Program Repair with Generative Pre-trained Transformer (GPT-2)	2022	[62]
39	Repair	An Analysis of the Automatic Bug Fixing Performance of ChatGPT	2023	[63]
40	Repair	CIRCLE: Continual Repair Across Programming Languages	2023	[64]
41	Repair	Domain Knowledge Matters: Improving Prompts with Fix Templates for Repairing Python Type Errors	2023	[65]
42	Repair	Fixing Hardware Security Bugs with Large Language Models	2023	[66]
43	Repair	Framing Program Repair as Code Completion	2023	[67]
44	Repair	How Effective are Neural Networks for Fixing Security Vulnerabilities	2023	[68]
45	Repair	Impact of Code Language Models on Automated Program Repair	2023	[69]
46	Repair	InferFix: End to End Program Repair with LLMs	2023	[70]
47	Repair	Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT	2023	[71]
48	Repair	Neural Program Repair with Program Dependence Analysis and Effective Filter Mechanism	2023	[72]
49	Repair	Towards Generating Functionally Correct Code Edits from Natural Language Issue Descriptions	2023	[73]
50	Repair	VulRepair: A T5-based Automated Software Vulnerability Repair	2023	[74]

Note that: we put [36] and [37], [48] and [49] together considering they are respectively the original paper and its extension by the same authors.

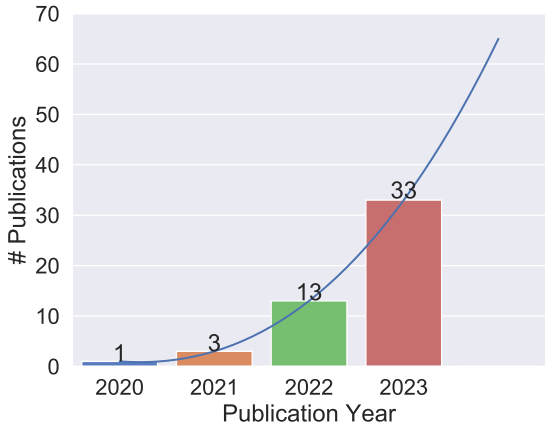

Fig. 2: Trend in the number of papers with year

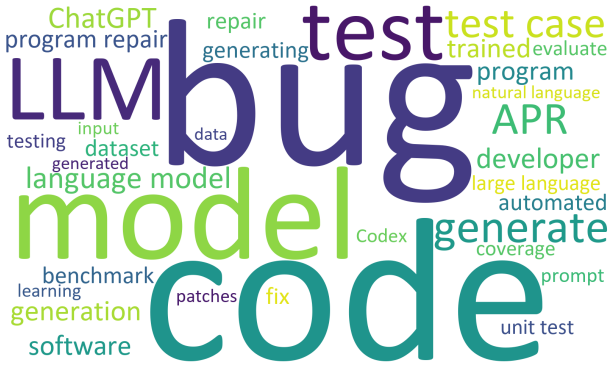

Fig. 3: Topics discussed in the collected papers

following analysis, we only include the extended paper. In other words, the following analysis is based on 50 collected studies.

3.4 General Overview of Collected Paper

Among the papers, 38% papers are published in software engineering venues, among which 7 papers are from ICSE, 2 papers are from FSE, and 3 papers are from ISSTA. 4% papers are published in artificial intelligence venues such as EMNLP and ICLR, and 6% papers are published in program analysis or security venues like PLDI and S&P. Besides, 52% of the papers have not yet published via peer-reviewed venues (i.e., 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 assessment process

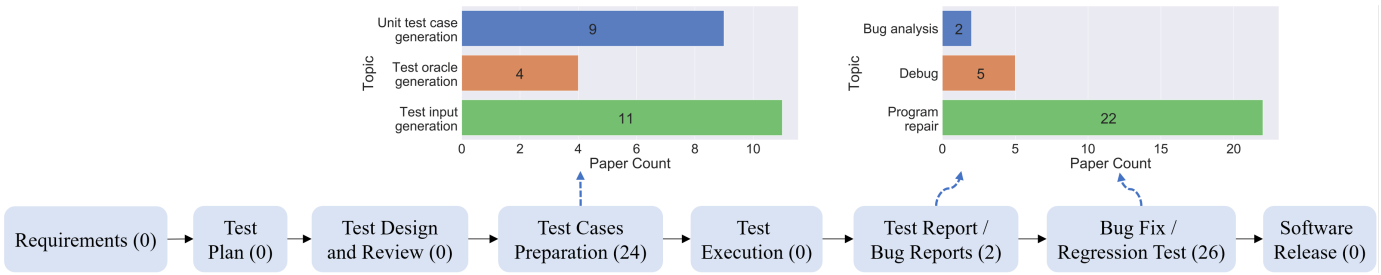

Fig. 4: Distribution of testing tasks with LLMs (aligned with software testing life cycle [75]–[77], the number in bracket indicates the number of collected studies per task, and one paper might involve multiple tasks)

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

Figure 2 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 3 papers, respectively. In 2022, there were 13 papers, and in the first half of 2023, there were already 33 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.

To provide a general overview of these papers, Figure 3 shows the word cloud created by the abstracts of the collected papers, from which we can generally observe the involved topics in these collected papers.

4 Analysis from Software Testing

This section conducts the analysis based on the viewpoints of software testing, and will organize the collected studies in terms of testing tasks. Figure 4 lists the distribution of each involved testing tasks, aligned with the software testing life cycle. We first provide some general overview of the distribution, following by the 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 mid to late stages of the software testing lifecycle. In the test case preparation phase, LLMs have been utilized for tasks such as generating unit test cases, test oracle generation, 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, bug fix and regression test 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.

4.1 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 as 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 [1], [2], constraint-based [3] or random-based strategies [4] 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.

Early practices for fine-tuning LLMs. The early practices mainly focus on how to pre-train or fine-tune LLMs with domain-specific data, in order to enhance their effectiveness on the unit test generation task. For example, [27] first pre-trained the LLM with the focal method and asserted statements to enable the LLM having 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. [24] 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.

Later studies focus on designing effective prompts. Subsequently, LLMs possess enhanced abilities, allowing them to excel at targeted tasks without the pre-training or fine-tuning. Therefore the later studies typically focus on how to design the prompt, to make the LLM better at understanding the context and nuances of this task. [29] 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 incorporate these context into a prompt to query the LLM. [32] first performed an empirical study to evaluate ChatGPT's capability of unit test generation with both a quantitative analysis and a user study in

TABLE 2: Performance of unit test case generation

Dataset	Correctness	Coverage	LLM	Paper
5 Java projects from Defects4J	16.21%	5%-13% (line coverage)	BART	[24]
10 Jave projects	40%	89% (line coverage), 90% (branch coverage)	ChatGPT	[29]
CodeSearchNet	41%	N/A	ChatGPT	[32]
HumanEval	78%	87% (line coverage), 92% (branch coverage)	Codex	[31]
SF110	2%	2% (line coverage), 1% (brach coverage)	Codex	[31]

Note that, [31] experiments with Codex, CodeGen, and ChatGPT, and the best performance was achieved by Codex.

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 proposed 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.

Incorporating LLM with search based software testing for unit test generation. The aforementioned studies utilize LLMs for the whole unit test case generation task, while [30] focus on a different direction, i.e., first letting the traditional search-based software testing techniques (e.g., Pynguin [78]) 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 redirects its search to more useful areas of the search space.

Performance of unit test case generation. Since the aforementioned studies of unit test case generation are based different datasets, one can hardly derive a fair comparison and we present the details in Table 2 to let the readers obtain a general view. We can see that in SF110 benchmark, all three evaluated LLMs have quite low performance, i.e., 2% coverage [31]. This SF110 dataset, which is an Evosuite (a search-based unit test case generation technique) benchmark consisting of 111 open-source Java projects retrieved from SourceForge, contains 23,886 classes, over 800,000 bytecode-level branches, and 6.6 million lines of code. The authors did not present the detailed reasons for the low performance, and can be further explored in future.

4.2 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. Most the collected studies in this category target at 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 researches specifically focus on the generation of effective test assertion. Actually, before using LLMs, researchers have proposed RNN-based approach which aims at learning from thousands of unit test methods to generate meaningful assert statements [79], 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.

[36], [37] 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%. [33] 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 exact match rate. Besides the syntax-level data as previous studies, [34] 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 with 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 are able to perform well on specific tasks without these specialized pre-training or fine-tuning datasets. Subsequently, [35] utilized the 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.

4.3 Test Input Generation

This category covers the studies related to creating test input for enabling the automation of test execution, and for our collected studies, this category is mainly for system testing. The generation of system-level test inputs for software testing varies for different types of software. For 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), which is key to test the application's functionality and user interface; while

for Deep Learning (DL) libraries, the test input is a program which covers diversified DL APIs.

Moreover, different from the unit test case generation, for these system-level testing tasks, the test assertions usually cannot be obtained directly, so this type of work usually requires cooperation with a specialized testing technology to complete the entire testing task (e.g., differential testing); or only considers those bugs with observable oracles (e.g., crash). In other words, these studies typically only involve the creation of test inputs without a corresponding test oracle; thereby they are usually paired with automated testing techniques to help derive a test oracle which can be used to evaluate the correctness of the software under test. We will present more details about these collected studies.

Test input generation for DL libraries. The input for testing DL libraries is DL programs, and the difficulty for 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 [80], [81] or model-level fuzzing [82], [83] 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, [46] uses 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 program (i.e., code snippets that use the target DL APIs). Then it replaces part of the seed program with masked tokens using different mutation oprators, and leverages the ability of infilling LLM (InCoder) to perform code infilling to generate new code that replaces the masked tokens. And their follow-up study [45] 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 shows improved bug detection performance.

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 [6], [84] are far from generating the meaningful text input. [44] employs the LLM to intelligently generate the semantic input text according to the GUI context. In detail, it automatically extracts the component information related to the EditText and generates the prompt, and then inputs the prompt 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, 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 [5], [6], model-based methods [7], [8], and learning-based methods [9] are unable to understand the semantic information of the GUI page thus could not conduct the trace planning effectively. [42] 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. Within it, the approach extracts the static context of the GUI page and the dynamic context of the iterative testing process, designs prompts for inputting these information to LLM which enables the LLM better understand the GUI page as well as the whole testing process.

Test input generation for others. Findings bugs in a commercial cyber-physical system (CPS) development tool such as Simulink is even challenging. Given the complexity of Simulink language, generating valid Simulink model files for testing is an ambitious task for traditional machine learning or deep learning techniques. [40] 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.

In addition, there are studies employing the LLMs for test input generation to serve as the testing of JavaScript engine, autonomous driving systems, and video game. For example, [38] utilizes the LLM for generating the JavaScript programs, and utilizes the well-structured ECMAScript specifications to automatically generate test data along with the test programs, and then applies differential testing to expose bugs. [47] uses LLM to extract key information related to the test scenario from a traffic rule, and represents the extracted information in a test scenario schema, then synthesizes the corresponding scenario scripts to construct the test scenario.

4.4 Bug Analysis

This category involves analyzing identified software bugs to facilitate the understanding of the bug. [51] 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. [50] targets at automatically generating 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.

4.5 Debug

This category focuses on identifying and resolving software errors to ensure that the code works as expected, which may involve analyzing code, tracing execution, collecting error information, and outputting information to find and fix issues. For example, [52] 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. [56] conducts a study of the deep learning program debugging ability, including fault

detection, fault localization and program repair of LLM. [54] 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.

There are also studies focusing on a sub-phase of the debugging process. For example, [53] proposes a framework to harness the LLM to reproduce bugs, and suggest bug reproducing test cases to the developer for facilitating debug. [55] focuses on a similar aspect about 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.

4.6 Program Repair

This category denotes the task to fix 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 to perform code-related tasks such as fixing bugs.

There have been template-based [85], heuristic-based [86], and constraint-based [87], [88] 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 adapt deep learning models to take a buggy software program as input and generate a patched program. They would build a neural network model from a training set, and learns the relations between the buggy code and 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 amount 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.

Patch single-line bugs. In the early era of program repair, the focus is mainly on addressing defects related to single-line code errors, which is relatively uncomplicated and did not require the repair of complex program logic. [62] proposes to fine-tune the LLM with JavaScript code snippets to serve as the purpose for the JavaScript program repair. [72] employs the 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 focus on the buggy code. 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).

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 better understand the code semantics. [74] fine-tunes the LLM by employing BPE tokenization to handle Out-Of-Vocabulary (OOV) issues which makes the approach can generate new tokens that never appear in a training function but are newly introduced in the repair. The aforementioned studies (including the ones in patching single-line bugs) would predict the fixed programs directly, and [59] utilizes a different setup which 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 the 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 [56], [58], [60], [63], [68], [69], which will be explored in depth later, while other studies focus on proposing new techniques. [67] takes 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. [71] proposes 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. [61] proposes 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.

Repair from natural language issue descriptions. The aforementioned studies all consider the buggy code as the input, and let the LLM conduct the automatic program repair. [73] focuses on program repair from natural language issue descriptions, i.e., generating the patch with the bug and fix related information described in issue reports in repositories.

Repair with static code analyzer. Most of the program repair studies would suppose the bug has been detected, while [70] proposes a program repair framework paired with a static analyzer to first detect the bugs, and then fix it.

TABLE 3: Performance of program repair

Dataset	% Correct patches	LLM	Paper
Defects4J v1.2, Defects4J v2.0, QuixBugs, HumanEval-Java	22/40 Jave bugs (QuixBugs dataset, with InCoder-6B, correct code infilling setting)	PLBART, CodeT5, CodeGen, InCoder (each with variant parameters, 10 LLMs in total)	[69]
QuixBugs	23/40 Python bugs, 14/40 Java bugs (complete function generation setting)	Codex-12B	[58]
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)	[60]
QuixBugs	31/40 Python bugs (completion function generation setting)	ChatGPT-175B	[63]
DL programs from StackOverflow	16/72 Python bugs (complete function generation setting)	ChatGPT-175B	[56]

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

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.

Empirical study about program repair. There are several studies related with the empirical or experimental evaluation of the various LLMs on program repair, and we summarize the performance in Table 3. [60], [69] conduct relatively comprehensive experimental evaluations with various LLMs and on different automated program repair benchmarks, while [56], [58], [63] focus on a specific LLM and on one dataset, i.e., QuixBugs. There are two commonly-used repair settings one can use 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 setting which is explicitly marked in the table. The commonly-used datasets are QuixBugs, Defects4J, etc. These datasets only involve the fundamental functionalities such as sorting algorithm, each program's average number of lines ranging from 13 to 22, implementing one functionality, and involving few dependencies. [56] 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.

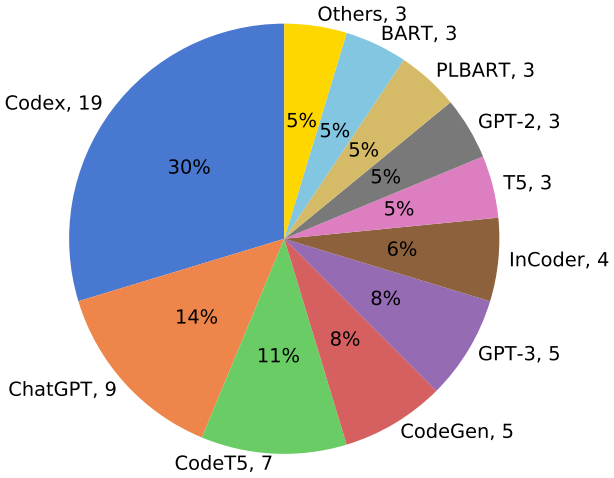

Fig. 5: LLMs used in the collected papers

5 Analysis from LLM

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

5.1 LLM Models

As shown in Figure 5, the most commonly utilized LLM is Codex (a LLM based on GPT-3) which is trained on a massive code corpus containing examples from many programming languages such as JavaScript, Python, C/C++, and Java. Codex is released on Sep. 2021 by OpenAI and powers GitHub Copilot– an AI pair programmer that generates whole code snippets, given a natural language description as a prompt. Since a large portion of our collected studies involve the source code (e.g., repair, unit test case generation), it is not surprising that researchers choose Codex as the LLM in assisting them accomplishing the coding related tasks.

ChatGPT, which is released on Nov. 2022, is the second most commonly used LLM in our collected studies. It is trained on a large corpus of natural language text data, and primarily designed for natural language processing and conversation. Since Codex is released more than one year earlier than ChatGPT, it is most commonly utilized by the studies earlier, and later studies tend to employ ChatGPT to investigate its feasibility in various software testing tasks and tackle the challenges of generating human-like text responses. In software testing domain, we have not observed there are studies employing the newly developed GPT-4, which is demonstrated to be more powerful in various tasks. This maybe because it is lunched on Mar. 2023, which is just a few months earlier before we conduct the paper collection. And we believe there would be more attempts about employing GPT-4 for software testing tasks, especially these related with multimodal such as UI screenshots.

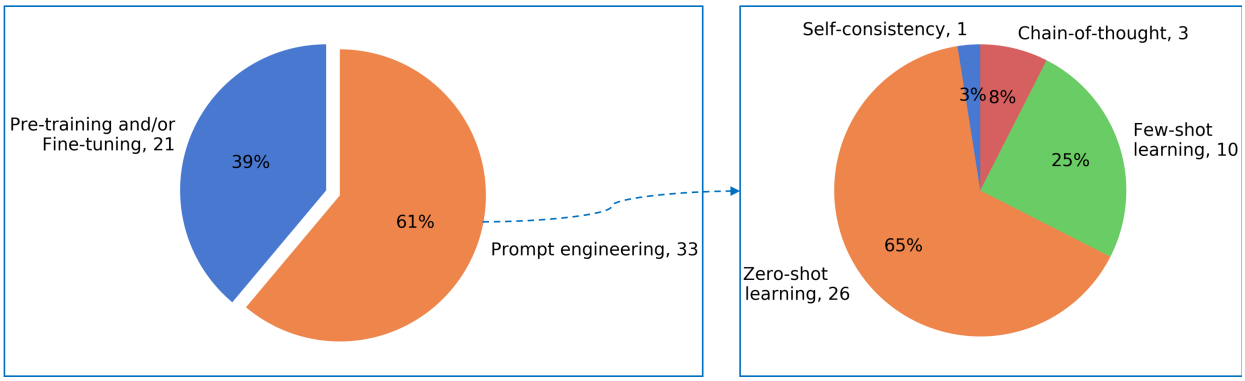

Fig. 6: Distribution about how LLM is used (Note that, a study can involve multiple types of prompt engineering)

The third-ranked LLM is CodeT5, which is an open sourced LLM developed by salesforce[3]. Thanks to its open source, researchers can easily conduct the pre-training and fine-tuning with domain specific data to achieve better performance. Similarly CodeGen is also open sourced and ranked relatively higher. Besides, for CodeT5 and CodeGen, there are more than half of the related studies involve the empirical evaluations (which employ multiple LLMs), e.g., program repair [68], [69], unit test case generation [31].

There are also five studies utilizing GPT-3, which is also widely popular in the general domain, for software testing. As GPT-3 can provide good performance in general natural language understanding, it is mainly used for some general tasks, such as the input generation for automatic mobile application testing [42], [44], and the traffic rule parsing in autonomous driving tests [47].

5.2 Types of Prompt Engineering

As shown in Figure 6, among our collected studies, 21 studies utilize the LLMs through pre-training or fine-tuning schema, while 33 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 33 studies with prompt engineering, 27 studies involve zero-shot learning, and 9 studies involve the few-shot learning (a study may involve multiple types). There are also respectively 3 and 1 studies involving chain-of-though and self-consistency.

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 5.1), 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 query 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, [29] 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, [31] only provides 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 the tasks that is not so straightforward or intuitive to the LLM. For example, when conducting the automatic test generation from general bug reports, [53] provides 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 test assertion generation, [35] provides demonstrations of the focal method, the test method containing an <AssertPlaceholder>, and the expected assertion, which enables the LLMs 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 eventually lead to the final answer. For example, for program repair from the natural language issue descriptions [73], given the buggy code and issue report, the authors first ask the LLM to localize the bug, then they ask to explain why the localized lines are buggy, finally they ask to fix the bug. Another example is for generating unusual programs for fuzzing deep learning libraries, [45] first generates a possible "bug" (bug description) before generating the actual "bug-triggering" code snippet that invokes the target API. The predicted bug description provides additional hint to the LLM, indicating that the generated code should try to cover specific potential buggy behavior.

3. https://blog.salesforceairesearch.com/codet5/

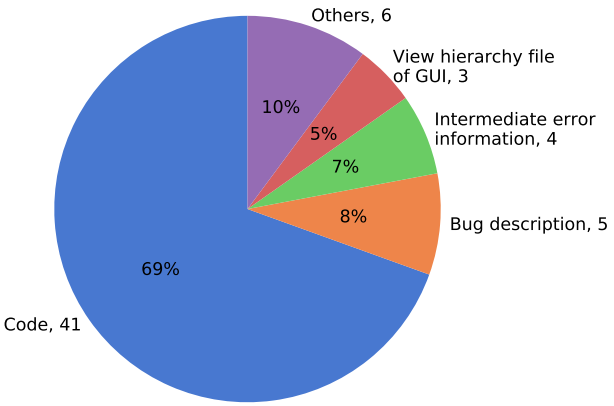

Fig. 7: Input of LLM

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 debug. [54] employs a hypothesize-observe-conclude loop, which first generates a hypothesis about what the bug is and construct 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.

We also want to mention that there are eight studies 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, [71] interleaves patch generation with test validation feedback to prompt future generation in an iterative manner. 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 prompt which can help the model understand the failure reason and provide guidance towards generating the correct fix. Another example is for mobile GUI testing, [42] 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.

5.3 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 Figure 7.

The most commonly utilized input is the **source code**, since a large portion of collected studies relate with 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 [24], [32]. For program repair, there can be different setups and involves different inputs, including (i) inputting a buggy function with the goal to output the patched function, (ii) inputting the buggy location with the goal to generate the correct replacement code (can be a single line change) given the prefix and suffix of the buggy function [60]. Besides, there can be variations for the buggy location input, i.e., (i) not contain the buggy lines (but the bug location is still known), (ii) give the buggy lines as lines of comments.

There are also five studies taking the **bug description** as input for the LLM. For example, [53] takes the bug description as input when querying LLM and let the LLM generate the bug reproducing test cases, and [73] inputs the natural language descriptions of bugs to the LLM, and generate the correct code fixes.

There are four studies which would provide the **intermediate error information**, e.g., test failure information, to the LLM, and would conduct the iterative prompt (as described in Section 5.2) to enrich the context provided to the LLM. These 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 researches would provide the LLM with the **view hierarchy file** which represents the details of the GUI page. Nevertheless, with the emerging 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.

5.4 Incorporating *X* with LLM

There are divided opinions on whether LM has reached an all-powerful status that requires no other techniques. As shown in Figure 8, among our collected studies, 28 of them utilize LLMs to address the entire testing task, while 22 studies incorporate additional techniques. These techniques include mutation testing, differential testing, syntactic checking, program analysis, statistical analysis, as well as natural language processing and formal methods.

The reason why researchers still choose to combine LLMs with other techniques might be because, despite exhibiting enormous potential in various tasks, LLM 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.

LLM + program analysis. When utilizing LLMs to accomplish tasks such as generating unit test cases and

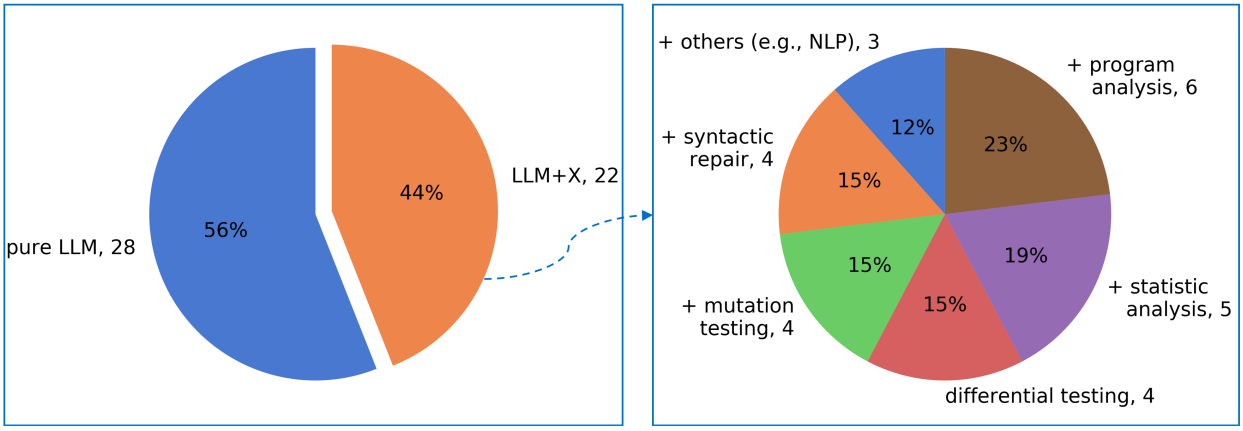

Fig. 8: Distribution about LLM + X (Note that, a study can involve multiple types LLM + X schema)

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) [51], 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 [61], or extracts additional code context from other code files [32], to enable the models to focus on the most task-relevant information in the codebase and lead to more accurate predictions.

LLM + statistic 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 [26], [34], [53], [60], [72] to efficiently filter through LLM's outputs and ultimately obtain more accurate results.

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 response for generating the 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, [38] 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 as JavaScriptCore, ChakraCore, SpiderMonkey, QuickJS, etc. [45], [46] design a similar approach to let the LLM generate the test input and then conduct the differential testing for fuzzing DL libraries. [55] 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 performs the differential testing with the program under test and the generated programs to find the failure-inducing test cases.

LLM + mutation testing. It is mainly targeting at generating more diversified test inputs. For example, [46] first uses 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.

LLM + syntactic repair. Although LLMs have shown remarkable performance in various natural language processing tasks, the generated code from these models can sometimes 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 [27] for unit test case generation, the authors 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). Another example is [29] which also validates the generated unit test case and employs rule-based repair to fix syntactic and simple compile errors.

Besides, interpreting LLMs' predictions and outputs remain challenging, thus combining various techniques like the natural language processing can help to post-process LLM's outcome and filtering unsatisfied results.

6 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. These include utilizing LLMs for a more diverse set of software testing tasks and phases, expanding their usage to a wider variety of testing types and software, supplying more comprehensive benchmark datasets together with thorough experimental evaluations, integrating advanced prompt engineering, and combining LLMs with relevant existing techniques, etc.

6.1 Extending to More Tasks and More Phases

Utilizing LLMs for tasks in early stage of testing. As shown in Figure 4, LLMs have not been used in the early stage of testing, e.g., test requirements, 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.

Nevertheless, challenges and opportunities always coexist. At the early stages, a crucial task is to automatically generate test requirements based on software requirement specifications and/or user manuals. This task is akin to machine translation, where LLMs have already demonstrated impressive performance [89]. However, unlike machine translation, which involves relatively clear semantic mapping, generating test requirements requires significant domain knowledge and may involve integrating information about the operation of the software being tested. It is also essential to consider the coverage of the generated test requirements for the software under test.

Exploring LLMs for tasks that popularly studied with machine/deep learning techniques. Despite its success in task related to test case prepartion and program repair, there are testing tasks which have been popularly studied with traditional techniques or machine/deep learning techniques, e.g., test prioritization [90], regression testing [91], bug triage [92], yet have not been touched by the LLM. For test prioritization and regression testing, the common practice with machine/deep learning techniques would analyze the code and its dependencies and prioritize the execution order of test cases based on their impact or likelihood of detecting faults. With the LLM, these can be conducted more accurately. In addition, LLMs can assist in analyzing and triaging bug reports by leveraging their comprehension and inference abilities. They can identify duplicates, classify bugs based on severity or priority, extract relevant information, and suggest potential resolutions or workarounds.

Exploring LLMs for integration testing and acceptance testing. We further analyze the distribution of testing phases for the collected studies. As shown in Figure 9, 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, the integration testing might be merged with unit testing, which can be a possible reason why LLM is rarely utilized in 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.

For tackle this, a potential reference can be found in Section 4.1, where [29] designs 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 demonstrated in Section 4.3, previous work has demonstrated the LLM's capability in generating diversified test input for system testing, in conjunction with mutation testing techniques [38], [46]. And these can provide the insights about generating the diversified interface data for integration testing.

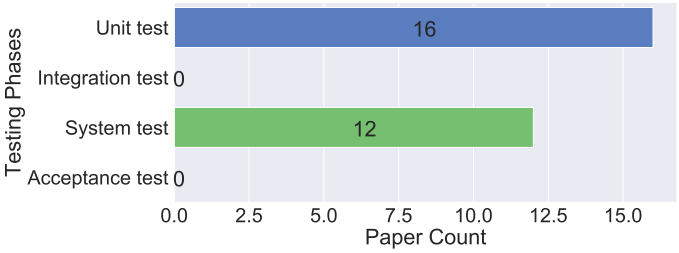

Fig. 9: Distribution of testing phases (note that we omit the studies which donot explicitly specify the testing phases, e.g., program repair)

For acceptance testing, it is usually conducted by business analysts or end-users to validate the system's functionality and usability, which require 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 [93], 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.

6.2 Serving Other Types of Testing and Software

Employing LLMs for non-functional testing. In our collected studies, LLMs are primarily used for functional testing, with only few applications in security 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 are able to convert functional testing problems into code generation or natural language generation problems, which LLMs are particularly adept at solving. For our collected studies related with security testing, all of them involve the vulnerability detection and repair in the source code, rather than the security of the whole software system. This also attributes to the capability of LLM in understanding and processing the source code.

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) which can generate a workload that simulates real-world usage scenarios and achieve relative satisfactory performance.

The potential opportunities might let the LLM integrates the performance testing tools and acts like the LangChain [94], for better simulating 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.

Employing LLMs for other types of software. We also analyze what types of software have been explored in the collected studies. Results show that the following types of software are explored, as shown in Figure 10. Note that, since a large portion of studies are focused on the unit testing or program repair, they are conducted on publicly available datasets and do not involve specific software types.

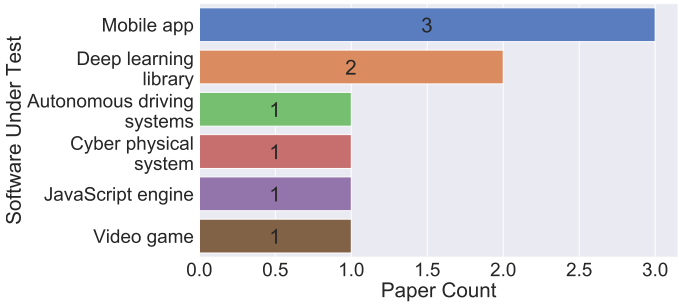

Fig. 10: Distribution of software under test

From the analysis in Section 4.3, 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, including testing DL libraries, mobile apps, and CPS. Its versatility would makes it useful for testing the software in other domains.

From one point of view, some propose techniques can be applicable to other types of software. For example, in the paper proposed for testing deep learning libraries [45], since it proposes techniques for generating the 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., compilers, interpreters, database systems, SMT solvers, and other popular libraries. And we also notice there are exploration about using LLM for database testing in a technical blog [95], which can generate an equivalent SQL based on the Ternary Logic Partitioning (TLP) principle to help derive the test oracle.

From other point of view, other types of software can also benefit from the capabilities of LLMs. For example, for Web applications, the LLMs can generate test inputs such as HTTP requests, form inputs, and data payloads to test the functionality, security, and performance of web applications. For network protocols, the LLMs can generate network traffic and protocol messages to test the robustness of network protocols, which can include testing communication between different network components, handling of error and edge cases, and adherence to protocol specifications. For IoT devices, the LLMs can generate test inputs and commands to test device communication, sensor data processing, firmware updates, and integration with other systems.

6.3 More Solid Benchmarks and Rigorous Evaluations

We find the following limitations which hinders the rigorous evaluations and thorough comparison of the proposed techniques. These limitations highlight the need for more solid benchmarks and rigorous evaluations.

The first is the data leakage issue, i.e., the LLMs may have seen these benchmarks in their pre-training data. [69] checks the CodeSearchNet and BigQuery, which are the datasources of common LLMs, 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 [96] 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 researches. It also suggests that we need more specialized datasets that are not seen by LLMs to serve as benchmarks. One way is to collecting it from specialized sources, e.g., user-generated content from niche online communities.

The second is the limited number of benchmarks. For unit test case generation, there is even no widely recognized benchmarks, and different studies would utilize different datasets for performance evaluation, as demonstrated in Table 2. For program repair, there are only two well-known and commonly-used benchmarks, i.e., Defect4J and QuixBugs, as demonstrated in Table 3. Furthermore, these datasets are not specially designed for testing the LLMs. For example, as reported in [60], 39 out of 40 Python bugs in QuixBugs dataset can be fixed by Codex, yet in real-world practice, the successful fix rate can be nowhere near as high. This motivates to build more specialized and diversified benchmark as mentioned in the first limitation.

The third is the performance variation across datasets. This is not only the problem for LLMs, but also for traditional machine learning and deep learning models. For example for unit test case generation, as shown in Table 2, with the same technique [31], on HumanEval dataset it can generate 78% correct unit test cases, and the line coverage is 87%, yet on SF110 dataset, both the correctness of generated tests and the line coverage is merely 2%. The performance variation among different datasets indicates the fragile of current unit test case generation techniques. This performance variation is also frequently observed in traditional machine learning tasks, e.g., in defect prediction [97], [98]. Typically solution evolves more reliable benchmark and extension comparison as mentioned in the first and second limitations.

The fourth is the performance inconsistency under the same setting. For example for program repair, even based on the same dataset, with the same LLM, and use the same repair setup, the results report in different studies might differ. For example, as shown in Table 3, on QuixBugs Python dataset, with Codex (12B), for the completion function generation setting, the correctly repaired bugs are respectively

23 [58] and 37 [60]. We assume there might be other imperceptible differences in these two studies. For instance, they might employ different prompt expression when querying the LLM, which results in performance variation, since existing studies have also reveal that a slight modification in the prompts can result in dramatic performance changes. This phenomenon is not frequently observed in traditional machine learning tasks, but it is worth noting in the context of LLMs due to the unique nature of prompts. More detailed description of the proposed approach and experimental setup is encouraged in future work to facilitate the follow-up comparison.

6.4 Boosting LLM Performance

Exploring advanced prompt engineering. There are a total of 11 commonly used prompt engineering techniques as list in a pupluar prompt engineering guide [99], as shown in Figure 11. Currently, in our collected studies, only the first four techniques are being utilized. The more advanced techniques have not been employed yet, and can be explored in the future for prompt design.

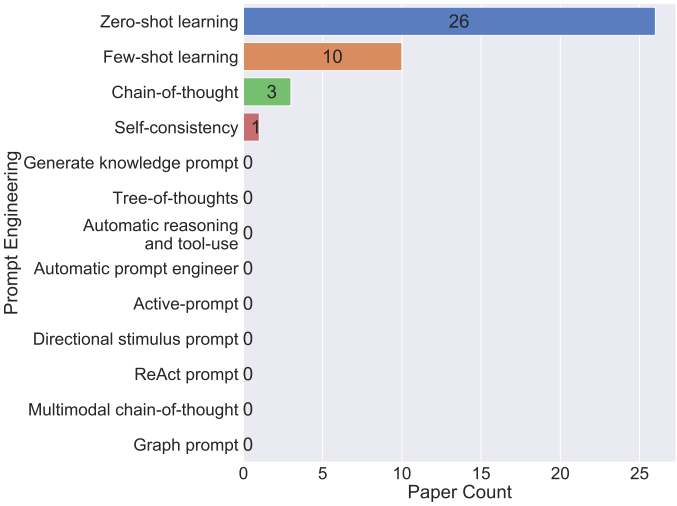

Fig. 11: List of advanced prompt engineering practices and those utilized in the collected papers

For instance, graph prompting involves the representation of information using graphs or visual structures to facilitate understanding and problem-solving. Graph prompting can be a natural 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 these 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.

Multimodal chain of thought prompting involves using diverse sensory and cognitive cues to stimulate thinking and creativity in LLMs. 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.

LLMs with relevant existing techniques. There is currently no clear consensus on the extent to which LLMs can solve software testing problems. From the analysis in Section 5.4, 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. Since there are already many mature software testing techniques and tools, while the capabilities of LLMs are not yet outstanding, therefore it is necessary to explore other better ways to combine LLMs with traditional testing techniques and tools for better software testing.

From the collected studies, the LLMs have successfully utilized together with differential testing and mutation testing as shown in Figure 8. More explorations are encouraged in terms of combining LLMs with other testing techniques. For instance, metamorphic testing involves generating test cases based on the expected relationship between inputs and outputs, rather than on specific input-output pairs, to determine whether the software system behaves correctly when the input changes in a predictable way. LLMs can be used to recommend the potential metamorphic relations, which can help people divergent thinking. Following that, LLMs can be used to generate test cases automatically based on the metamorphic relations that have been defined, which can covers a variety of inputs. They can also be used to augment existing test suites with additional test cases generated based on metamorphic relations, thus improve the diversity of the test suite and increase the chances of detecting bugs.

Another example is model-based testing. We have mentioned that the LLMs have successfully been utilized in generating the Simulink model files and find bugs in the Simulink toolchain [40]. And LLMs might be used to learn the model of the software system by analyzing natural language descriptions of the system's behavior. Specifically, LLMs can be used to analyze the system documentation, user manuals, or other sources of information to learn about the system's behavior, so as to improve the accuracy of the model for testing. In addition, LLMs can be used to help evolve the model of the software system as the system changes or evolves, e.g., analyze the natural language descriptions of system changes or bug reports to automatically update the model or generate new test cases.

The LLMs can be utilized through the LLM in the loop manner, which invokes the LLM when it is needed. Besides, since there are many mature software testing tools, and one can let the LLMs integrate these tools and acts like the LangChain for better explore the potential of these tools.

Fine-tuning open-source LLMs with software-specific data. As we mentioned in Section 5.2, 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 which 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 instead. In such cases, using the software-specific data within the organization for model fine-tuning can further improve the performance.

Building high-quality datasets for fine-tuning is crucial, and existing research has shown that high-quality training data can significantly improve the performance of tasks such as code search [100]. However, manually constructing such datasets can be time-consuming and labor-intensive. Meanwhile, in the past few years, researchers have explored automated techniques to extract the key information (e.g., issue and solution) from Stack Overflow and Gitter chatrooms [101]. Since these platforms provide a wealth of information on programming languages, frameworks, libraries, as well as common coding patterns and practices, automatically extracting the targeted information can serve as an important source for the fine-tuning dataset.

In addition, exploring the methodology about how to better fine-tune the LLMs with software-specific data is worth considering because software-specific data differs from natural language data in that it contains more structural information, such as data flow and control flow. Previous research on code representations has considered employing the data flow which is a semantic-level structure of code that depicts the relation of "whether-value-comes-from" between variables, and demonstrates its advantages than considering the code without structure [102]. Besides, researches also utilize the abstract syntax tree and control flow [103], [104] for code representation and achieve good performance. These can provide valuable insights for fine-tuning LLMs with software-specific data.

7 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. 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] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," vol. 36, no. 2, 2010, pp. 226–247.

[2] 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. Software Eng.*, vol. 49, no. 4, pp. 2580–2596, 2023.

[3] X. Xiao, S. Li, T. Xie, and N. Tillmann, "Characteristic studies of loop problems for structural test generation via symbolic execution," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, E. Denney, T. Bultan, and A. Zeller, Eds. IEEE, 2013, pp. 246–256.

[4] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. IEEE Computer Society, 2007, pp. 75–84.

[5] A. Developers, "Ui/application exerciser monkey," 2012.

[6] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: a lightweight ui-guided test input generator for android," in *ICSE*. IEEE, 2017.

[7] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 245–256.

[8] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel testing of android apps," in *ICSE*. IEEE, 2020.

[9] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of android applications," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 153–164.

[10] J. Li, Y. Li, G. Li, X. Hu, X. Xia, and Z. Jin, "Editsum: A retrieve-and-edit framework for source code summarization," in *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 2021, pp. 155–166. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678724

[11] Y. Dong, X. Jiang, Z. Jin, and G. Li, "Self-collaboration code generation via chatgpt," *CoRR*, vol. abs/2304.07590, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2304.07590

[12] M. Shanahan, "Talking about large language models," *CoRR*, vol. abs/2212.03551, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2212.03551

[13] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, Y. Du, C. Yang, Y. Chen, Z. Chen, J. Jiang, R. Ren, Y. Li, X. Tang, Z. Liu, P. Liu, J. Nie, and J. Wen, "A survey of large language models," *CoRR*, vol. abs/2303.18223, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2303.18223

[14] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," in *NeurIPS*, 2022. [Online]. Available: http://papers.nips.cc/paper_files/paper/2022/hash/8bb0d291acd4acf06ef112099c16f326-Abstract-Conference.html

[15] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *NeurIPS*, 2022. [Online]. Available: http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html

[16] S. Pan, L. Luo, Y. Wang, C. Chen, J. Wang, and X. Wu, "Unifying large language models and knowledge graphs: A roadmap," *CoRR*, vol. abs/2306.08302, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2306.08302

[17] G. J. Myers, T. Badgett, T. M. Thomas, and C. Sandler, *The art of software testing*. Wiley Online Library, 2004, vol. 2.

[18] C. Treude and H. Hata, "She elicits requirements and he tests: Software engineering gender bias in large language models," *CoRR*, vol. abs/2303.10131, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2303.10131

[19] 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," *CoRR*, vol. abs/2302.07371, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2302.07371

[20] 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 *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*. ACM, 2022, pp. 167–178. [Online]. Available: https://doi.org/10.1145/3524842.3528440

[21] D. Erhabor, S. Udayashankar, M. Nagappan, and S. Al-Kiswany, "Measuring the runtime performance of code produced with github copilot," *CoRR*, vol. abs/2305.06439, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2305.06439

[22] R. Wang, R. Cheng, D. Ford, and T. Zimmermann, "Investigating and designing for trust in ai-powered code generation tools," *CoRR*, vol. abs/2305.11248, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2305.11248

[23] 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," *CoRR*, vol. abs/2304.10778, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2304.10778

[24] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit test case generation with transformers and focal context," *arXiv preprint arXiv:2009.05617*, 2020.

[25] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, "Codet: Code generation with generated tests," *arXiv preprint arXiv:2207.10397*, 2022.

[26] S. K. Lahiri, A. Naik, G. Sakkas, P. Choudhury, C. von Veh, M. Musuvathi, J. P. Inala, C. Wang, and J. Gao, "Interactive code generation via test-driven user-intent formalization," *arXiv preprint arXiv:2208.05950*, 2022.

[27] S. Alagarsamy, C. Tantithamthavorn, and A. Aleti, "A3test: Assertion-augmented automated test case generation," *arXiv preprint arXiv:2302.10352*, 2023.

[28] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "Adaptive test generation using a large language model," *arXiv preprint arXiv:2302.06527*, 2023.

[29] Z. Xie, Y. Chen, C. Zhi, S. Deng, and J. Yin, "Chatunitest: a chatgpt-based automated unit test generation tool," *arXiv preprint arXiv:2305.04764*, 2023.

[30] 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 *International conference on software engineering (ICSE)*, 2023.

[31] 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," *arXiv preprint arXiv:2305.00418*, 2023.

[32] Z. Yuan, Y. Lou, M. Liu, S. Ding, K. Wang, Y. Chen, and X. Peng, "No more manual tests? evaluating and improving chatgpt for unit test generation," *arXiv preprint arXiv:2305.04207*, 2023.

[33] M. Tufano, D. Drain, A. Svyatkovskiy, and N. Sundaresan, "Generating accurate assert statements for unit test cases using pretrained transformers," in *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, 2022, pp. 54–64.

[34] P. Nie, R. Banerjee, J. J. Li, R. J. Mooney, and M. Gligoric, "Learning deep semantics for test completion," *arXiv preprint arXiv:2302.10166*, 2023.

[35] N. Nashid, M. Sintaha, and A. Mesbah, "Retrieval-based prompt selection for code-related few-shot learning," in *Proceedings of the 45th International Conference on Software Engineering (ICSE'23)*, 2023.

[36] A. Mastropaolo, S. Scalabrino, N. Cooper, D. Nader-Palacio, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Studying the usage of text-to-text transfer transformer to support code-related tasks," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 336–347.

[37] A. Mastropaolo, N. Cooper, D. Nader-Palacio, S. Scalabrino, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Using transfer learning for code-related tasks," *IEEE Trans. Software Eng.*, vol. 49, no. 4, pp. 1580–1598, 2023. [Online]. Available: https://doi.org/10.1109/TSE.2022.3183297

[38] G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, X. Sun, L. Bian, H. Wang, and Z. Wang, "Automated conformance testing for javascript engines via deep compiler fuzzing," in *Proceedings of the 42nd ACM SIGPLAN international conference on programming language design and implementation*, 2021, pp. 435–450.

[39] 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," *arXiv preprint arXiv:2210.02506*, 2022.

[40] S. L. Shrestha and C. Csallner, "Slgpt: using transfer learning to directly generate simulink model files and find bugs in the simulink toolchain," in *Evaluation and Assessment in Software Engineering*, 2021, pp. 260–265.

[41] D. Zimmermann and A. Koziolek, "Automating gui-based software testing with gpt-3," in *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2023, pp. 62–65.

[42] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, X. Che, D. Wang, and Q. Wang, "Chatting with gpt-3 for zero-shot human-like mobile automated gui testing," *arXiv preprint arXiv:2305.09434*, 2023.

[43] A. Khanfir, R. Degiovanni, M. Papadakis, and Y. L. Traon, "Efficient mutation testing via pre-trained language models," *arXiv preprint arXiv:2301.03543*, 2023.

[44] Z. Liu, C. Chen, J. Wang, X. Che, Y. Huang, J. Hu, and Q. Wang, "Fill in the blank: Context-aware automated text input generation for mobile gui testing," *arXiv preprint arXiv:2212.04732*, 2022.

[45] 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," *arXiv preprint arXiv:2304.02014*, 2023.

[46] ——, "Large language models are zero shot fuzzers: Fuzzing deep learning libraries via large language models," *arXiv preprint arXiv:2209.11515*, 2023.

[47] Y. Deng, J. Yao, Z. Tu, X. Zheng, M. Zhang, and T. Zhang, "Target: Traffic rule-based test generation for autonomous driving systems," *arXiv preprint arXiv:2305.06018*, 2023.

[48] C. Tsigkanos, P. Rani, S. Müller, and T. Kehrer, "Large language models: The next frontier for variable discovery within metamorphic testing?" in *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2023, Taipa, Macao, March 21-24, 2023*, T. Zhang, X. Xia, and N. Novielli, Eds. IEEE, 2023, pp. 678–682. [Online]. Available: https://doi.org/10.1109/SANER56733.2023.00070

[49] ——, "Variable discovery with large language models for metamorphic testing of scientific software," in *Computational Science - ICCS 2023 - 23rd International Conference, Prague, Czech Republic, July 3-5, 2023, Proceedings, Part I*, ser. Lecture Notes in Computer Science, J. Mikyska, C. de Mulatier, M. Paszynski, V. V. Krzhizhanovskaya, J. J. Dongarra, and P. M. A. Sloot, Eds., vol. 14073. Springer, 2023, pp. 321–335. [Online]. Available: https://doi.org/10.1007/978-3-031-35995-8_23

[50] T. Zhang, I. C. Irsan, F. Thung, D. Han, D. Lo, and L. Jiang, "itiger: an automatic issue title generation tool," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1637–1641.

[51] P. Mahbub, O. Shuvo, and M. M. Rahman, "Explaining software bugs leveraging code structures in neural machine translation," *arXiv preprint arXiv:2212.04584*, 2022.

[52] N. D. Bui, Y. Wang, and S. Hoi, "Detect-localize-repair: A unified framework for learning to debug with codet5," *arXiv preprint arXiv:2211.14875*, 2022.

[53] S. Kang, J. Yoon, and S. Yoo, "Large language models are few-shot testers: Exploring llm-based general bug reproduction," *arXiv preprint arXiv:2209.11515*, 2022.

[54] S. Kang, B. Chen, S. Yoo, and J.-G. Lou, "Explainable automated debugging via large language model-driven scientific debugging," *arXiv preprint arXiv:2304.02195*, 2023.

[55] T.-O. Li, W. Zong, Y. Wang, H. Tian, Y. Wang, and S.-C. Cheung, "Finding failure-inducing test cases with chatgpt," *arXiv preprint arXiv:2304.11686*, 2023.

[56] 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," *arXiv preprint arXiv:2304.08191*, 2023.

[57] Z. Fan, X. Gao, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models," *arXiv preprint arXiv:2205.10583*, 2022.

[58] J. A. Prenner, H. Babii, and R. Robbes, "Can openai's codex fix bugs? an evaluation on quixbugs," in *Proceedings of the Third International Workshop on Automated Program Repair*, 2022, pp. 69–75.

[59] Y. Hu, X. Shi, Q. Zhou, and L. Pike, "Fix bugs with transformer through a neural-symbolic edit grammar," *arXiv preprint arXiv:2204.06643*, 2022.

[60] C. S. Xia, Y. Wei, and L. Zhang, "Practical program repair in the era of large pre-trained language models," *arXiv preprint arXiv:2210.14179*, 2022.

[61] J. Zhang, J. Cambronero, S. Gulwani, V. Le, R. Piskac, G. Soares, and G. Verbruggen, "Repairing bugs in python assignments using large language models," *arXiv preprint arXiv:2209.14876*, 2022.

[62] M. Lajkó, V. Csuvik, and L. Vidács, "Towards javascript program repair with generative pre-trained transformer (gpt-2)," in *Proceedings of the Third International Workshop on Automated Program*

Repair, 2022, pp. 61–68.

[63] D. Sobania, M. Briesch, C. Hanna, and J. Petke, "An analysis of the automatic bug fixing performance of chatgpt," *arXiv preprint arXiv:2301.08653*, 2023.

[64] W. Yuan, Q. Zhang, T. He, C. Fang, N. Q. V. Hung, X. Hao, and H. Yin, "Circle: continual repair across programming languages," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 678–690.

[65] 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," *CoRR*, vol. abs/2306.01394, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2306.01394

[66] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2022, pp. 1–18.

[67] F. Ribeiro, R. Abreu, and J. Saraiva, "Framing program repair as code completion," in *Proceedings of the Third International Workshop on Automated Program Repair*, 2022, pp. 38–45.

[68] Y. Wu, N. Jiang, H. V. Pham, T. Lutellier, J. Davis, L. Tan, P. Babkin, and S. Shah, "How effective are neural networks for fixing security vulnerabilities," *arXiv preprint arXiv:2305.18607*, 2023.

[69] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," *arXiv preprint arXiv:2302.05020*, 2023.

[70] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "Inferfix: End-to-end program repair with llms," *arXiv preprint arXiv:2303.07263*, 2023.

[71] C. S. Xia and L. Zhang, "Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt," *arXiv preprint arXiv:2304.00385*, 2023.

[72] Y. Zhang, G. Li, Z. Jin, and Y. Xing, "Neural program repair with program dependence analysis and effective filter mechanism," *arXiv preprint arXiv:2305.09315*, 2023.

[73] S. Fakhoury, S. Chakraborty, M. Musuvathi, and S. K. Lahiri, "Towards generating functionally correct code edits from natural language issue descriptions," *arXiv preprint arXiv:2304.03816*, 2023.

[74] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, "Vulrepair: a t5-based automated software vulnerability repair," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 935–947.

[75] G. J. Myers, *The art of software testing (2. ed.)*. Wiley, 2004. [Online]. Available: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0471469122.html

[76] P. Farrell-Vinay, *Manage software testing*. Auerbach Publ., 2008.

[77] A. Mili and F. Tchier, *Software testing: Concepts and operations*. John Wiley & Sons, 2015.

[78] S. Lukasczyk and G. Fraser, "Pynguin: Automated unit test generation for python," in *44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2022, Pittsburgh, PA, USA, May 22-24, 2022*. ACM/IEEE, 2022, pp. 168–172. [Online]. Available: https://doi.org/10.1145/3510454.3516829

[79] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On learning meaningful assert statements for unit test cases," in *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 1398–1409.

[80] A. Wei, Y. Deng, C. Yang, and L. Zhang, "Free lunch for testing: Fuzzing deep-learning libraries from open source," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 995–1007.

[81] D. Xie, Y. Li, M. Kim, H. V. Pham, L. Tan, X. Zhang, and M. W. Godfrey, "Docter: documentation-guided fuzzing for testing deep learning API functions," in *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, S. Ryu and Y. Smaragdakis, Eds. ACM, 2022, pp. 176–188.

[82] Q. Guo, X. Xie, Y. Li, X. Zhang, Y. Liu, X. Li, and C. Shen, "Audee: Automated testing for deep learning frameworks," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 2020, pp. 486–498.

[83] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang, "Deep learning library testing via effective model generation," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds. ACM, 2020, pp. 788–799.

[84] Y. He, L. Zhang, Z. Yang, Y. Cao, K. Lian, S. Li, W. Yang, Z. Zhang, M. Yang, Y. Zhang, and H. Duan, "Textexerciser: Feedback-driven text input exercising for android applications," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1071–1087.

[85] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 298–309. [Online]. Available: https://doi.org/10.1145/3213846.3213871

[86] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1–11. [Online]. Available: https://doi.org/10.1145/3180155.3180233

[87] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 416–426.

[88] J. Xuan, M. Martinez, F. DeMarco, M. Clément, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2017.

[89] B. Zhang, B. Haddow, and A. Birch, "Prompting large language model for machine translation: A case study," *arXiv preprint arXiv:2301.07069*, 2023.

[90] Q. Peng, A. Shi, and L. Zhang, "Empirically revisiting and enhancing ir-based test-case prioritization," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 324–336.

[91] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry, "An information retrieval approach for regression test prioritization based on program changes," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, A. Bertolino, G. Canfora, and S. G. Elbaum, Eds. IEEE Computer Society, 2015, pp. 268–279. [Online]. Available: https://doi.org/10.1109/ICSE.2015.47

[92] Y. Su, Z. Xing, X. Peng, X. Xia, C. Wang, X. Xu, and L. Zhu, "Reducing bug triaging confusion by learning from mistakes with a bug tossing knowledge graph," in *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 2021, pp. 191–202. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678574

[93] 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," *arXiv preprint arXiv:1506.03365*, 2015.

[94] LangChain, Inc., "Langchain," 2023, https://docs.langchain.com/docs/.

[95] celerdata.com, "Database testing with llm," 2023, https://celerdata.com/blog/chatgpt-is-now-finding-bugs-in-databases?s=05.

[96] 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 *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, G. T. Leavens, A. Garcia, and C. S. Pasareanu, Eds. ACM, 2018, pp. 307–318. [Online]. Available: https://doi.org/10.1145/3236024.3236054

[97] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," *Empirical Software Engineering*, vol. 17, pp. 531–577, 2012.

[98] G. G. Cabral and L. L. Minku, "Towards reliable online just-in-time software defect prediction," *IEEE Transactions on Software Engineering*, vol. 49, no. 3, pp. 1342–1358, 2022.

[99] Prompt engineering, "Prompt engineering guide," 2023, https://github.com/dair-ai/Prompt-Engineering-Guide.

[100] 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 *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 1609–1620. [Online]. Available: https://doi.org/10.1145/3510003.3510160

[101] L. Shi, Z. Jiang, Y. Yang, X. Chen, Y. Zhang, F. Mu, H. Jiang, and Q. Wang, "ISPY: automatic issue-solution pair extraction from community live chats," in *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 2021, pp. 142–154. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678894

[102] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=jLoC4ez43PZ

[103] W. Wang, G. Li, S. Shen, X. Xia, and Z. Jin, "Modular tree network for source code representation learning," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–23, 2020.

[104] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 261–271.