A Survey on Large Language Models for Software Engineering

Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang, Yun Yang, Weisong Sun, Shengcheng Yu, Zhenyu Chen

Abstract—Software Engineering (SE) is the systematic design, development, and maintenance of software applications, underpinning the digital infrastructure of our modern mainworld. Very recently, the SE community has seen a rapidly increasing number of techniques employing Large Language Models (LLMs) to automate a broad range of SE tasks. Nevertheless, existing information of the applications, effects, and possible limitations of LLMs within SE is still not well-studied.

In this paper, we provide a systematic survey to summarize the current state-of-the-art research in the LLM-based SE community. We summarize 30 representative LLMs of Source Code across three model architectures, 15 pre-training objectives across four categories, and 16 downstream tasks across five categories. We then present a detailed summarization of the recent SE studies for which LLMs are commonly utilized, including 155 studies for 43 specific code-related tasks across four crucial phases within the SE workflow. Besides, we summarize existing attempts to empirically evaluate LLMs in SE, such as benchmarks, empirical studies, and exploration of SE education. We also discuss several critical aspects of optimization and applications of LLMs in SE, such as security attacks, model tuning, and model compression. Finally, we highlight several challenges and potential opportunities on applying LLMs for future SE studies, such as exploring domain LLMs and constructing clean evaluation datasets. Overall, our work can help researchers gain a comprehensive understanding about the achievements of the existing LLM-based SE studies and promote the practical application of these techniques. Our artifacts are publicly available and will continuously updated at the living repository: https://github.com/iSEngLab/AwesomeLLM4SE.

Index Terms—Software Engineering, Large Language Model, AI and Software Engineering, LLM4SE

1 Introduction

Software engineering (SE) stands as an essential pursuit focused on systematically and predictably designing, developing, testing, and maintaining software systems [1]. As software increasingly becomes the infrastructure of various industries (*e.g.*, transportation, healthcare, and education) nowadays, SE plays a crucial role in modern society by ensuring software systems are built in a systematic, reliable, and efficient manner [2]. As a very active area, SE has been extensively investigated in the literature and has sustained attention from both the academic and industrial communities for several decades [3], [4].

Very recently, one of the most transformative advancements in the realm of SE is the emergence of large language models (LLMs). Advanced LLMs (*e.g.*, BERT [5], T5 [6] and GPT [7]) have significantly improved performance across a wide range of natural language processing (NLP) tasks, such as machine translation and text classification. Typically, such models are pre-trained to derive generic language representations by self-supervised training on large-scale unlabeled data and then are transferred to benefit multiple downstream tasks by supervised fine-tuning on limited labeled data. Inspired by the success of LLMs in NLP, many recent attempts have been adopted to boost numerous code-related tasks (*e.g.*, code summarization and code search) with LLMs (*e.g.*, CodeBERT [8] and CodeT5 [9]). The application of LLMs to SE has had a profound impact on the field, transforming how developers approach code-related tasks automatically. For example, ChatGPT [10], one of the most notable LLMs with billions of parameters, has demonstrated remarkable performance in a variety of tasks, showcasing the potential of LLMs to revolutionize the SE industry. Overall, the SE community has seen a rapidly increasing number of a broad range of SE studies equipped with LLMs, already yielding substantial benefits and further demonstrating a promising future in follow-up research.

However, the complex SE workflow (*e.g.*, software development, testing, and maintenance) and a mass of specific code-related tasks (*e.g.*, vulnerability detection, fault localization, and program repair) make it difficult for interested researchers to understand state-of-the-art LLM-based SE research and improve upon them. Besides, the constant emergence of advanced LLMs with different architectures, training methods, sources, and a plethora of fine-tuning methods brings challenges in keeping pace with and effectively utilizing these advancements. For example, researchers have conducted various studies to extensively investigate the effectiveness of LLMs in the field of program repair [11], [11], [12]. These studies encompass different research aspects

- *Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang, Shengcheng Yu and Zhenyu Chen are with the State Key Laboratory for Novel Software Technology, Nanjing University, China.*
E-mail: quanjun.zhang@smail.nju.edu.cn, fangchunrong@nju.edu.cn, serialxy@outlook.com, zhangyaxin032@gmail.com, yusc@smail.nju.edu.cn, zychen@nju.edu.cn
- *Weisong Sun is with the School of Computer Science and Engineering, Nanyang Technological University.*
E-mail: weisong.sun@ntu.edu.sg.
- *Yun Yang is with the Department of Computing Technologies, Swinburne University of Technology, Melbourne, VIC 3122, Australia.*
E-mail: yyang@swin.edu.au
- *Chunrong Fang is the corresponding author.*
- *This work was mainly done when Quanjun Zhang was a visiting student at Swinburne University of Technology.*

Manuscript received xxx xxx, 2023; revised xxx xxx, 2023.

(*e.g.*, empirical and technical studies [13]), types of LLMs (*e.g.*, open-source or closed-source [11]), mode architectures (*e.g.*, encoder-decoder or encoder-only [12]), model parameters (*e.g.*, CodeT5-60M and InCoder-6B [14]), types of bugs (*e.g.*, semantic bugs and security vulnerabilities [15]), and utilization paradigms (*e.g.*, fine-tuning [16], few-shot [17] and zero-shot [18]).

In this paper, we summarize existing work and provide a retrospection of the LLM-based community field after years of rapid development. Community researchers can have a thorough understanding of the advantages and limitations of the existing LLM-based SE techniques. We discuss how LLMs are integrated into specific tasks in the typical workflow of SE research. Based on our analysis, we point out the current challenges and suggest possible future directions for LLM-based SE research. Overall, our work provides a comprehensive review of the current progress of the LLM-based SE community, enabling researchers to obtain an overview of this thriving field and make progress toward advanced practices.

Contributions. To sum up, the main contributions of this paper are as follows:

- *Survey Methodology.* We conduct a detailed analysis of 185 relevant SE studies empowered with LLMs in terms of publication trends and distribution of venues until November 2023.
- *LLMs of Code.* We summarize 30 representative LLMs of Code for the SE community according to different aspects, such as model architectures, pre-training objectives, downstream tasks, and open science.
- *LLM-based SE Research.* We explore the typical application of leveraging the advance of recent LLMs to automate the SE research, involving 155 relevant studies for 43 code-related tasks across four SE phases, *i.e.*, software requirements and design, software development, software testing, and software maintenance.
- *Empirical Evaluation*. We detail existing benchmarks, empirical studies, and the exploration of SE education to understand the process of LLM-based SE research better and facilitate future studies.
- *SE Optimization.* We discuss some other crucial aspects when LLMs are applied in the SE field, such as security attacks and model tuning.
- *Outlook and challenges.* We pinpoint open research challenges and provide several practical guidelines on applying LLMs for future SE studies.

Comparison with Existing Surveys. Watson [19] presents a systematic literature review of research at the intersection of SE and DL. Recently, Zan *et al.* [20] present a survey to organize 27 LLMs for source code. Unlike existing surveys mainly covering either DL&SE or LLMs of Code, our work focuses on the applications of LLMs in SE, particularly the integration of LLMs in the software design, development, and maintenance phases, as well as the corresponding challenges. Besides, our survey summarizes the existing studies until November 2023.

Paper Organization. The remainder of this paper is organized as follows. Section 2 provides a detailed exposition of four research questions and the methodology employed for conducting the survey. Sections 3 summarize existing LLMs of source code. Section 4 illustrates existing SE studies empowered with LLMs. Section 5 summarizes empirical evaluations and Section 6 discusses the optimization of LLMs for SE. Section 7 highlights the challenges and promising opportunities for future research. Section 8 draws the conclusions.

Availability. All artifacts of this study are available in the following public repository. The living repository continuously updates the latest research on LLMs, LLM4SE, and related studies.

https://github.com/iSEngLab/AwesomeLLM4SE

2 Survey Methodology

2.1 Research Questions

To provide a comprehensive overview of LLMs and the current achievements in SE, our work aims to address the following research questions (RQs):

- **RQ1:** How are LLMs designed to support code-related tasks?
  - **RQ1.1:** What LLMs have been released?
  - **RQ1.2:** What pre-training tasks have been used to train LLMs?
  - **RQ1.3:** What downstream tasks are LLMs spread to?
  - **RQ1.4:** How are LLMs open-sourced to support the open science community?
- **RQ2:** How are LLMs used in software engineering research?
- **RQ3:** How are LLMs empirically evaluated in software engineering research?
- **RQ4:** How can LLMs be strengthened and optimized to enhance their applications in software engineering?

2.2 Search Strategy

Following existing DL for SE surveys [21], [3], [19], we divide the search keywords used for searching papers into two groups: (1) a SE-related group containing some commonly used keywords related to SE research; and (2) an LLM-related group containing some keywords related to LLM research. Besides, considering a significant amount of relevant papers from SE, AI, and NLP communities, following Zhang *et al.* [22], we attempt to identify some preliminary search keywords from three sources: (1) existing LLM surveys [20] to derive LLM-related keywords; (2) existing SE surveys [21], [19] to derive SE-related keywords; (3) a limited number of LLM-based SE research papers manually collected from top-tier conferences and journals beforehand to refine LLM-related and SE-related keywords. The search strategy can capture the most relevant studies from existing surveys while achieving better efficiency than a purely manual search. Finally, the complete set of search keywords is as follows

- *SE-related Keywords:* Software Engineering, SE, Software Requirements, Software Design, Software Development, Software Testing, Software Maintenance, Code generation, Code Search, Code Completion, Code Summarization, Fault Detection, Fault Localization, Vulnerability Prediction, Testing Minimization, Test Generation, Fuzzing, GUI testing, NLP testing, Program

TABLE 1
Details of venues of collected papers

Publisher	Venues	Work on LLMs of Code	Work on LLM-based SE	Total
arXiv	N.A.	7	52	59
ICSE	International Conference on Software Engineering	2	3	32
FSE	European Software Engineering Conference and Symposium on Foundations of Software Engineering	1	14	15
ASE	International Conference on Automated Software Engineering	1	8	9
ICLR	International Conference on Learning Representations	6	N.A.	6
ACL	Meeting of the Association for Computational Linguistics	2	N.A.	5
EMNLP	Empirical Methods in Natural Language Processing	4	1	5
ISSTA	International Symposium on Software Testing and Analysis	N.A.	5	5
IEEE Access		N.A.	4	4
NeurIPS	Neural Information Processing Systems	2	2	4
TSE	Transactions on Software Engineering	N.A.	4	4
ICML	International Conference on Machine Learning	1	2	3
ICST	IEEE International Conference on Software Testing	N.A.	3	3
MSR	Mining Software Repositories	N.A.	3	3
KDD	ACM SIGKDD	1	1	2
TOSEM	Transactions on Software Engineering Methodology	N.A.	2	2
Others	N.A.	3	24	27
Total	N.A.	**30**	**155**	**185**

Repair, Code Review, Vulnerability Repair, Patch Correctness.

- *LLM-related Keywords:* LLM, Large Language Model, Language Model, LM, PLM, Pre-trained model, Pre-training, Natural Language Processing, NLP, Machine Learning, ML, Deep Learning, DL, Artificial Intelligence, AI, Transformer, BERT, Codex, GPT, T5, ChatGPT.

Our survey focuses on LLMs in the field of SE, encompassing existing LLMs and their applications in SE workflow. Thus, we classify papers that need to be summarized into two categories. For LLMs research, we search for papers whose titles contain the second keyword set. For LLM-based SE research, a paper is considered relevant only if it contains both sets of keywords. Then we conduct an automated search on three widely used databases until November 2023, *i.e.*, Google Scholar repository, ACM Digital Library, and IEEE Explorer Digital Library. Finally, we retrieve a total of 13,700 papers from three databases by automated keyword searching.

2.3 Study Selection

Once the potentially relevant studies based on our search strategy are collected, we conduct a three-stage paper filtering to further determine which papers are relevant to this survey. First, we attempt to filter out the papers before 2017, considering that the Transformer architecture [23] is proposed in 2017, which is the foundation of LLMs. Second, we automatically filter out any paper less than 7 pages and duplicated papers, resulting in 421 papers. Third, we inspect the remaining papers manually to decide whether they are relevant to the LLM-based SE field. The manual inspection is conducted independently by two authors, and any paper with different decisions will be handed over to a third author to make the final decision. As a result, we collect 28 papers related to the LLM research and 150 papers related to the LLM-based SE research.

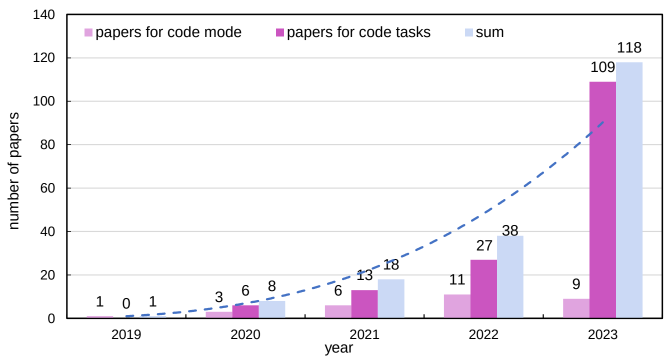


Fig. 1. Number of collected papers over years

To mitigate potential omissions in our automated search and to ensure a thorough collection of papers, we further employed a snowballing search strategy [19]. Snowballing involves meticulously reviewing the reference lists and citations of each paper, to uncover additional relevant studies that our initial search may have missed. In particular, we look at every reference within the collected papers and determine if any of those references are relevant to our study. Through this rigorous manual analysis, we succeed in additionally identifying two papers related to LLMs and 5 papers related to LLM-based SE, thereby enriching our survey with a diverse range of insights.

2.4 Trend Observation

We finally obtain 185 relevant research papers after automated searching and manual inspection. Fig. 1 shows the number of collected papers from 2019 to 2023, It can be observed that studies on proposing LLMs and their applications in SE have rapidly increased since 2019, indicating the growing recognition among researchers of LLMs as a viable and promising approach to automating SE tasks One

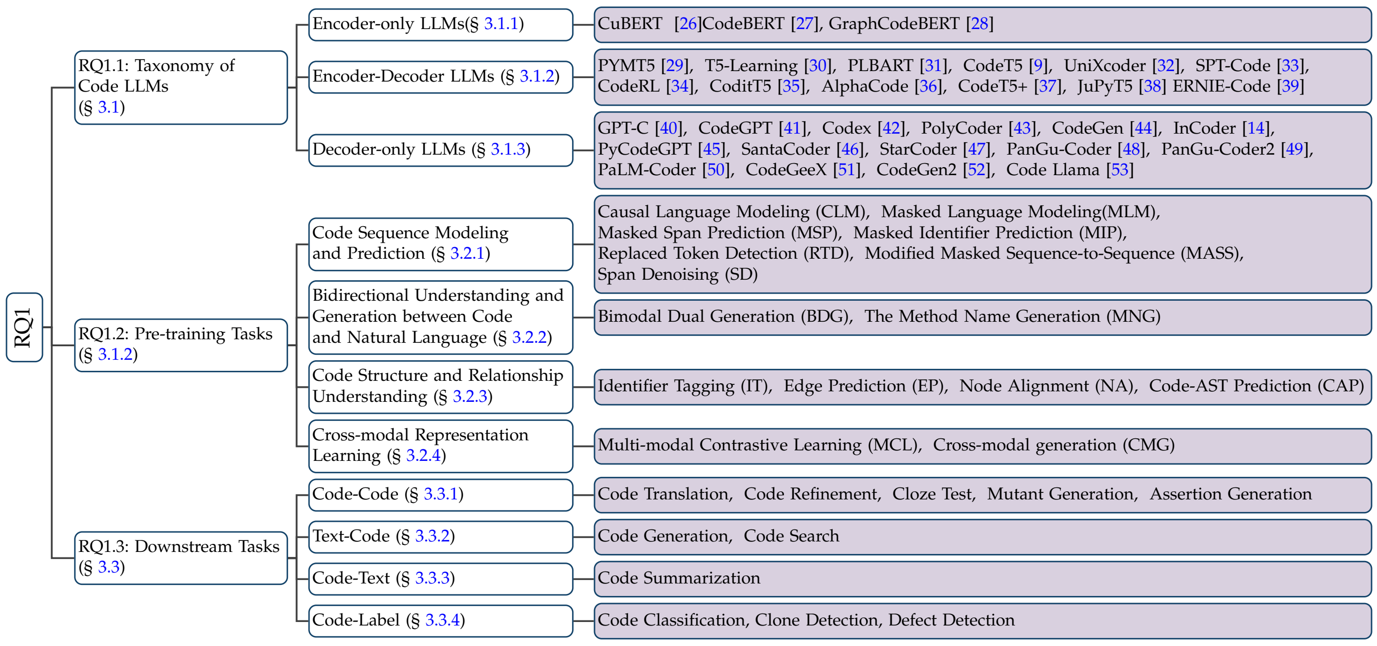

RQ1
RQ1.1: Taxonomy of Code LLMs (§ 3.1)
Encoder-only LLMs(§ 3.1.1)
CuBERT [26]CodeBERT [27], GraphCodeBERT [28]
Encoder-Decoder LLMs (§ 3.1.2)
PYMT5 [29], T5-Learning [30], PLBART [31], CodeT5 [9], UniXcoder [32], SPT-Code [33], CodeRL [34], CoditT5 [35], AlphaCode [36], CodeT5+ [37], JuPyT5 [38] ERNIE-Code [39]
Decoder-only LLMs (§ 3.1.3)
GPT-C [40], CodeGPT [41], Codex [42], PolyCoder [43], CodeGen [44], InCoder [14], PyCodeGPT [45], SantaCoder [46], StarCoder [47], PanGu-Coder [48], PanGu-Coder2 [49], PaLM-Coder [50], CodeGeeX [51], CodeGen2 [52], Code Llama [53]
RQ1.2: Pre-training Tasks (§ 3.1.2)
Code Sequence Modeling and Prediction (§ 3.2.1)
Causal Language Modeling (CLM), Masked Language Modeling(MLM), Masked Span Prediction (MSP), Masked Identifier Prediction (MIP), Replaced Token Detection (RTD), Modified Masked Sequence-to-Sequence (MASS), Span Denoising (SD)
Bidirectional Understanding and Generation between Code and Natural Language (§ 3.2.2)
Bimodal Dual Generation (BDG), The Method Name Generation (MNG)
Code Structure and Relationship Understanding (§ 3.2.3)
Identifier Tagging (IT), Edge Prediction (EP), Node Alignment (NA), Code-AST Prediction (CAP)
Cross-modal Representation Learning (§ 3.2.4)
Multi-modal Contrastive Learning (MCL), Cross-modal generation (CMG)
RQ1.3: Downstream Tasks (§ 3.3)
Code-Code (§ 3.3.1)
Code Translation, Code Refinement, Cloze Test, Mutant Generation, Assertion Generation
Text-Code (§ 3.3.2)
Code Generation, Code Search
Code-Text (§ 3.3.3)
Code Summarization
Code-Label (§ 3.3.4)
Code Classification, Clone Detection, Defect Detection

Fig. 2. Taxonomy of RQ1

possible reason is that DL technologies have already shown promising performance in various SE tasks over the past several years [19]. As a derivative of DL, LLMs bring more powerful code understanding capabilities with larger model sizes and training datasets, demonstrating the potential of being a brand-new way to address SE problems. The second reason is the recent flourishing of the open-source community, which provides millions or even hundreds of millions of open-source code snippets, laying the foundation for training such LLMs.

Table 1 further shows the number of collected papers across different venues. We only consider the venue with more than one paper due to page limitations. First, we find that these papers span multiple research fields, including SE and NLP, which indicates the wide range of attention this direction has received. Second, unlike previous work [24], [25], it can be found that a significant number of papers have not been peer-reviewed. The reason behind this phenomenon lies in the rapid development in this field, especially after the release of the ChatGPT model at the end of 2022, which has stimulated a considerable amount of research in SE. Third, the top five venues are top-tier conferences, such as ICSE, FSE ASE, and ICLR, indicating a current inclination towards conferences in this field, due to the timeliness of conference proceedings.

3 RQ1: How are LLMs designed to support code-related tasks?

Built upon the foundation of the Transformer architecture [23], a mass of LLMs have been proposed with record-breaking parameters in the SE field. In this section, we summarize existing representative LLMs of Code in Section 3.1, pre-training tasks in Section 3.2, fine-tuning tasks in Section 3.3, and discuss the open science issue in Section 3.4. The detailed taxonomy is presented in Fig. 2, including the three sub-RQs and their corresponding categorizations.

3.1 RQ1.1: What LLMs have been released to support SE?

Typically, existing LLMs can be classified into three types according to the model architecture, *i.e.,* encoder-only, decoder-only, and encoder-decoder models. Table 2 presents the summary and comparison of these representative LLMs, The columns summarize the year of release, model name, the publisher or the conference where the model was introduced, the architecture types, and the initialization method or the base model used for pre-training. From Table 2, it can be found that these LLMs are usually derived from foundational architectures in the NLP community and trained with some code-aware objectives. Second, A considerable number of LLMs (*e.g.,* CodeBERT and CodeT5) are introduced by leading companies (*e.g.,* Microsoft and Google). The possible reason is that the resources to train these highly parametric models and to collect vast datasets far exceed the capabilities of the academic community. Third, inspired by the success of foundational LLMs like ChatGPT, the size of model parameters continues to set new benchmarks, and decoder-only architectures gaining increasing popularity. In the following, we summarize these individual LLMs in details.

3.1.1 *Encoder-only LLMs*

Encoder-only LLMs refer to a class of LLMs that utilize only the encoder stack of the Transformer architecture. Regarding architecture, encoder-only models use multiple layers of encoders and each encoder layer consists of a multi-head self-attention mechanism followed by feed-forward neural networks. Regarding training, encoder-only LLMs are typically pre-trained on a massive corpus using a masked language modeling (MLM) task, which is used to learn to predict the identity of masked words based on their context. Regarding usage, because encoder-only LLMs generate fixed-size representations for variable-length input text, they are

TABLE 2
A Summary and Comparison of LLMs of Code

Year	Model	Publisher	Architecture	Size	Tokenizer	Init	Organization	Public
2019	CuBERT	ICML	Encoder-only	350M	N.A.	BERT	Google	Yes
2020	GPT-C	FSE	Decoder-only	366M	BPE	GPT-2	Microsoft	No
2020	PyMT5	EMNLP	Encoder-Decoder	374M	BBPE	GPT-2	Microsoft	No
2020	CodeBERT	EMNLP	Encoder-only	125M	WordPiece	Scratch	Microsoft	Yes
2021	CodeGPT	NeurIPS	Decoder-only	124M	BBPE	GPT-2	Microsoft	Yes
2021	Codex	arXiv	Decoder-only	12M,25M,42M,85M, 300M,679M,2.5B,12B	BBPE	GPT-3	OpenAI	No
2021	CodeT5	EMNLP	Encoder-Decoder	60M,220M	BBPE	Scratch	Salesforce	Yes
2021	T5-Learning	ICSE	Encoder-Decoder	60M	SentencePiece	T5	Università della Svizzera italiana	Yes
2021	PLBART	NAACL	Encoder-Decoder	140M,406M	SentencePiece	Scratch	University of California	Yes
2021	GraphCodeBERT	ICLR	Encoder-only	125M	WordPiece	Scratch	Microsoft	Yes
2022	PolyCoder	ICLR	Decoder-only	41B,160M,400M	BPE	GPT-2	Carnegie Mellon University	Yes
2022	PaLM-Coder	arXiv	Decoder-only	8B,62B,540B	SentencePiece	Scratch	Google	No
2022	PanGu-Coder	arXiv	Decoder-only	317M,2.6B	SentencePiece	Scratch	Huawei	No
2022	PyCodeGPT	IJCAI	Decoder-only	110M	BPE	GPT-Neo	Microsoft	Yes
2022	AlphaCode	Science	Encoder-Decoder	300M,3B,9B,41B	SentencePiece	Scratch	DeepMind	No
2022	JuPyT5	arXiv	Encoder-Decoder	300M	BBPE	PyMT5	Microsoft	Yes
2022	UnixCoder	ACL	Encoder-Decoder	125M	WordPiece	Scratch	Microsoft	Yes
2022	SPT-Code	ICSE	Encoder-Decoder	262M	BPE	Scratch	Nanjing University	Yes
2022	ERNIE-Code	ACL	Encoder-Decoder	560M	SentencePiece	Scratch	Baidu	Yes
2022	CodeRL	NeurIPS	Encoder-Decoder	770M	BBPE	CodeT5	Salesforce	Yes
2022	CoditT5	ASE	Encoder-Decoder	220M	BBPE	CodeT5	The University of Texas at Austin	Yes
2023	CodeGen	ICLR	Decoder-only	350M,2.7B,6.1B,16.1B	BPE	Scratch	Salesforce	Yes
2023	InCoder	ICLR	Decoder-only	6.7B,1.3B	BBPE	Scratch	Meta	Yes
2023	CodeGeeX	KDD	Decoder-only	13B	BBPE	GPT-2	Tsinghua University	Yes
2023	SantaCoder	ICLR	Decoder-only	1.1B	BPE	Scratch	Hugging Face	Yes
2023	StarCoder	arXiv	Decoder-only	15.5B	BBPE	GPT-2	Hugging Face	Yes
2023	CodeGen2	ICLR	Decoder-only	16B,3.7B,7B	BPE	Scratch	Salesforce	Yes
2023	PanGu-Coder2	arXiv	Decoder-only	15B	SentencePiece	PanGu-Coder	Huawei	No
2023	Code Llama	arXiv	Decoder-only	13B,7B,34B	N.A.	Llama2	Microsoft	Yes
2023	CodeT5+	EMNLP	Encoder-Decoder	770M,2B,6B,16B	BBPE	CodeT5	Salesforce	Yes

particularly suited for tasks that require understanding the context or meaning of a piece of text without generating new text, such as code search and vulnerability detection.

Among various encoder-only LLMs, BERT has been acknowledged as a foundational work in the NLP field, and provides crucial guidance for the conception and development of follow-up code-related LLM works, such as CodeBERT and GraphCodeBERT. In the following, we summarize some representative code-related encoder-only LLMs.

CuBERT. CuBERT [26] is the first attempt to apply BERT to source code by replicating the training procedure of BERT on a code corpus. In particular, Kanade *et al.* [26] construct a massive corpus of 7.4M Python files from GitHub and pre-train CuBERT with masked language modeling and next sentence prediction as the objectives. CuBERT is fine-tuned on six downstream tasks, including five classification tasks and one program repair task, demonstrating its superior performance over LSTM and vanilla Transformer models.

CodeBERT. CodeBERT [8] represents a successful adaption of BERT from NLP to the source code domain. CodeBERT follows the BERT architecture (*i.e.,* a multi-layer bidirectional Transformer model), but unlike BERT, which only considers natural language (NL), CodeBERT takes into account both NL and programming language (PL). Regarding input representation, CodeBERT's input is divided into two parts: NL and PL, forming the format $[CLS], w_1, w_2, ..., w_n, [SEP], c_1, c_2, ..., c_m, [EOS]$, where the special marker $[CLS]$ is positioned before these two segments. CodeBERT's output comprises contextual representations for each token and the representation of $[CLS]$. In the pre-training phase, CodeBERT employs two training objectives: masked language modeling and replaced token detection. The first objective aims to predict the original tokens that are masked, a goal established by Devlin *et al.* [5], where only bimodal data (NL-PL pairs) are utilized for training. The second objective is optimized to train on both unimodal and multimodal data, implying that the generator uses both NL and PL data.

Compared with CuBERT [26], CodeBERT is more powerful due to several improvements during pre-training. First, CuBERT is pre-trained with code snippets, while CodeBERT is pre-trained with both bimodal NL-PL data and unimodal PL/NL data. Second, CuBERT is only pre-trained with Python, while CodeBERT is pre-trained with six programming languages. Third, CuEBRT follows the objectives of BERT, while CodeBERT is trained with a new learning objective based on replaced token detection.

❷GraphCodeBERT: Structure-aware Pre-training for Source Code. Although CodeBERT introduces code snippets during pre-training, its training paradigm is still derived from NLP by regarding a code snippet as a sequence of tokens while overlooking the inherent structure of source code. In 2020, Guo *et al.* [28] introduce GraphCodeBERT, a graph-based LLM built upon the BERT architecture designed for code-related applications. GraphCodeBERT employs a representation approach rooted in data flow learning for code. It involves extracting ASTs through tree-sitter and capturing variables from the ASTs to form a sequence of variables. The relationships between extracted variables, such as data source connections, are used to construct a data flow graph. During the model's pre-training phase, GraphCodeBERT introduces two innovative training tasks alongside the inherited MLM task from CodeBERT, *i.e.,* edge prediction and node alignment. The edge prediction task aims to learn code structural information by predicting edges within the data flow graph, while the node alignment task aims to learn which specific node in the data flow graph corresponds to which code token in the input code. Besides, to accommodate the structure of AST graphs, GraphCodeBERT employs graph-guided masked attention.

3.1.2 *Encoder-decoder LLMs*

Encoder-decoder LLMs refer to a class of LLMs that utilize both the encoder and decoder parts of the Transformer architecture, working in tandem to transform one sequence into another. In particular, the encoder takes the input sequence and compresses its information into a fixed-size hidden state, which can capture the essence or meaning of the input sequence, while the decoder takes the hidden state and produces the corresponding output sequence, step by step, often using attention mechanisms to refer back to parts of the input sequence as needed. Thus, this architecture is particularly suited for sequence-to-sequence tasks in NLP and SE, where the input and output sequences can be of different lengths and structures, such as code summarization and program repair.

Among existing encoder-decoder LLMs, T5 (the Text-to-Text Transfer Transformer) is a significant development in the NLP field and serves as a catalyst for follow-up works, such as CodeT5 and UniXCoder. In the following, we summarize some representative code-related encoder-decoder LLMs.

PYMT5: First Attempt of Encoder-decoder LLM. Similar to CuBERT [26] in the encoder-only LLM domain, as early as 2020, PYMT5 [29] is the first attempt to apply encoder-decoder LLMs to source code by replicating the pre-training process of T5 on a code corpus. PYMT5 is pre-trained with a similar span masking objective from T5 on 26 million Python code snippets and built on an encode-decoder Transformer with 374 million parameters. PYMT5 is fine-tuned with two tasks, *i.e.,* method and comment generation, demonstrating superior performance against GPT-2.

T5-Learning: Adaption of T5 for Source Code. In parallel with PYMT5 [29], Mastropaolo *et al.* [30] propose T5-learning, to empirically investigate how the T5 model performs when pre-trained and fine-tuned to support code-related tasks. T5-learning is first pre-trained in a self-supervised way from T5 on CodeSearchNet with both natural language text and programming language code, *i.e.,* masking tokens in code and asking the model to guess the masked tokens. T5-learning is then fine-tuned to support four downstream tasks, *i.e.,* program repair, mutant injection, assertion generation, and code summarization. The results demonstrate that T5-learning outperforms previous baselines, showcasing the potential of T5 in code-related tasks.

PLBART: BART-based LLM for Code. Unlike PYMT5 [29] only focusing on Python code generation, in 2021, Ahmad *et al.* [31] propose PLBART, an encoder-decoder LLM capable of performing a broad spectrum of code understanding and generation tasks. PLABRT is pre-trained with the denoising objective and built on the BART architecture. During the pre-training, PLABRT learns to reconstruct an original text that is corrupted using an arbitrary noise function, including three noise strategies in this work, *i.e.,* token masking, token deletion, and token infilling. PLBART is fine-tuned for two categories of four downstream tasks (*i.e.,* code generation, translation, summarization, and classification) across seven programming languages. The experimental results demonstrate that PLBART outperforms previous LLMs, such as CodeBERT and GraphCodeBERT, demonstrating its promise in both code understanding and generation.

CodeT5: Code-aware T5-based LLM. Despite introducing source code, PLBART simply processes code snippets as natural language and ignores the code-specific characteristics. CodeT5 [8] represents a successful adaption of encoder-decoder LLMs from NLP to the source code domain and has been widely used in SE research. In 2021, Wang *et al.* [9] introduce CodeT5, a unified encoder-decoder LLM based on the T5 architecture by leveraging the code semantics from the developer-assigned identifiers. CodeT5 considers two types of input representations based on whether a code snippet has a corresponding NL description: unimodal (*i.e.,* PL) and bimodal (*i.e.,* PL-NL pairs) data. To encode the input data, CodeT5 concatenates PL and NL into a whole sequence X with a delimiter token $[SEP]$, *i.e.,* $X = (w_1, \cdots, w_n, [SEP], c_1, \cdots, c_m, [SEP]])$, where n and m denote the number of NL word tokens and PL code tokens, respectively. CodeT5 employs three identifier-aware pre-training tasks (*i.e.,* masked span prediction, masked identifier prediction, and identifier tagging) to consider the crucial token type information and a bimodal dual generation pre-training task to learn a better NL-PL alignment between the code and its accompanying comment. CodeT5 is then fine-tuned with the CodeXGLUE benchmark to perform both code generation and understanding tasks, *i.e.,* code summarization, code generation, code translation, code refinement, defect detection, and clone detection. The results demonstrate that CodeT5 significantly outperforms previous LLMs in most downstream tasks, such as RoBERTa, CodeBERT, GraphCodeBERT, GPT2, CodeGPT, and PLBART.

SPT-Code. However, previous LLMs simply reuse the pre-training tasks designed for NL, while failing to learn the the connection between a piece of code and the associated NL for code-related tasks. In May 2022, Niu *et al.* [33] introduce SPT-Code, which is a sequence-to-sequence LLM designed for source code. When given a complete method, SPT-Code aims to acquire general knowledge from the method's source code, its underlying code structure, and the corresponding natural language description. The input is represented as $\{c_1, \cdots, c_l, [SEP], a_1, \cdots, a_m, [SEP], n_1, \cdots, n_p\}$, where l represents the number of code tokens, m denotes the length of the linearized Abstract Syntax Tree (AST) sequence, and p signifies the number of tokens in the natural language description. SPT-Code introduces three specialized code-specific pre-training tasks, *i.e.,* Code-AST Prediction (CAP), Masked Sequence to Sequence (MASS), and Method Name Generation (MNG). Each of these tasks enables SPT-Code to capture a distinct aspect of the data instance. Specifically, CAP focuses on understanding the source code by masking a random fragment of the code tokens. MNG aims to predict whether a given AST accurately represents a particular code fragment, thereby gaining insights into the syntactic structure. Finally, MNG's objective is to generate subtokens corresponding to the method name, a concise natural language description of the method. These three pre-training tasks are meticulously designed to enable SPT-Code to learn about source code, its underlying structure, and the natural language descriptions associated with it. Importantly, SPT-Code does not rely on any bilingual corpora. This knowledge is leveraged when SPT-Code is applied to downstream tasks, making use of these three informational sources.

CodeRL: CodeT5-derived LLM for program synthesis. Unlike previous general-propose LLMs, in 2022, Le *et al.* [34] propose CodeRL, a successor of CodeT5, for the program synthesis task based on deep reinforcement learning. CodeRL is built on top of CodeT5-large architecture with (1) an enlarged pre-training dataset, which has 10.5B tokens and is 10x larger than the CodeSearchNet corpus used in the original CodeT5; and (2) enhanced learning objectives, *i.e.,* masked span prediction and next-token prediction. In particular, CodeRL considers program synthesis as a reinforcement learning problem and applies the actor-critic reinforcement learning method, enhancing CodeT5's performance by leveraging unit test signals during model optimization and generation.

CoditT5: CodeT5-derived LLM for Code Editing. Despite achieving impressive performance in numerous code-related generation tasks, previous LLMs are not well-suited for editing tasks. In 2022, Zhang *et al.* [35] propose CoditT5, an encoder-decoder LLM for code-related editing tasks based on CodeT5. Initialized from the CodeT5-base model, CoditT5 is pre-trained with an edit-aware pre-training objective on the CodeSearchNet dataset, *i.e.,* generating the edit-based output sequence given the corrupted input sequence. CoditT5 is fine-tuned on three downstream tasks, including comment updating, bug fixing, and automated code review, demonstrating superior performance against previous generation-based LLMs (*e.g.,* PLBART and CodeT5) in tackling code editing tasks, such as program repair.

AlphaCode: Competition-level Code Generation LLM. Despite demonstrating remarkable abilities in code generation, previous LLMs have shown limited success when confronted with competition-level programming problems that require problem-solving skills beyond simply translating instructions into code. In 2022, Li *et al.* [36] from DeepMind propose AlphaCode, an encoder-decoder LLM specifically designed to generate solutions for competitive programming solutions problems that require deep reasoning. AlphaCode is built on top of an encoder-decoder transformer-based architecture and is pre-trained with 86.31 million files across 13 programming languages from public GitHub repositories. The encoder and decoder are pre-trained with masked language modeling and next-token prediction objectives, respectively. AlphaCode takes the problem description as input to the encoder and generates a code autoregressively from the decoder one token at a time until an end-of-code token is produced. AlphaCode is then fine-tuned with the CodeContests dataset and the results show that AlphaCode performs roughly at the level of the median competitor, *i.e.,* achieving on average a ranking of top 54.3% in competitions with more than 5,000 participants.

CodeT5+: successor LLM of CodeT5. Although existing LLMs are adept at learning rich contextual representations applicable to a variety of code-related tasks, they often rely on a limited set of pre-training objectives. Such objectives might result in substantial performance degradation in certain downstream tasks due to the discrepancy between the pre-training and fine-tuning stages. In 2023, Wang *et al.* [37] present CodeT5+, a successor of CodeT5 where component modules can be flexibly combined to accommodate a wide range of downstream code tasks. CodeT5+ is pre-trained with two objectives (*i.e.,* span denoising and causal language modeling) on unimodal code corpora and two objectives(*i.e.,* text-code contrastive learning and text-code matching) on bimodal text-code corpora. Codet5+ is built on top of the encoder-decoder Transformer architecture and is classified into two groups according to mode size. CodeT5+ 220M and 770M are trained from scratch following T5's architecture and CodeT5+ 2B, 6B, 16B are initialized from off-the-shelf CodeGen checkpoints [44]. The evaluation experiments are conducted on 20 code-related benchmarks under different settings, including zero-shot, fine-tuning, and instruction-tuning. The experimental results demonstrate that CodeT5+ achieves substantial performance on various code-related tasks, such as code generation and completion, math programming, and text-to-code retrieval tasks

JuPyT5: PyMT5-derived LLM for Jupyter Notebook. Unlike existing LLMs generating code from descriptions, in 2022, Chandel *et al.* [38] propose JuPyT5, an encoder-decoder LLM designed as a data science assistant for the Jupyter Notebook. JuPyT5 is built on the BART architecture and initialized from a pre-trained PyMT5 checkpoint with the same training hyperparameters. JuPyT5 is then pre-trained with a cell-infilling objective on a Data Science Problems (DSP) dataset, which is constructed from almost all publicly available Jupyter Notebook GitHub repositories. DSP consists of 1119 problems curated from 306 pedagogical notebooks with 92 dataset dependencies, natural language and Markdown problem descriptions, and assert-based unit tests. These problems are designed to assess university students' mastery of various Python implementations in Math and Data Science. The experimental results demonstrate that JuPyT5 achieves a 77.5% success rate in solving DSP problems based on 100 sampling attempts, proving the potential of using LLMs as data science assistants.

ERNIE-Code: Multilingual-NL-and-PL LLM. Despite achieving impressive performance in various SE tasks, existing LLMs has been essentially connecting English texts (*e.g.,* comments or docstring) and multilingual code snippets (*e.g.,* Python and Java). Such an English-centricity issue dramatically limits the application of such LLMs in practice, given that 95% of the world's population are non-native English speakers. In 2023, Chai *et al.* [39] from Baidu propose ERNIE-Code, which is a unified LLM designed to bridge the gap between multilingual natural languages (NLs) and multilingual programming languages (PLs). The cross-lingual NL-PL ability of ERNIE-Code is learned from two pre-training tasks, *i.e.,* span-corruption language modeling and Pivot-based translation language modeling. The former learns intra-modal patterns from PL or NL only, while the latter learns cross-modal alignment from many NLs and PLs. ERNIE-Code is built on the T5 encoder-decoder architecture and trained on PL corpus (*i.e.,* CodeSearchNe with six PLs), monolingual NL corpus (*i.e.,* CC100 with monolingual NLs), and parallel NL corpus (*i.e.,* OPUS with 105 bilingual pairs). ERNIE-Code is capable of understanding and generating code and text in 116 different NLs and 6 PLs, and outperforms previous LLMs such as PLBART and

CodeT5 in various code tasks, such as code-to-text, text-to-code, code-to-code, and text-to-text generation. Importantly, ERNIE-Code demonstrates superior performance in zero-shot prompting for multilingual code summarization and text-to-text translation.

3.1.3 Decoder-only LLMs

Decoder-only LLMs refer to a class of LLMs that utilize only the decoder portion of the Transformer architecture. These models are specifically designed for generating sequences of text. Unlike encoder-decoder models, which map an input sequence to an output sequence, decoder-only models primarily focus on generating text based on a given context or prompt. In particular, decoder-only models use multiple layers of decoders from the Transformer architecture. Each decoder layer consists of a multi-head self-attention mechanism followed by feed-forward neural networks. These models are designed to generate text autoregressively, meaning they produce one token at a time and use what has been generated so far as context for subsequent tokens. During training, decoder-only LLMs are typically pre-trained using a language modeling objective. The model learns to predict the next word in a sentence based on the preceding words. After pre-training, these models can be fine-tuned on specific tasks, though GPT-3 has shown that with sufficient scale, fine-tuning becomes less necessary, and the model can perform ”few-shot” or ”zero-shot” learning based on prompts alone. Given their design, decoder-only LLMs are particularly suited for text generation tasks. However, with the right prompts and fine-tuning, they can also be used for tasks like text classification, question answering, and more. The versatility of models like GPT-3 has demonstrated that decoder-only models can be adapted to a wide range of tasks.

Among existing decoder-only LLMs, GPT (Generative Pre-trained Transformer) and its subsequent versions (like GPT-2, GPT-3, and so on) are the most well-known examples of decoder-only LLMs. Developed by OpenAI, these models have set multiple benchmarks in various NLP tasks due to their size and the vast amount of data they were trained on. In summary, decoder-only LLMs, with GPT being the flagship model, leverage the decoder part of the Transformer architecture to generate sequences of text. Their design, combined with massive scale and extensive training data, has enabled them to achieve remarkable performance across a wide variety of NLP tasks.

In the following, we summarize some representative code-related decoder-only LLMs.

GPT-C: first attempt LLM for Code Generation. As early as 2020, Svyatkovskiy *et al.* [40] from Microsoft propose GPT-C, a variant of the GPT-2 trained from scratch on a large unsupervised multilingual source code dataset. GPT-C is a multi-layer generative transformer model trained to predict sequences of code tokens of arbitrary types, generating up to entire lines of syntactically correct code. The pre-training dataset contains 1.2 billion lines of code in Python, C#, JavaScript, and TypeScript. The experimental results demonstrate that GPT-C achieves an average edit similarity of 86.7% on code completion tasks for Python programming language. Importantly, GPT-C is implemented as a cloud-based web service, offering real-time code completion suggestions in Visual Studio Code IDE and Azure Notebook environments.

CodeGPT: a variant of GPT-2 for Source Code In 2021, similar to GPT-C, Lu *et al.* [41] from Microsoft propose CodeGPT, a decoder-only Transformer-based LLM, following the model architecture and training objectives of GPT-2. As one of the baseline LLMs in CodeXGLUE, CodeGPT is designed to support code completion and text-to-code generation tasks. CodeGPT undergoes pre-training on the CodeSearchNet dataset, particularly focusing on the Python and Java corpora. There exist two versions of CodeGPT, *i.e.,* the original CodeGPT, which is pre-trained from scratch with randomly initialized parameters; and CodeGPT-adapted, which is re-trained from the checkpoint of GPT-2 on the code corpus. The experimental results show that in code completion tasks on the PY150 and Github Java Corpus datasets, CodeGPT achieves a performance of 70.65%, while its enhanced version, CodeGPT-adapted, reaches 71.28%. In the text-to-code generation task on the CONCODE dataset, CodeGPT attains a CodeBLEU performance of 32.71%, and CodeGPT-adapted achieves 35.98%.

Codex: A descendant of GPT-3 for Code Tasks Inspired by the considerable success of LLMs (such as GPT-3) in NLP and the abundance of publicly available code, Chen *et al.* [42] from OpenAI propose Codex, a descendant of GPT-3 model fine-tuned with publicly available code corpus from GitHub. Codex is primarily trained for the task of generating independent Python functions from docstrings. The HumanEval benchmark is constructed to evaluate the functional correctness of generated code with 164 handwritten programming problems, each accompanied by a function signature, docstring, body, and several unit tests. The experimental results demonstrate that Codex exhibits remarkable performance, with its model solving more problems on the HumanEval dataset than GPT-3 and GPT-J, achieving a success rate of 28.8%. Furthermore, Codex can solve 70.2% of the questions using a repeated sampling strategy, with 100 samples per question. This suggests that generating multiple samples from the model and selecting the optimal solution is a highly effective approach for challenging prompts. Importantly, Codex and descendants are deployed in GitHub Copilot, indicating the power of LLMs in transforming the landscape of code-related tasks.

PolyCoder: open-sourced LLM comparable to Codex. Despite the impressive success of LLMs of code, some powerful LLMs (such as Codex) are not publicly available, preventing the research community from studying and improving such LLMs. In 2022, Xu *et al.* [43] propose PolyCoder, a decoder-only LLM based on GPT-2 architecture. PolyCoder is trained with 249GB of code from 12 programming languages and contains three sizes, 160M, 400M, and 2.7B parameters. The results demonstrate that despite Codex's primary focus on Python, it still performs well on other programming languages, even outperforming GPT-J and GPT-NeoX. However, for the C programming language, PolyCoder outperforms all LLMs including Codex. Importantly, unlike Codex, three PolyCoder models of different sizes are made available for the research community.

CodeGen: LLM for program synthesis. To investigate program synthesis with LLMs, in 2023, Nijkamp *et al.* [44] from Salesforce introduce CodeGen, which employs a selfregressive Transformer architecture and is trained sequentially on natural language and programming language datasets (THEPILE, BIGQUERY, and BIGPYTHON). It is designed for multi-round program synthesis. CodeGen undergoes evaluation for both single-round and multi-round program synthesis. In single-round evaluation, the synthetic benchmark HumanEval is utilized, and it is observed that CodeGen performance improves with data sizes. Experimental results demonstrate that the performance of the CodeGen NL model either surpasses or is comparable to that of GPT-NEO and GPT-J. CodeGen-Multi demonstrates a significant performance advantage over GPT-NEO, GPT-J, and CodeGen-NL. Furthermore, CodeGen-Mono, fine-tuned on a pure Python dataset, exhibits remarkable enhancements in program synthesis.

InCoder: LLM for Code Infilling and Synthesis. Existing LLMs generate code in a left-to-right manner, which may be unsuitable to many many ubiquitous code editing tasks, such as bug fixing. In 2022, Fried *et al.* [14] from Facebook propose InCoder, a decoder-only LLM designed for program synthesis and editing. InCoder is pre-trained by a causal masking objective, *i.e.,* learning through the random replacement of code segments with sentinel tokens, moving them to the end of the sequence. InCoder's training data consists solely of open-licensed code (Apache 2.0, MIT, BSD-2, and BSD-3 licenses) from online sources such as GitHub, GitLab, and StackOverflow. It primarily focuses on Python and JavaScript but encompasses a total of 28 languages, amounting to approximately 200GB of data in total. There are two versions of the publicly released pre-trained models: one with 6.7 billion parameters and another with 1.3 billion parameters. The experimental results demonstrate that InCoder is able to infill arbitrary regions of code under a zero-shot setting for several tasks, such as type inference and comment generation ,InCoder achieves performance roughly equivalent to CodeGen-Multi on the HumanEval benchmark.

PyCodeGPT: LLM for library-oriented code generation. Previous state-of-the-art LLMs are not publicly available, hindering the progress of related research topics and applications. Similar to PolyCoder [43], in 2022, Zan *et al.* [45] propose PyCodeGPT, a publicly available LLM particular designed for Python. PyCodeGPT is derived from GPT-Neo 125M with a vocabulary size of 32K and incorporates a novel byte-level BPE tokenizer tailored for Python source code. The training dataset consists of 13 million Python files with 96GB crawled from GitHub. The experimental results demonstrate that PyCodeGPT achieves a pass@1 score of 8.33% and pass@10 of 13.53% on the HumanEval benchmark, surpassing other LLMs with similar parameters, such as AlphaCode, CodeClippy, and CodeParrot.

SantaCoder. Regarding the removal of personally identifiable information, the BigCode community [46] propose SantaCoder, a decoder-only LLM with 1.1 billion parameters. SantaCoder's architecture is based on GPT-2 with multi-query attention and Fill-in-the-Middle objective. Its training dataset consists of Python, Java, and JavaScript files from The Stack v1.1. The dataset has undergone several preprocessing steps, including partial data removal, near-duplication removal, de-identification of personally identifiable information, and filtering based on line length and the percentage of alphanumeric characters. Files containing test samples from benchmarks such as HumanEval, APPS, MBPP, and MultiPL-E have also been excluded. The experimental results on the MultiPL-E benchmark demonstrate that SantaCoder outperforms InCoder-6.7B and CodeGen-2.7B in code generation and filling tasks.

StarCoder. Committed to developing responsible LLMs, Li *et al.* [47] from Hugging Face present StarCoder and StarCoderBase, which are LLMs for code. StarCoder is trained on Stack v1.2, and to ensure the secure release of open-source LLMs, it has improved the personally identifiable information editing pipeline and introduced innovative attribution tracking tools. StarCoder undergoes evaluations on HumanEval and MBPP, the experimental results show that StarCoder outperforms PaLM, LaMDA, LLaMA, CodeGen-16B-Mono, and OpenAI's code-cushman-001 (12B) on HumanEval.

PanGu-Coder: LLM for text-to-code generation. To address the specific task of text-to-code generation, adapt to more specific language domains, and handle signals beyond natural language, In 2022, Christopoulou1 *et al.* [48] from Huawei propose PanGu-Coder, a decoder-only LLM for text-to-code generation, *i.e.,* generating stand-alone Python functions from docstrings and evaluating the correctness of code examples through unit tests. PanGu-Coder is built on top of the PanGu-Alpha architecture, a uni-directional decoder-only transformer with an extra query layer stacked on top. PanGu-Coder is trained with two objectives, *i.e.,* a causal language modeling on raw programming language data, and a combination of causal language modeling and masked language modeling for the downstream task of text-to-code generation. The results under a zero-shot manner show that PanGu-Coder outperforms industry LLMs such as Codex and AlphaCode on the HumanEval and MBPP datasets.

PanGu-Coder2: LLM with Reinforcement Learning. Inspired by the success of Reinforcement Learning from Human Feedback in LLMs, in 2023, Shen *et al.* [49] propose PanGu-Coder2, a successor of PanGu-Coder with more powerful code generation capability. PanGu-Coder2 is trained with a new training paradigm, *i.e.,* Rank Responses to align Test&Teacher Feedback and built on top of the advanced StarCoder 15B model. The experimental results demonstrate that PanGu-Coder2 is able to outperform previous LLMs, such as StarCoder, CodeT5+, and AlphaCode on HumanEval, CodeEval, and LeetCode benchmarks.

PaLM-Coder: a variant of PaLM for source code. To investigate the captivity of PaLM for source code, in 2022, Chowdhery *et al.* [50] from Google propose PaLM-Coder, a variant of PaLM by code-specific fine-tuning. The based model PaLM is pre-trained with a high-quality corpus of 780 billion tokens, including 196GB of source code from open-source repositories on GitHub. PaLM-Coder is further derived from PaLM with a two-stage fine-tuning process, *i.e.,* (1) an initial fine-tuning over 6.5 billion tokens, consisting of a blend with 60% Python code, 30% multi-language code and 10% natural language; and (2) an extended fine-tuning with 1.9 billion Python code tokens. The experimental results show that PaLM-Coder is able to achieve 88.4% pass@100 on HumanEval and 80.8% pass@80 on MBPP. Besides, PaLM-Coder demonstrates impressive performance on the DeepFix code repair task with a compile rate of 82.1%, opening up opportunities for fixing complex errors that arise during software development.

CodeGeeX: LLM for Multilingual Code Generation. Despite demonstrating impressive performance, previous LLMs (such as Codex) mainly focus on code generation and are closed-source. In 2023, Zheng *et al.* [51] introduce CodeGeeX, a multilingual decoder-only open-sourced LLM with 13 billion parameters for both code generation and translation tasks. CodeGeeX is implemented with the Huawei MindSpore framework and pre-trained on 850 billion tokens from 23 programming languages, including C++, Java, JavaScript, and Go. Besides, on top of the well-known HumanEval benchmark, a multilingual code generation benchmark HumanEval-X is constructed to evaluate CodeGeeX by hand-writing the solutions in C++, Java, JavaScript, and Go. The experimental results demonstrate that CodeGeeX performs exceptionally well in code generation and translation tasks on the HumanEval-X benchmark. Importantly, CodeGeeX has been integrated with Visual Studio Code, JetBrains, and Cloud Studio. It generates 4.7 billion tokens weekly for tens of thousands of active developers, enhancing the coding efficiency of 83.4% of its users. Besides, CodeGeeX is open-sourced, as well as its code, model weights, API, and HumanEval-X, facilitating the understanding and advances in the community.

CodeGen2: A Successor of CodeGen. Considering the high computational cost of training LLMs, in 2023, Nijkamp *et al.* [52] propose CodeGen2, an successor of CodeGen, aimed at addressing the challenge of more efficiently training LLMs for program synthesis and understanding tasks. CodeGen2 provide a training framework along with open-source CodeGen2 models in four variations, including 1B, 3.7B, 7B, and 16B parameters in size. CodeGen2 is trained on the BigPython dataset and evaluated on the Stack dataset to assess its learning performance in HumanEval and HumanEval-Infill tasks. The experimental results demonstrate that CodeGen2 performs well across various model sizes and program synthesis and understanding tasks, outperforming InCoder in the evaluation on HumanEval.

Code Llama: Llama-based LLM for Source Code. On top of the powerful Llama 2 model in NLP, in 2023, Roziere *et al.* [54] from Meta AI propose Code Llama, a series of LLMs specialized in handling code-related tasks. Code Llama exhibits capabilities such as infilling, support for large input contexts, and zero-shot instruction-following abilities. The dataset of Code Llama primarily comprises an extensive collection of programming language content, with a special emphasis on Python language, trained through a code-heavy dataset containing 500B tokens and an additional Python-intensive data mix of 100B tokens. It comprises multiple versions, covering Code Llama, Code Llama - Python, and Code Llama - Instruct with 7B, 13B and 34B parameters each. Code Llama undergoes evaluation on HumanEval and MBPP, the experimental results indicate that Code Llama outperforms LLama and Llama 2 in terms of performance.

Answer to RQ1.1: Overall, existing LLMs are mainly developed along three directions, the encoder-decoder represented by Google's T5, the encoder-only represented by Microsoft's BERT, and the decoder-only represented by OpenAI's GPT. While different model architectures excel in their own areas, it is challenging to pinpoint a single best LLM for all tasks. For example, encoder-only models (like BERT) focus on representing input text and are typically not used for sequence generation tasks, while decoder-only models (like GPT) are primarily used for generating sequences of text without a separate encoding step.

3.2 RQ1.2: How are LLMs used in pre-training tasks?

In this section, we summarize some representative pre-training tasks utilized to train LLMs of Code in the literature. Table 3 categorizes pre-training tasks into four major classes, including code sequence modeling and prediction in Section 3.2.1, bidirectional understanding and generation in Section 3.2.2, code structure and relationship understanding in Section 3.2.3 and cross-modal representation learning in Section 3.2.4. Now, we list and summarize these pre-training tasks as follows.

3.2.1 *Code Sequence Modeling and Prediction*

Such tasks involve predicting and completing code fragments, such as masked spans or identifiers, so as to enhance LLMS' ability to understand and fill in missing parts of code.

Causal Language Modeling (CLM). CLM[1] attempts to predict the next most probable token in a sequence based on the context provided by the previous tokens. Such a task has usually been utilized to train decoder-only LLMs (*e.g.,* CodeGen and CodeGPT) to generate complete programs from the beginning to the end for supporting auto-regressive tasks, such as code completion. For a sequence $x = (x_1, \ldots, x_n)$ of n tokens, the task is to predict the token x_i given previous tokens $(xj : j < i)$. For example, CLM predicts `x` for the given piece of incomplete code `int add(int x, int y){ return`.

Masked Language Modeling (MLM). MLM attempts to predict the original masked word from an artificially masked input sequence and is utilized in encoder-only LLMs such as CodeBERT. Similar to the original BERT, 15% of the code tokens from the input sequence are masked out. As the prediction of masked tokens is made based on the bidirectional contextual tokens, LLMs need to take into account the tokens forward and backward from the masked token in the input sequence. MLM is instrumental in training the model to comprehend not merely isolated code tokens, but also the relationships between tokens within a piece of code.

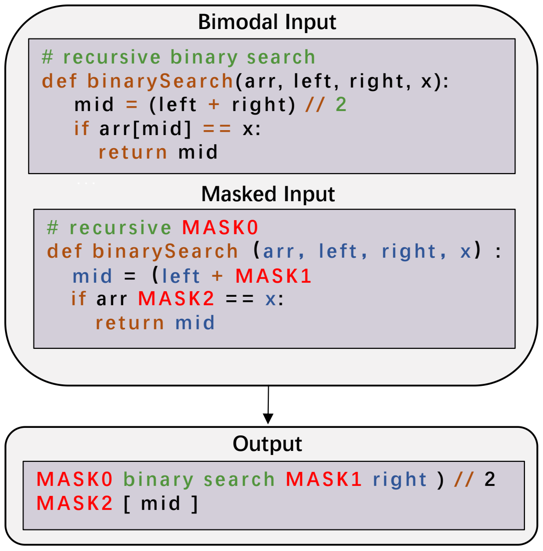
Masked Span Prediction (MSP). MSP attempts to predict the masked code tokens in the input code snippet and is utilized in encoder-decoder LLMs such as CodeT5. Inspired by the original T5, MSP randomly masks spans with arbitrary lengths and then predicts these masked spans

1. The training objective is called causal language modeling in LLMs such as CodeGen2, but also referred to as next token prediction in LLMs such as CodeGen, and unidirectional language modeling in LLMs such as UniXcoder.

TABLE 3
A summary and comparison of pre-training objectiveness in existing LLMs of Code

Category	Task
Code Sequence Modeling and Prediction	Causal Language Modeling (CLM) Masked Language Modeling (MLM) Masked Span Prediction (MSP) Masked Identifier Prediction (MIP) Replaced Token Detection (RTD) Modified Masked Sequence-to-Sequence (MASS) Span Denoising (SD)
Bidirectional Understanding and Generation between Code and Natural Language	Bimodal Dual Generation (BDG) The Method Name Generation (MNG)
Code Structure and Relationship Understanding	Identifier Tagging (IT) Edge Prediction (EP) Node Alignment (NA) Code-AST Prediction (CAP)
Cross-modal Representation Learning	Multi-modal Contrastive Learning (MCL) Cross-modal generation (CMG)

combined with some sentinel tokens at the decoder. Fig. 3 is an example of using MSP in CodeT5 [9], the source sequence is first processed with some noise function, and then the encoder is asked to recover the original text. The input of the model is the original sequence, the mask sequence is processed by the noise function, and the output is the denoised sequence.



```
# recursive binary search
def binarySearch(arr, left, right, x):
   mid = (left + right) // 2
   if arr[mid] == x:
      return mid
```

```
# recursive MASK0
def binarySearch (arr, left, right, x) :
   mid = (left + MASK1
   if arr MASK2 == x:
      return mid
```

```
MASK0 binary search MASK1 right ) // 2
MASK2 [ mid ]
```

Fig. 3. Example of Masked Span Prediction

Masked Identifier Prediction (MIP). Instead of randomly masking spans like in MSP, MIP masks all identifiers in the code snippet, using a unique sentinel token for each different mask. Inspired by the insight that changing identifier names does not impact code semantics, LLMs are tasked to predict the original identifiers from the masked input in an auto-regressive manner. MIP is a more challenging task as it requires the model to comprehend the code semantics based on obfuscated code and link the occurrences of the same identifiers together. Fig. 4 presents an example of using MIP in CodeT5 [9], which arranges the unique identifiers with the sentinel tokens into a target sequence.

Replaced Token Detection (RTD). Originally proposed by Clark *et al.* [55], RTD attempts to predict whether a word is the original word or not, and is utilized in LLMs such as CodeBERT. RTD replaces the original word at the location of the mask with an alternative text, and perform a binary classification problem by training a discriminator to determine if a word is the original one. The discriminator is trained as a binary classifier to distinguish between original and generated tokens. The process involves sampling alternative tokens $\hat{w}_i$ from $p_{Gw}(w_i|w_{masked})$ for positions i in m_w, and sampling alternative tokens $\hat{c}_i$ from $p_{Gc}(c_i|c_{masked})$ for positions i in m_c. Then, the corrupted input $x_{corrupt}$ is formed by replacing the masked words in w and c with their corresponding alternatives. The RTD objective aims to improve the efficiency of training by replacing masked tokens with plausible alternatives, enabling the model to benefit from both bimodal and unimodal data during the learning process.

Modified Masked Sequence-to-Sequence (MASS). MASS attempts to reconstruct a sentence fragment by predicting the masked tokens in the encoder-decoder model architectures and is utilized in LLMs such as SPT-Code. Given a code snippet C, the modified version $C^{u:v}_{origin}$ is obtained by masking the fragment from position u to v. The model is pre-trained using this modified version to predict the fragment of C from u to v As a result, the model learns to predict masked parts of code sequences, enhancing its ability to understand and generate complex code structures accurately.

Span Denoising (SD). SD involves randomly masking a span of tokens in the input and then training the model to reconstruct the original tokens, and is utilized in LLMs such as CodeT5+. In the SD task, 15% of the tokens in the encoder inputs are randomly replaced with indexed sentinel tokens (e.g., [MASK0]), and the decoder is required to recover these masked tokens by generating a combination of spans. SD helps in learning deeper contextual representations of code snippets, enhancing LLMs' understanding of language structure and semantics. In CodeT5+, spans are sampled for masking, where the span lengths are determined by a uniform distribution with a mean of 3, so as to avoid masking partial words and enhance the model's understanding of whole words in the code.

3.2.2 *Bidirectional Understanding and Generation between Code and Natural Language*

Such tasks involve the conversion and understanding between source code and natural language, including generating method names.

Bimodal Dual Generation (BDG) BDG attempts to perform bidirectional translation between PL and NL and is utilized in LLMs such as CodeT5. Specifically, the NL->PL generation and PL->NL generation are treated as dual tasks, and LLMs are optimized simultaneously on both tasks. For each NL->PL bimodal data point, two training instances are created with reverse directions, and language identifiers (*e.g.*, ¡java¿ and ¡en¿ for Java PL and English NL, respectively) are included. The main objective of BDG is to enhance the alignment between the NL and PL, so as to generate syntactically correct NL descriptions for code snippets and code snippets for NL queries in downstream tasks.

Method Name Generation (MNG). MNG attempts to leverage method names to enhance LLMs' understanding of code intent and functionality, and has been utilized in LLMs such as SPT-Code. In the MNG task, the model takes the input representation, denoted as Input $=$ C, [SEP], A, [SEP], N, where C is the code snippet, A is the corresponding AST sequence, and N is a natural language. The method name is represented as name $= c_i$, where i is the index of the method name token in C. The split name $= c_i$ also appears in N. Hence, tokens in N that are derived from the method name split are removed. Assuming that the method name is split into s subtokens, the final input for the MNG task will be as follows:

$$\begin{aligned}\text{Input}_{\text{MNG}} = & c_0, \ldots, c_{i-1}, [\text{MASK}], c_{i+1}, \ldots, c_l, \\ & [\text{SEP}], a_1, a_2, \ldots, a_m, \\ & [\text{SEP}], n_{s+1}, n_{s+2}, \ldots, n_p\end{aligned} \tag{1}$$

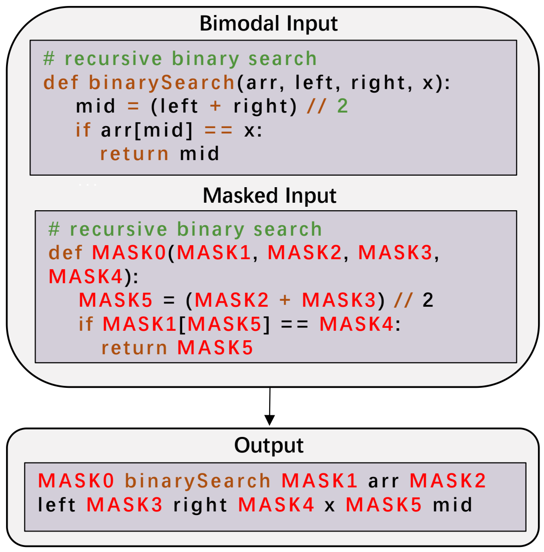
The decoder in the LLM is then tasked with generating the split method name, *i.e.*,, $n_1, n_2, \ldots, n_s$. By incorporating the MNG task, the model can better understand the method names and improve its ability to generate informative code summaries.

3.2.3 *Code Structure and Relationship Understanding*

Such tasks usually involve understanding the structure and relationships within source code, including the arrangement of code elements and their connections.

Identifier Tagging (IT). IT attempts to make LLMs learn whether a code token is an identifier or not and is utilized in LLMs such as CodeT5, which is inspired by the syntax highlighting in some coding tools. IT can help LLMs to learn the code syntax and the data flow structures of source code. Fig. 5 is an example of using IT in CodeT5 [9], which takes a code snippet as the input and returns a sequence of labels of the same length, each of which represents whether the corresponding code token is an identifier or not.

Edge Prediction (EP). EP attempts to learn representations from data flow in the context of code understanding and is utilized in LLMs such as GraphCodeBERT. The primary motivation behind this task is to encourage the

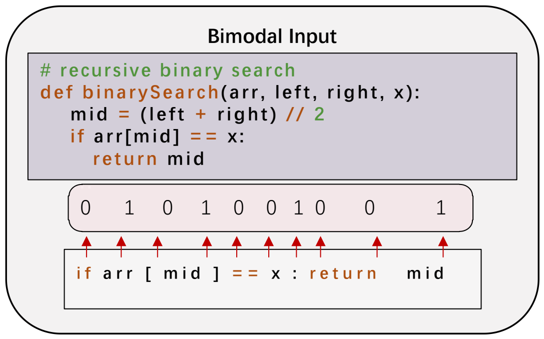


```
# recursive binary search
def binarySearch(arr, left, right, x):
   mid = (left + right) // 2
   if arr[mid] == x:
      return mid
```

```
# recursive binary search
def MASK0(MASK1, MASK2, MASK3,
MASK4):
   MASK5 = (MASK2 + MASK3) // 2
   if MASK1[MASK5] == MASK4:
      return MASK5
```

```
MASK0 binarySearch MASK1 arr MASK2
left MASK3 right MASK4 x MASK5 mid
```

Fig. 4. Example of Masked Identifier Prediction



```
# recursive binary search
def binarySearch(arr, left, right, x):
   mid = (left + right) // 2
   if arr[mid] == x:
      return mid
```

```
0  1  0  1  0  0 1 0   0     1
if arr [ mid ] == x : return  mid
```

Fig. 5. Example of Identifier Tagging

model to learn structure-aware representations that capture the relationships of "where-the-value-comes-from" in the code, thus enhancing its ability to comprehend code. In this pre-training task, a graph representing the data flow is constructed, where nodes represent variables or data elements, and edges represent the flow of data between these nodes. The objective is to predict the edges that are masked (hidden) in the graph. To do this, approximately 20% of the nodes in the data flow graph are randomly sampled, and the direct edges connecting these sampled nodes are masked by adding an infinitely negative value in the mask matrix.

Node Alignment (NA). Similar to EP, NA attempts to align representations between source code and data flow, and is utilized in LLMs such as GraphCodeBERT. This alignment helps the model better understand the relationships between code tokens and nodes in the data flow, leading to improved comprehension of code semantics. In the NA pre-training task, a graph is constructed representing the data flow, and nodes in this graph represent variables or data elements. Additionally, code tokens in the source code are considered as another set of nodes. The objective is to predict the edges between code tokens and nodes, representing the alignment of variables in the code with their data flow counterparts. To achieve this, approximately 20%

of the nodes in the graph are randomly sampled, and edges between code tokens and these sampled nodes are masked. Then, the model predicts which code token a specific node in the data flow is associated with. This prediction helps the model learn the relationships and dependencies between variables and their values in the data flow.

Code-AST Prediction (CAP). Inspired by the NSP task, CAP attempts to incorporate structural information of source code into the pre-training input and is utilized in LLMs such as SPT-Code [33]. The input of CAP includes code and its corresponding abstract syntax tree (AST) representation. The CAP task is formulated as a binarized task that can be easily generated from any given code. In constructing the input representation, the format used is as [Input = C, [SEP], A, [SEP], N], where C represents the code snippet, A represents the corresponding AST sequence, and N is a natural language description.

3.2.4 *Cross-modal Representation Learning*

Such tasks involve understanding source code and other modalities (such as comments) together, enhancing the model's capabilities in understanding and representing code.

Multi-modal Contrastive Learning (MCL). MCL attempts to learn semantic embedding of code fragments by distinguishing between positive and negative samples, and has been utilized by LLMs such as UniXcoder. The positive sample refers to the same input but uses a different hidden dropout mask, while the negative sample refers to other representations in the batch. In UniXcoder [32], MCL encodes the mapped AST sequence and then applies an average pooling layer on the hidden state of the source input to obtain semantic embedding. The contrastive is calculated as shown in formula 2, where b is the batch size, τ is a temperature hyperparameter, and cos is the cos similarity between two vectors.

$$\text{loss}_{MCL} = -\sum_{i=0}^{b-1} \log \frac{e^{\cos\left(\tilde{h}_i, \tilde{h}_i^{+}\right)/\tau}}{\sum_{j=0}^{b-1} e^{\cos\left(\tilde{h}_i, \tilde{h}_j^{+}\right)/\tau}} \tag{2}$$

Cross-modal generation (CMG). CMG attempts to generate comments for code segments to aid the model in understanding the code semantics. The generation of the comment is conditioned on the code, integrating semantic information into the code's hidden states. Furthermore, to enhance the semantic embedding of natural language, there's a strategy of randomly swapping the source and target inputs with a 50% probability. This approach helps in learning more robust language representations by exposing the model to varied contexts.

Answer to RQ1.2: Overall, the recent trends of pre-training tasks for Code LLMs reflect a significant shift from early NLP-derived objectives towards more code-aware objectives. Initially, LLMs are pre-trained with language modeling objectives from NLP tasks, including CLM for decoder-only LLMs (*e.g.*, CodeGPT), MLM for encoder-only LLMs (*e.g.*, CodeBERT), and MSP for encoder-decoder LLMs (*e.g.*, CodeT5). The follow-up works evolve to consider code variables and structural features specifically, as well as cross-modal learning for source code and natural language. This progression signifies a continuous advancement towards LLMs that not only process code as a sequence of tokens but deeply understand its semantic and functional aspects, bridging the gap between source code and natural language.

3.3 RQ1.3: How are LLMs used in downstream tasks?

Once LLMs are trained on a vast corpus, it is critical to evaluate the effectiveness and applicability of LLMs on downstream tasks. Fine-tuning is the primary method for transferring the knowledge acquired during pre-training to downstream tasks, requiring LLMs to demonstrate code understanding, reasoning, and generation capabilities. A downstream task can be categorized by the task type (*i.e.*, code understanding and code generation) or data type (*i.e.*, code-code, code-text, text-code, and code-labels). We summarize 15 representative downstream tasks that are evaluated by existing LLMs in their original papers according to a well-maintained repository[2], detailed as follows.

3.3.1 *Code-Code*

Code-code tasks involve transforming one code snippet into another, such as code translation and code repair.

Code Translation. Code translation involves the process of converting code from one programming language into another while preserving its functionality. It is often used for porting applications, cross-platform development, and legacy system maintenance. Fig. 6 illustrates the general workflow of code translation, where an input code snippet is translated to another programming language.

LLMs in Use: Various LLMs have been applied to code translation tasks, including models like CodeT5 [9], CodeBERT [27], and CodeT5 [9].

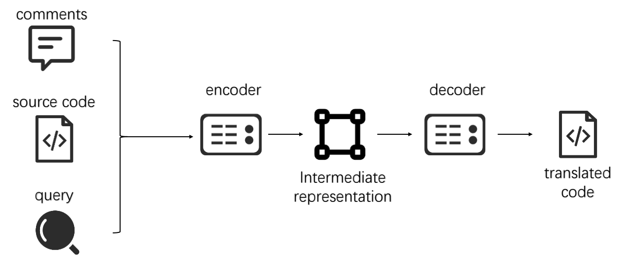


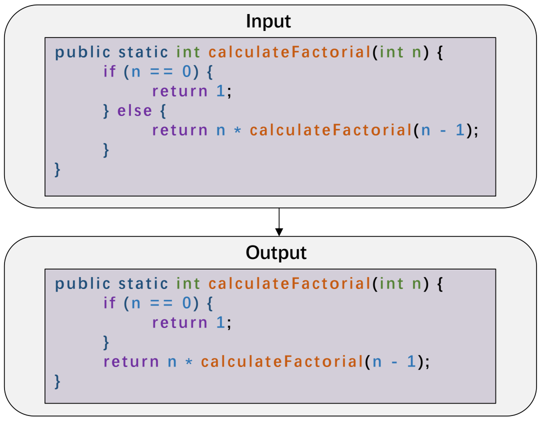
Fig. 6. Example of code translation

Code Refinement. Code refinement is the process of altering a specific piece of code, which might contain errors or be overly complex, with the goal of transforming a flawed function into one that functions correctly. Fig. 7 shows an example of code refinement to the 'else' statement.

LLMs in Use: Various LLMs have been applied to the code translation task, including prominent models like CodeT5 [9], GraphCodeBERT [28] and SPT-Code [33].

Cloze Test Cloze test aims to predict a hidden token in a code segment, involving two parts: one, testing the accuracy of predicting this token from a broad vocabulary, and two, assessing semantic reasoning skills, particularly in differentiating terms like "max" and "min". Fig. 8 shows

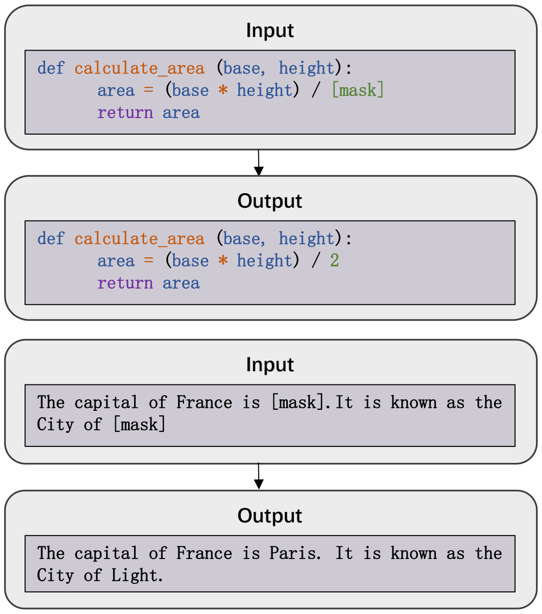
2. https://microsoft.github.io/CodeXGLUE/



```
public static int calculateFactorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * calculateFactorial(n - 1);
    }
}
```

```
public static int calculateFactorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * calculateFactorial(n - 1);
}
```

Fig. 7. Example of code refinement



```
def calculate_area (base, height):
    area = (base * height) / [mask]
    return area
```

```
def calculate_area (base, height):
    area = (base * height) / 2
    return area
```

```
The capital of France is [mask].It is known as the
City of [mask]
```

```
The capital of France is Paris. It is known as the
City of Light.
```

Fig. 8. Example of cloze test

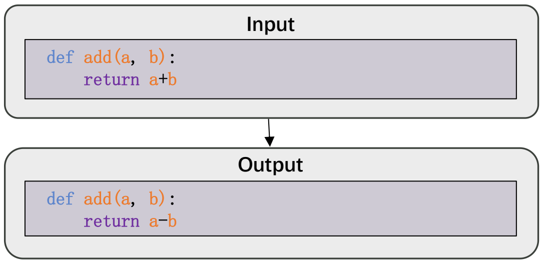
an example of the cloze test in programming language and natural language.

LLMs in Use: Various LLMs have been applied to the code translation task, including prominent models like CodeGPT [51].

Mutant Generation. Mutant generation is a core step in mutation testing. Mutation testing is a fault-based testing method that evaluates the quality of a test suite by introducing small artificial defects (mutations) into the source code, creating variations of the original program, commonly known as mutants. Fig. 9 shows an example of the mutant generation in Python where the ’+’ operator is replaced with ’-’.

LLMs in Use: Various LLMs have been applied to the code translation task, including prominent models like T5-learning [30].

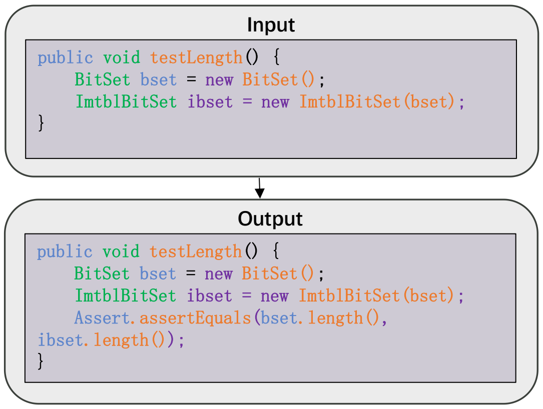
Assertion Generation. Asserts refer to statements that check certain conditions to be true during program execution. Assert generation automatically creates assert statements to verify the correctness of programs and validate certain assumptions. Meaningful assertion generation is one of the key challenges in automatic test case generation. Fig. 10 shows an example of assertion generation for a single method.



```
def add(a, b):
    return a+b
```

```
def add(a, b):
    return a-b
```

Fig. 9. Example of mutant generation



```
public void testLength() {
    BitSet bset = new BitSet();
    ImtblBitSet ibset = new ImtblBitSet(bset);
}
```

```
public void testLength() {
    BitSet bset = new BitSet();
    ImtblBitSet ibset = new ImtblBitSet(bset);
    Assert.assertEquals(bset.length(),
ibset.length());
}
```

Fig. 10. Example of assertion generation

LLMs in Use: Various LLMs have been applied to the code translation task, including prominent models like T5-learning [30].

3.3.2 Text-Code

Text-code tasks involve transforming human language descriptions into code snippets, such as code generation and code search.

Code Generation. Code generation is a task that involves creating code snippets based on natural language descriptions. This process entails taking a problem description, such as ”write a factorial function,” and generating a solution in a specific programming language. The model used for code generation is designed to understand the prompt, which may include declarations and docstrings, and then produce the corresponding implementation of the function.

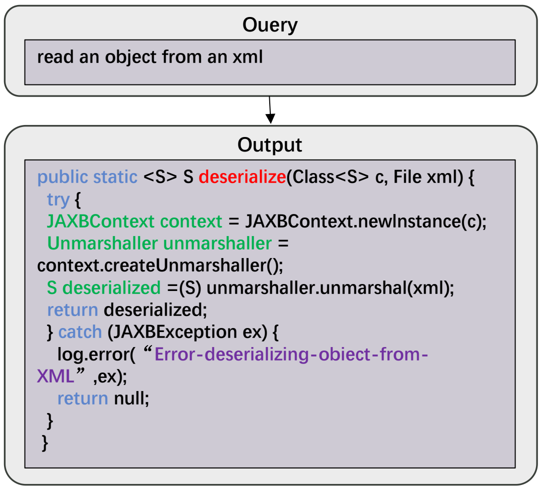
LLMs in Use: LLMs are utilized for Code Generation tasks, including: PyMT5 [29], CodeT5 [9], Codex [42].

❷ **Code Search** Code search is the task of identifying the most relevant code sample from a collection of code candidates that best matches a given natural language query. An example of code search is shown in Fig. 11.

LLMs in Use: LLMs are utilized for Code Generation tasks, including: CodeT5+ [37], CodeGPT [41], UnixCoder [32] and SPT-Code [33].

3.3.3 Code-Text

Code-text tasks involve transforming code snippets into human language descriptions, such as code summarization



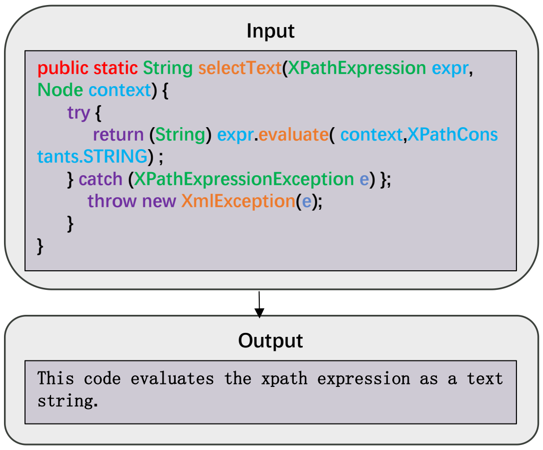
```
read an object from an xml
```

```
public static <S> S deserialize(Class<S> c, File xml) {
  try {
  JAXBContext context = JAXBContext.newInstance(c);
  Unmarshaller unmarshaller =
context.createUnmarshaller();
  S deserialized =(S) unmarshaller.unmarshal(xml);
  return deserialized;
  } catch (JAXBException ex) {
    log.error( “Error-deserializing-object-from-
XML” ,ex);
    return null;
  }
 }
```

Fig. 11. Example of code search

and code review.

Code Summarization. Code summarization is the task of automatically generating a concise and accurate natural language description, or docstring, that encapsulates the actions and purpose of a given source code snippet. Fig. 12 shows an example of code summarization.



```
public static String selectText(XPathExpression expr,
Node context) {
      try {
          return (String) expr.evaluate( context,XPathCons
tants.STRING) ;
      } catch (XPathExpressionException e) };
          throw new XmlException(e);
      }
}
```

```
This code evaluates the xpath expression as a text
string.
```

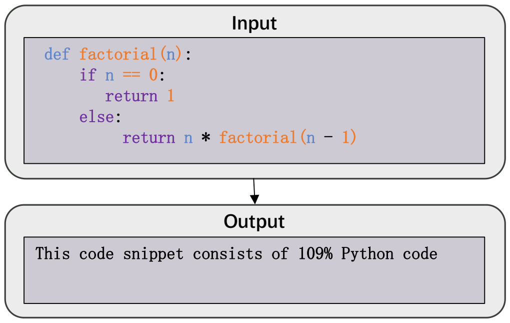
Fig. 12. Example of code summarization

LLMs in Use: Various LLMs have been applied to code translation tasks, including models like CodeT5 [9], GraphCodeBERT [28], PLBART [31], CodeGPT [41], UnixCoder [32], SPT-Code [33] and ERNIE-Code [39].

3.3.4 Code-Label

Code-label tasks involve performing classification on code snippets, such as defect detection and clone detection.

Code Classification.Code classification refers to the process of categorizing or organizing source code files and snippets based on certain criteria. Code classification can be applied for various purposes, such as improving code management, identifying patterns, assisting in code search, and facilitating software maintenance. Generally, source code is classified based on the programming language it is written in, such as Python, Java, C++, etc. Fig. 13 shows an example



```
def factorial(n):
    if n == 0:
       return 1
    else:
          return n * factorial(n - 1)
```

```
This code snippet consists of 109% Python code
```

Fig. 13. Example of code classification

of code classification based on the programming language of the code snippet.

Clone Detection Clone detection is a task focused on identifying whether two code snippets are functionally or semantically similar. This involves measuring the similarity between two pieces of code and predicting if they share the same functionality. Fig. 14 illustrates the general workflow of clone detection, where the process consists of six phases.

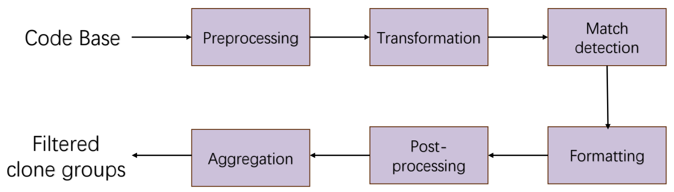


Fig. 14. General Workflow of Clone Detection

- **Preprocessing Phase:** In this phase, models filter irrelevant parts and select the granularity of source units to extract potential source code units for comparison.
- **Transformation Phase:** Comparison units of the source code are transformed into another intermediate internal representation to ease comparison or extract comparable properties.
- **Match Detection Phase:** Transformed code is input to a suitable comparison algorithm where transformed comparison units are compared to find a match.
- **Formatting Phase:** The obtained list is converted to a clone pair list with respect to the original code base.
- **Post-processing Phase:** False positive clones are filtered out.
- **Aggregation Phase:** Clone pairs are aggregated to reduce the amount of data or perform certain analyses.

LLMs in Use: Various LLMs have been applied to clone detection tasks, including models like CodeBERT [8], CodeT5 [9] and CodeT5+ [37], which have demonstrated effectiveness in identifying code clones.

Defect Detection. Defect detection is the task of predicting whether a piece of source code contains defects that could potentially make software systems vulnerable to attacks. The aim is to identify vulnerabilities in the code to prevent exploitation and ensure software security. The effectiveness of defect detection methods is typically evaluated using an accuracy score, indicating how reliably the system can identify defective or vulnerable code. **LLMs**

TABLE 4
The Details of LLMs Availability

Model	Hosting Site	CA	DA	MA	Model Site	URL
CuBERT	GitHub	Yes	Yes	Yes	Google Cloud	https://github.com/google-research/google-research/tree/master/cubert
PyMT5	GitHub	Yes	Yes	Yes	PyPi	https://github.com/devcartel/pymt5
CodeBERT	GitHub	Yes	Yes	Yes	Hugging Face	https://github.com/microsoft/CodeBERT
CodeGPT	GitHub	No	No	Yes	Hugging Face	https://github.com/microsoft/CodeXGLUE
CodeT5	GitHub	Yes	Yes	Yes	Hugging Face	https://github.com/salesforce/CodeT5
T5-Learning	GitHub	Yes	Yes	Yes	Google Drive	https://github.com/antonio-mastropaolo/T5-learning-ICSE_2021
PLBART	GitHub	Yes	Yes	Yes	Google Drive	https://github.com/wasiahmad/PLBART
GraphCodeBERT	GitHub	Yes	Yes	Yes	Hugging Face	https://github.com/microsoft/CodeBERT
PolyCoder	GitHub	No	Yes	Yes	Hugging Face	https://github.com/VHellendoorn/Code-LMs
PyCodeGPT	GitHub	No	Yes	Yes	Hugging Face	https://github.com/microsoft/pycodegpt
JuPyT5	GitHub	No	No	No	N.A.	https://github.com/microsoft/DataScienceProblems
UnixCoder	GitHub	No	Yes	Yes	Hugging Face	https://github.com/microsoft/CodeBERT/tree/master/UniXcoder
SPT-Code	GitHub	Yes	Yes	Yes	OneDrive	https://github.com/NougatCA/SPT-Code
ERNIE-Code	GitHub	Yes	No	No	N.A.	https://github.com/PaddlePaddle/PaddleNLP/tree/develop/model_zoo/ernie-code
CodeRL	GitHub	Yes	Yes	Yes	Google Cloud	https://github.com/salesforce/CodeRL
CoditT5	GitHub	Yes	Yes	Yes	Hugging Face	https://github.com/engineeringsoftware/coditt5
CodeGen	GitHub	No	Yes	Yes	Hugging Face	https://github.com/salesforce/CodeGen
InCoder	GitHub	No	Yes	Yes	Hugging Face	https://sites.google.com/view/incoder-code-models
CodeGeeX	GitHub	No	No	Yes	N.A.	https://github.com/THUDM/CodeGeeX
SantaCoder	Hugging Face	No	Yes	Yes	Hugging Face	https://huggingface.co/bigcode/santacoder
StarCoder	GitHub	No	Yes	Yes	Hugging Face	https://github.com/bigcode-project/starcoder
CodeGen2	GitHub	No	Yes	Yes	Hugging Face	https://github.com/salesforce/CodeGen2
Code Llama	GitHub	Yes	No	Yes	Hugging Face	https://github.com/facebookresearch/codellama
CodeT5+	GitHub	No	Yes	Yes	Hugging Face	https://github.com/salesforce/CodeT5/tree/main/CodeT5%2B

in Use: Various LLMs have been applied to code translation tasks, including models like CodeT5 [9], PLBART [31], and CodeGPT [41].

Answer to RQ1.3: Overall, as the direct applications of LLMs, these downstream tasks can be categorized into four classes according to input-output types, *i.e.*, code-code, code-test, test-code and code-labels, or into two classes according to task types, *i.e.*, code understanding and code generation. We observe some trends in a majority of existing downstream tasks where LLMs can be directly applied. First, these tasks involve only code snippets or the corresponding natural language comments. Second, these tasks are usually evaluated automatically using well-designed metrics (*e.g.*, BLUE for generation tasks and Accuracy for classification tasks), thus supporting the large-scale evaluation benchmarks. Third, these tasks can effectively reduce the programming efforts of developers and can be integrated into modern IDEs as plug-ins to aid programming. Finally, these tasks have received attention and have been investigated in both the fields of SE and artificial intelligence. LLMs have shown preliminarily promising results on these tasks, importantly indicating their potential in a wider and more in-depth range of SE tasks, detailed in Section 4.

3.4 RQ1.4: How are LLMs open-sourced to support the open science community?

Very recently, the literature has seen a surge in the application of LLMs for a variety of SE problems. LLM brings a fresh perspective on the challenges associated with code-related tasks, shifting the focus from traditional learning-based and rule-based approaches to a new pre-training-and-fine-tuning paradigm. However, this shift also presents unique reproducibility challenges, distinct from those in traditional studies. For example, training complex LLMs can require substantial computational resources, often exceeding what academic institutions and most businesses can provide. Besides, the need for extensive data collection and hyper-parameter tuning adds complexity and feasibility issues for replication. Given these challenges, there is a growing imperative to adhere to open science principles in the LLM-driven SE field. Open science encourages researchers to share their artifacts (*e.g.*, datasets, trained models, scripts) with the broader research community, fostering reproducibility and free knowledge exchange. While numerous LLMs have been proposed for automating code-related tasks with promising results, there is a need for more support to address the critical issue of open science. In particular, we investigate the extent to which LLMs make their artifacts publicly available and how they provide this information.

Among 30 investigated LLMs, 27 of them provide the corresponding open-source repositories, which are summarized in Table 4. For each LLM we collect, we check whether an accessible link for its model or data is provided in the main text or footnotes of the paper. We only present the studies that provide the link of publicly available data or tools due to limited space, listed in the first column. The second column lists which hosting site the available artifact is uploaded to for public access (*e.g.*, GitHub). The third column lists whether the source code (*e.g.*, training scripts) is available in the artifacts. The fourth column lists whether the dataset (*e.g.,* raw data and training data) is available in the artifacts. The fifth column lists whether the trained model is available in the artifacts, and the sixth column lists the corresponding site. We also list the accessible URL links in the last column. After carefully checking the collected papers, we find that 27 of 30 LLMs have made their artifacts available to the public. Almost all studies upload their artifacts on Github, which is the most popular platform for hosting open-source code publicly. Similar to GitHub, nearly all checkpoints of LLMs are hosted on Hugging Face, with which developers can conveniently download these trained models and conduct training or inference on their own machines. Meanwhile, we find that several papers fail to provide the source code, dataset, or already trained models [37] perhaps due to commercial reasons.

Answer to RQ1.4: Overall, compared with traditional DL studies, the need for high-quality artifacts in LLMs is even more vital for replication and future research. While numerous LLms have been introduced for code-related tasks, there remains a significant gap in their adherence to open science principles. On one hand, abundant training time and expensive equipment (*e.g.,* GPUs) are required to train LLMs, making it much harder to reproduce existing works. On the other hand, some LLMs require complex environment settings (*e.g.,* the hyperparameters and the random seed) and high-quality datasets. Therefore, we hope that researchers can provide high-quality open-source code and detailed instructions for convenient reproduction.

4 RQ2: How are LLMs used in software engineering research?

In this section, we summarize existing SE studies empowered with LLMs, which can be categorized into four crucial phases within the SE life cycle, including software requirements and design in Section 4.1, software development in Section 4.2, software testing in Section 4.3, and software maintenance in Section 4.4. Each SE phase contains several distinct code-related tasks, such as fault localization and program repair in the software maintenance phase. Fig. 15 presents the taxonomy of this section, including LLMs' applications across four sub-RQs and 43 code-related tasks in the SE domain, summarized as follows.

4.1 Software Requirements & Design

Software requirements refer to specific descriptions of conditions or capabilities needed by users, systems, or system components, typically presented in document form. These requirements are categorized into functional and non-functional requirements. The purpose of software requirements is to ensure that the developed software can meet the expectations of users and relevant stakeholders, as well as the conditions and capabilities specified in contracts, standards, regulations, or other formal documents. Software design involves the process of defining the structure, components, functionalities, interfaces, and their relationships within a software system. During the software design phase, software engineers need to create detailed plans and design blueprints based on software requirements and specifications to ensure that the software system can meet the users' needs and expectations.

4.1.1 Software Specifications Generation

Software specifications generation refers to the automated process of deriving formal descriptions and requirements for software systems from unstructured data sources such as comments or documentation within the software's source code. Traditional techniques for extracting software specifications usually involve rule-based or machine learning-based methods that necessitate manual effort and domain knowledge, and have limited the ability to generalize across various domains.

LLMs provide a promising avenue for automating the process of software specifications generation. LLMs, which have been utilized successfully in numerous software engineering tasks, offer the potential in automatically extracting software specifications from textual information. For example, Xie *et al.* [56] conduct the first empirical study to assess the capabilities of LLMs for generating software specifications from software comments or documentation. They employ few-shot learning techniques to enable LLMs to generalize from a limited number of examples and explore various prompt construction strategies. Additionally, the research conducts a comparative diagnosis of failure cases between LLMs and traditional methods to identify their respective strengths and weaknesses.

4.1.2 Software Specifications Repair

Software Specifications Repair refers to the process of fixing errors in software specifications, which are formal declarations of software system requirements or behaviors. This repair process has become increasingly significant with the growing complexity and usage of declarative languages like Alloy. The recent integration of LLMs like ChatGPT in this domain aims to automate and enhance the effectiveness of repair techniques. These LLMs are evaluated for their ability to correct inaccuracies in specifications and compared against existing automated program repair methods. The process involves identifying and rectifying various types of errors in software specifications, including logical inconsistencies, type errors, and misuse of programming constructs.

In 2023, Hasan *et al.* [57] evaluates the use of ChatGPT for repairing software specifications in the Alloy declarative language. It aims to assess ChatGPT's capabilities in correcting errors and identifies challenges in making it a viable solution. The study compares ChatGPT's performance with current APR techniques, including ARepair, BeAFix, ATR, and ICEBAR, across two benchmark suites, namely ARepair and Alloy4Fun. The findings demonstrate that ChatGPT successfully addresses unique errors that other tools fail to rectify, although it does not consistently surpass them in total repair count. The research also identifies various error types within ChatGPT-generated repairs, highlighting areas for improvement in repair effectiveness and consistency.

4.1.3 Requirements Classification

Requirement classification refers to the process of categorizing software requirements into different classes or

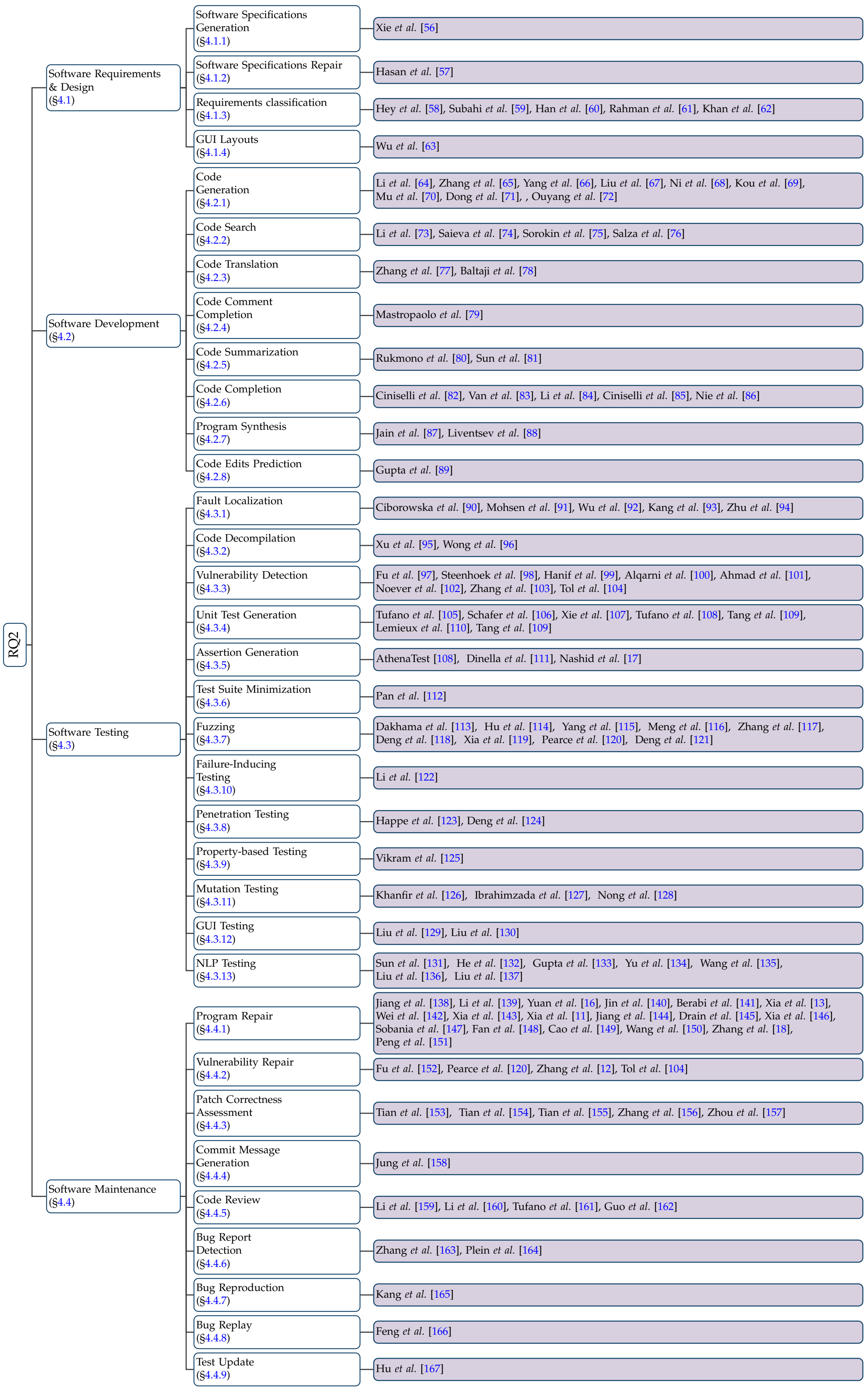

RQ2
Software Requirements & Design (§4.1)
Software Specifications Generation (§4.1.1)
Xie et al. [56]
Software Specifications Repair (§4.1.2)
Hasan et al. [57]
Requirements classification (§4.1.3)
Hey et al. [58], Subahi et al. [59], Han et al. [60], Rahman et al. [61], Khan et al. [62]
GUI Layouts (§4.1.4)
Wu et al. [63]
Software Development (§4.2)
Code Generation (§4.2.1)
Li et al. [64], Zhang et al. [65], Yang et al. [66], Liu et al. [67], Ni et al. [68], Kou et al. [69], Mu et al. [70], Dong et al. [71], , Ouyang et al. [72]
Code Search (§4.2.2)
Li et al. [73], Saieva et al. [74], Sorokin et al. [75], Salza et al. [76]
Code Translation (§4.2.3)
Zhang et al. [77], Baltaji et al. [78]
Code Comment Completion (§4.2.4)
Mastropaolo et al. [79]
Code Summarization (§4.2.5)
Rukmono et al. [80], Sun et al. [81]
Code Completion (§4.2.6)
Ciniselli et al. [82], Van et al. [83], Li et al. [84], Ciniselli et al. [85], Nie et al. [86]
Program Synthesis (§4.2.7)
Jain et al. [87], Liventsev et al. [88]
Code Edits Prediction (§4.2.8)
Gupta et al. [89]
Software Testing (§4.3)
Fault Localization (§4.3.1)
Ciborowska et al. [90], Mohsen et al. [91], Wu et al. [92], Kang et al. [93], Zhu et al. [94]
Code Decompilation (§4.3.2)
Xu et al. [95], Wong et al. [96]
Vulnerability Detection (§4.3.3)
Fu et al. [97], Steenhoek et al. [98], Hanif et al. [99], Alqarni et al. [100], Ahmad et al. [101], Noever et al. [102], Zhang et al. [103], Tol et al. [104]
Unit Test Generation (§4.3.4)
Tufano et al. [105], Schafer et al. [106], Xie et al. [107], Tufano et al. [108], Tang et al. [109], Lemieux et al. [110], Tang et al. [109]
Assertion Generation (§4.3.5)
AthenaTest [108], Dinella et al. [111], Nashid et al. [17]
Test Suite Minimization (§4.3.6)
Pan et al. [112]
Fuzzing (§4.3.7)
Dakhama et al. [113], Hu et al. [114], Yang et al. [115], Meng et al. [116], Zhang et al. [117], Deng et al. [118], Xia et al. [119], Pearce et al. [120], Deng et al. [121]
Failure-Inducing Testing (§4.3.10)
Li et al. [122]
Penetration Testing (§4.3.8)
Happe et al. [123], Deng et al. [124]
Property-based Testing (§4.3.9)
Vikram et al. [125]
Mutation Testing (§4.3.11)
Khanfir et al. [126], Ibrahimzada et al. [127], Nong et al. [128]
GUI Testing (§4.3.12)
Liu et al. [129], Liu et al. [130]
NLP Testing (§4.3.13)
Sun et al. [131], He et al. [132], Gupta et al. [133], Yu et al. [134], Wang et al. [135], Liu et al. [136], Liu et al. [137]
Software Maintenance (§4.4)
Program Repair (§4.4.1)
Jiang et al. [138], Li et al. [139], Yuan et al. [16], Jin et al. [140], Berabi et al. [141], Xia et al. [13], Wei et al. [142], Xia et al. [143], Xia et al. [11], Jiang et al. [144], Drain et al. [145], Xia et al. [146], Sobania et al. [147], Fan et al. [148], Cao et al. [149], Wang et al. [150], Zhang et al. [18], Peng et al. [151]
Vulnerability Repair (§4.4.2)
Fu et al. [152], Pearce et al. [120], Zhang et al. [12], Tol et al. [104]
Patch Correctness Assessment (§4.4.3)
Tian et al. [153], Tian et al. [154], Tian et al. [155], Zhang et al. [156], Zhou et al. [157]
Commit Message Generation (§4.4.4)
Jung et al. [158]
Code Review (§4.4.5)
Li et al. [159], Li et al. [160], Tufano et al. [161], Guo et al. [162]
Bug Report Detection (§4.4.6)
Zhang et al. [163], Plein et al. [164]
Bug Reproduction (§4.4.7)
Kang et al. [165]
Bug Replay (§4.4.8)
Feng et al. [166]
Test Update (§4.4.9)
Hu et al. [167]

Fig. 15. Taxonomy of the application of LLMs in different domains within software engineering

types. Software requirements typically encompass both functional and non-functional requirements. Functional requirements outline the functionalities and behaviors the software should achieve, while non-functional requirements cover broader system attributes such as performance, security, reliability, and maintainability. Through requirement classification, software developers can better organize and comprehend various types of requirements.

As early as 2020, Hey *et al.* [58] propose NoRBERT, a BERT-based requirement classification approach for both functional and non-functional classes by transfer learning. NoRBERT mainly leverages a pre-trained BERT model on a large corpus of text and fine-tunes it with a small amount of task-specific data. In particular, NoRBERT first categorizes requirements into functional and non-functional classes and further sub-categorizes non-functional requirements. Through experimentation on the PROMISE NFR dataset, NoRBERT demonstrates excellent classification performance, particularly showcasing strong generalization capabilities when dealing with unknown projects. Compared to traditional approaches, NoRBERT doesn't require manual preprocessing and maintains strong performance across various projects. Furthermore, NoRBERT introduces a novel task of classifying functional requirements based on their contained concerns (Function, Data, Behavior). This novel classification task aims to interpret the purposes and attributes of functional requirements automatically. In this task, NoRBERT displays high F1 scores, reaching a maximum accuracy of 92%.

In contrast to functional requirements, non-functional requirements are often overlooked, potentially leading to rework, increased maintenance, and inefficient resource utilization, impacting project costs. Unlike NoRBERT [58] considering both functional and non-functional requirements, Khan *et al.* [62] discuss the performance of LLMs in identifying and categorizing non-functional requirements. By employing transfer learning methods, Khan *et al.* evaluate multiple models, including XLNet, BERT, Distil BERT, Distil Roberta, Electra-base, and Electra-small. Among these, XLNet exhibits outstanding performance and is selected as the primary model to enhance NFR identification and classification. The experiments demonstrate that the integrated XLNet model can overcome the limitations of relying solely on previous models, such as BERT, for context understanding. It showcases successful applications of XLNet in a mobile banking application of a financial institution, resulting in significant improvements in stability, response time, and user satisfaction. These enhancements lead to increased user adoption rates and customer loyalty. Similarly, Subahi *et al.* [59] propose the transfer learning of BERT to map non-functional requirements to various dimensions of green software sustainability, constructing a proof-of-concept model for requirement classification. This method aims to address sustainability issues in software engineering by associating requirements with dimensions of green software sustainability. Leveraging the pre-training and fine-tuning capabilities of BERT, as well as domain-specific optimizations, the research provides an automated method for measuring sustainability in the software development phase of requirements engineering. Furthermore, the study emphasizes the significance of fine-tuning and transfer learning in classification tasks within requirements engineering.

To address the issue that traditional leaning-based approaches are limited by the lack of annotated training data in the domain of requirement engineering, Rahman *et al.* [61] propose to classify non-Functional requirements within software requirements specification documents by extracting features from pre-trained word embedding models. The new algorithm is designed specifically to extract relevant representative features from these pre-trained word embedding models. Additionally, each pre-trained model is paired with four customized neural network architectures (RPCNN, RPBiLSTM, RPLSTM, and RPANN) for NFR classification. This amalgamation yields 96 unique models, each possessing distinct configurations and features. The research findings demonstrate that the integration of the pre-trained GloVe model with RPBiLSTM exhibits the best performance, achieving an average AUC score of 85%, precision of 82%, and recall of 82%, highlighting its strong capability to accurately classify NFR. However, this study shows some limitations, such as generic models being unable to encompass domain-specific vocabulary and context, leading to suboptimal performance in NFR prediction and classification.

However, most of these models are considered black-box models, and thus the rationale of the classification is unclear, affecting the improvement of the classification model. In this paper, we propose a technique for improving requirements classification models based on an explainable AI (XAI) framework. Considering that previous work leaves LLMs in a black-box manner, Han *et al.* [60] propose a requirement classification approach based on BERT and an an explainable AI framework. First, they train a concern extraction model to extract concerns from requirement texts, which offers valuable insights into the decisive factors in requirement classification. Second, they employ SHAP, a widely-used explainability method, to generate explanations for the predictions of the requirement classification model, which may shed light on the noise in the dataset. Third, they create a new dataset by removing the identified noise and then fine-tune EBRT to obtain a more robust and accurate improved model. The experimental results indicate that the proposed approach significantly enhances the model's performance, with an overall increase in accuracy by 7.68%, a 7.44% boost in recall, and a 7.80% improvement in F1-score.

4.1.4 GUI layouts

GUI layouts refer to the arrangement and organization of various elements, such as widgets, images, banners, and icons, within the design of a GUI. The purpose of GUI layouts is to provide a user-friendly interface and ensure a positive user experience. This encompasses how various interface components are effectively positioned and displayed to facilitate user understanding and interaction with the application. Designing layouts involves considerations of spatial relationships between elements, overall page structure, usability, aesthetics, and other factors to create an intuitive, user-friendly, and visually pleasing user interface.

In 2023, Wu *et al.* [63] focus on the application of LLMs in GUI layouts, addressing the specific research question

TABLE 5
A summary and comparison of existing LLM-based fault localization studies

Year	Study	LLMs	Type	Repository
2023	Li *et al.* [64]	CodeGeeX, CodeGen, InCoder	Requirement-guided	N.A.
2023	Zhang *et al.* [65]	CodeGen, InCoder, GPT-Neo, GPT-J, PycodeGPT	Execution-feedback	N.A.
2023	Yang *et al.* [66]	CodeT5	Requirement-guided	https://github.com/NTDXYG/TurduckenGen
2023	Liu *et al.* [67]	ChatGPT	Empirical Study	N.A.
2023	Ni *et al.* [68]	GPT-3.5	Execution-feedback	https://github.com/niansong1996/lever
2023	Kou *et al.* [69]	InCoder, CoderGen, PolyCoder, CodeParrot, GPT-J	Empirical Study	N.A.
2023	Mu *et al.* [70]	GPT-4	Requirement-guided	https://github.com/ClarifyGPT/ClarifyGPT
2023	Ouyang *et al.* [72]	ChatGPT	Empirical Study	https://github.com/CodeHero0/Nondeterminism-of-ChatGPT-in-Code-Generation
2023	Dong *et al.* [71]	ChatGPT	Execution-feedback	N.A.

of whether a system based on LLMs can effectively search for relevant GUI layouts. Through a controlled study on Instigator, they evaluated its performance in searching web page GUI layouts. The results demonstrated a high correlation between the rankings of GUI layouts generated by Instigator based on given instructions and the rankings provided by practitioners, proving its ability to avoid generating irrelevant layouts. The paper further conducted a detailed analysis of Instigator's positive performance in transparency, intuitive use, usefulness, trustworthiness, and efficiency through user experience evaluations involving 34 practitioners. Finally, the paper showcased how GUI layouts generated by Instigator can be edited and combined to form the final GUI, providing a seamless integration into the user interface development lifecycle.

Overall, through the utilization of LLMs, particularly systems like Instigator, it is possible to effectively search, edit, and combine GUI layouts, thereby delivering high-quality user interface designs throughout the entire development lifecycle.

4.2 Software Development

Software Development is a creative process involving the use of computer programming languages, tools, and techniques to transform user requirements, functionality, and performance requirements into computer programs.

4.2.1 Code Generation

Code generation plays a pivotal role during software development and has always been the primary focus of most Code LLMs, such as AlphaCode [36] and CodeGen [44]. Code generation aims at automatically generating code snippets according to a requirement specification. Overall, we categorize the advancements of existing LLMs in code generation into three types, which are summarized in
detailed as follows.

Requirement-guided code generation.

In the early stages of development, developers commonly employed requirement-guided code generation, *i.e.,* taking a natural language description as the input and returning the correct code snippet, which is evaluated by corresponding unit tests. For example, Li *et al.* [64] propose AceCoder, which involves requirement-guided code generation and example retrieval to address two major challenges in code generation: requirement understanding and code implementation. Requirement-guided code generation entails LLMs initially generating an intermediate preparatory stage, such as test cases, to analyze and clarify requirements. This process aids in identifying specific details, input-output formats, and special cases. Example retrieval, inspired by human developers' code reuse behavior, selects similar programs as examples in prompts, teaching LLMs how to implement code by providing relevant content, such as algorithms and APIs. They discuss AceCoder's three-step process: example retrieval, prompt construction, and code generation, showcasing its application across three code generation benchmarks (MBPP, MBJP, MBJSP) in Python, Java, and JavaScript.

At the same time, Mu *et al.* [70] propose ClarifyGPT, with a similar premise of identifying ambiguity and unclear elements in user requirements and presenting targeted clarifying questions to enhance the quality and efficiency of code generation. Specifically, ClarifyGPT initially detects ambiguity in user requirements through code consistency checks, followed by guiding LLMs to generate specific clarifying questions for unclear demands. These questions further refine the requirements based on the responses and result in the generation of the final code solution. To evaluate the effectiveness of ClarifyGPT, the study employs GPT-4 and ChatGPT for human assessments on two public benchmark tests, demonstrating significant improvements in code generation performance. Additionally, a high-fidelity simulated user response method is introduced to automatically evaluate ClarifyGPT across different LLMs and benchmark tests without requiring user participation. Experimental results indicate that ClarifyGPT achieved an average improvement of 11.52% across four benchmark tests compared to default GPT-4 and 15.07% compared to default ChatGPT.

In contrast to ClarifyGPT, which enhances code generation by addressing user requirement ambiguity through targeted questions, Yang *et al.* [66] propose TurduckenGen, which employs syntax-guided multitask learning methods. Addressing "Turducken-style" code generation, it effectively integrates syntactic knowledge into generated code by utilizing CodeT5 and multitask learning. This aims to resolve the challenge of LLMs generating code that does not adhere to the syntactic constraints of the target language. The study outlines three primary challenges concerning the absence of syntactic constraints: (1) efficient representation of syntactic constraints, (2) effective integration of syntactic information, and (3) scalable syntax-based decoding algorithms. TurduckenGen explicitly adds type information within code tokens to capture syntactic constraint representation. It then formalizes the code generation with syntactic constraint representation as an auxiliary task to enable the model to learn the code's syntactic constraints. Finally, it accurately selects code snippets conforming to the syntactic specifications using compiler feedback from multiple candidates. In comparisons against six state-of-the-art baseline methods on two Turducken-style code datasets, TurduckenGen demonstrates superior code generation in terms of code readability and semantic similarity.

Execution-feedback Code Generation.

Recently, inspired by the process of human programming, developers transitioned to the adoption of execution-feedback code generation. Such studies involve generating an initial program, running a series of example test cases, and iteratively refining the code based on the test results.

For example, Zhang *et al.* [65] introduce Self-Edit, drawing inspiration from the human programming process, to enhance code quality by utilizing the code execution results generated by LLMs. The generated code undergoes execution on given example test cases, and the execution results are included as supplementary comments within the code. Subsequently, a specialized code editor is used for further improvements and corrections. The goal is to improve the code generation quality of LLMs for competitive programming tasks. Self-Edit is evaluated on two competitive programming benchmark datasets (APPS and HumanEval) and nine popular LLMs of different parameter sizes. In comparison to directly generating code from LLMs, Self-Edit shows advantages in significantly enhancing the performance of LLMs, especially concerning the pass@1 metric. It also demonstrates the capability to generalize across different datasets. Compared to other post-processing methods, the "Self-Edit" method maintains a fixed sample budget and significantly reduces the computational cost of LLMs. Moreover, by directly modifying programs, this editor outperforms re-ranking methods, particularly with limited sample budgets.

Besides, Dong *et al.* [71] present a self-collaboration framework for code generation based on ChatGPT. They assemble a team consisting of three ChaGPT roles (*i.e.*,, analyst, coder and tester) and give them role instructions. Given a requirement, the analyst decomposes the requirement into several easy-to-solve subtasks and develops a high-level plan that outlines the major steps of the implementation. The coder receives plans from an analyst or test reports from a tester throughout the workflow. The coder has two responsibilities, *i.e.*, (1) writing code according to the plan provided by the analyst; and (2) refining code according to the feedback of test reports provided by the tester. The tester acquires the code authored by the coder and subsequently documents a test report. They give the role instruction that includes not only the role description (role and its responsibilities) but also the team description and user requirements, which will work together to initialize ChatGPT. They then perform a comprehensive evaluation on four code-generation benchmarks: MBPP, HumanEval, MBPP-ET and HumanEval-ET. Results show that the self-collaboration framework significantly enhances the performance of the base LLMs. Besides, ChatGPT (GPT-3.5) achieves the best performance across four code-generation benchmarks with a simple three-member team, surpassing even GPT-4. They also notice that the self-collaboration code generation yields more significant improvements on the datasets associated with extended test cases, namely HumanEval-ET and MBPP-ET. They conduct experiments to show the effectiveness of role-playing and interaction.

Besides, Ni *et al.* [68] propose Lever, which aims to improve the process of Text-to-Code generation by validating and ranking generated programs using execution results. It involves learning to verify the generated programs based on natural language input, the code itself, and its execution outcomes. Lever combines validation scores and LLM generation probabilities to re-rank the extracted programs and marginalizes programs with identical execution results. Across four datasets in table queries, math questions, and basic Python programming, Lever consistently improves over the baseline LLMs (using code-davinci-002) with improvements ranging between 4.6% and 10.9%, achieving state-of-the-art results across all datasets.

Empirical Evaluation of LLMs.

Many developers have conducted empirical evaluations of LLMs for code generation. To compare the attention of models with human attention, Kou *et al.* [69] conduct a study of five LLMs on a popular benchmark HumanEval to explore whether LLMs attend to the same parts of a natural language description as human programmers during code generation. They evaluate LLMs including InCoder, PolyCoder, CodeGen-Mono, CodeParrot and GPT-J. Results show that there is a consistent misalignment between LLM attention and programmer attention in all settings, indicating that LLMs do not reason programming tasks as human programmers. Besides, they find that LLMs attend to different kinds of keywords compared with programmers in code generation. Specifically, programmers are more sensitive to adjectives and numbers in task descriptions compared with CodeGen. Their further analysis shows that there is no statistical evidence showing that LLMs with higher code generation accuracy are more aligned with programmers. They suggest that SHAP (a perturbation-based method) produces the best attention alignment to human programmers among all methods and participants preferred the perturbation-based method over the gradient-based and the self-attention-based methods.

In 2023, Liu *et al.* [67] conduct experiments to evaluate the effectiveness of ChatGPT for two code generation tasks (*i.e.*,, text-to-code and code-to-code generation). The authors analyze basic prompt designs to guide ChatGPT's generation. ChatGPT, based on the GPT-3.5 architecture, produces human-like responses in accordance with user inputs (*i.e.*, prompts). However, ChatGPT's performance heavily relies on the quality of prompts, requiring better prompt designs, often referred to as prompt engineering. Subsequently, They assess ChatGPT's performance on the CodeXGlue dataset and introduce a prompt enhancement method called "chain-of-thought". Experimental results show that improving prompt design significantly enhances ChatGPT's generation performance. Additionally, they compare its performance with other state-of-the-art fine-tuned LLMs, analyzing the correctness and quality of the generated code. Finally, the paper summarizes the primary contributions of the research and releases a replicable research package for use in future explorations.

At the same time, Ouyang *et al.* [72] extensively investigates the threat of non-determinism in ChatGPT's code generation tasks. Through experiments conducted on three widely studied code generation benchmarks, it is demonstrated that there are significant discrepancies in correctness, test outputs, as well as syntax and structure among code candidates generated for identical instructions across different requests. The experimental results indicate a notable non-determinism in ChatGPT's code generation, particularly concerning semantic consistency, where code candidates generated for different requests exhibit substantial differences. In CodeContests, APPS, and HumanEval, the rates of tasks generating code with zero identical test outputs are 72.73%, 60.40%, and 65.85%, respectively. Among the three similarity metrics (semantic, syntactic, and structural), semantic similarity demonstrates the highest instability, with considerable variances in test pass rates, wherein approximately 60-70% of tasks yield code candidates with entirely different test outputs. The temperature setting affects ChatGPT's non-determinism, but setting the temperature to 0 does not completely eliminate it, while setting it to 2 may lead to uniformly subpar performance of generated code candidates. Additionally, through comparison with GPT-4, it is observed that GPT-4 exhibits slightly more inferior non-determinism than GPT-3.5. These results underscore the potential challenges of ChatGPT in code generation tasks, especially in ensuring consistency and correctness.

4.2.2 Code Search

Code search is an essential practice in software development where developers search for specific portions of source code within extensive codebases. This activity is undertaken to locate particular code sections, identify feature implementations, and discover code examples for potential reuse. In essence, code search encompasses a comprehensive set of methods, tools, and techniques employed to efficiently and effectively find, retrieve, and utilize source code within software projects.

Overall, existing LLMs in code search can be classified into three categories [168]. The first type employs various techniques to enhance the semantics of query texts for achieving a more precise search intent, such as CodeRetriever [73] and CodeTF [169]. The second type introduces or proposes more efficient training techniques, leading to better outcomes in training LLMs, such as CodeBERT [27], GraphCodeBERT [28], COSCO [74] and CCT [75]. A third category involves empirical research that explores the real-world performance of LLMs in the field of code search.

Enhance the semantics of the query text

Enhancing the semantics of the query text implies improving the understanding and representation of the query text in a way that captures its meaning more effectively. This involves enriching the semantic content of the text, making it more contextually aware, and aligning it with the underlying functions or concepts in the code.

In 2022, Li *et al.* [73] propose CodeRetriever, which learns semantic representations of function-level code by pre-training on a large-scale dataset of code-text pairs. It aims to address existing issues in code pre-training methods, focusing on acquiring function-level code representations for code search scenarios. CodeRetriever employs text encoders and code encoders to convert text and code into dense vectors. It refines the encoders through training with two contrastive losses: Unimodal Contrastive Learning encourages similar functional code to cluster closely in the representation space, while Bimodal Contrastive Learning assists in understanding the correlation between code and text. The model undergoes training on the CodeSearchNet dataset, which includes matched data of functions and their respective documents, alongside unmatched data comprising only functions. Compared with CodeBERT and GraphCodeBERT, Experimental results indicate that CodeRetriever notably enhances eleven domain/language-specific code search tasks, surpassing existing code pre-training models and spanning different code granularities across six programming languages (function-level, snippet-level, and statement-level). Particularly, it demonstrates strong performance in resource-scarce and cross-language code search tasks. These findings confirm the effectiveness and robustness of CodeRetriever, affirming the efficacy of both Unimodal and Bimodal Contrastive Learning.

Introduction or proposal of more efficient training techniques

The introduction or proposal of more efficient training techniques involves considering both static and dynamic features, leveraging both similar and dissimilar examples, and validating model performance across programming languages. The aim of introducing or proposing these techniques is to enhance the performance of LLMs in source code search tasks. This is achieved through innovative training methods and model architectures, exploring more effective encoding representation learning.

For example, Saieva *et al.* [74] introduce COSCO, an innovative code-to-code search technique that enhances the performance of LLMs by incorporating both static and dynamic features during training, and leveraging both similar and dissimilar examples. This method encodes dynamic runtime information during training, eliminating the need to execute searches on a corpus or query at inference time. Additionally, it simultaneously employs positive and negative reference samples during training. COSCO's effectiveness is being validated across multiple model architectures and programming languages. In cross-language code-to-code search tasks, it outperforms state-of-the-art cross-language search tools, exhibiting a performance improvement of up to 44.7%. Furthermore, compared with CodeBERT, ablation studies indicate a significant performance boost even when using only one positive and one negative reference sample during training, demonstrating the significance of both similar and dissimilar reference samples in code search. Notably, it is found that a carefully tuned smaller LLM can outperform larger proprietary LLMs, underscoring the value and importance of open-source models for the research community.

At the same time, Sorokin *et al.* [75] propose a novel training process known as Cross-Consistency Training (CCT) for training language models on source code in different programming languages. The CCT-LM model, initialized with GraphCodeBERT and fine-tuned with CCT, demonstrated the efficacy of the CCT technique in generating useful representations for code search. For code search, CCT uses the CodeSearchNet AdvTest dataset. The CCT-LM model, initialized with CCT, achieves an average Mean Reciprocal Rank (MRR) of 47.18% in the AdvTest code search benchmark.

Combined with the effectiveness of transfer learning, Salza *et al.* [76] try to explore how to apply the transformer architecture and transfer learning to code search. Thus, they propose a pre-trained BERT-based approach with fine-tuning. This approach contains two pre-trained BERT-based encoders, one for queries and one for code. For the query encoder, they use the Devlin *et al.* [5] pre-trained English model. For the code encoder, they pre-train it on the CodeSearchNet dataset. They apply the BERT model like pre-training on natural language and the next line prediction for pre-training on source code. They then mine and filter question-answer pairs from StackOverflow as the dataset for fine-tuning. Finally, they combine the two pre-trained encoders as the multimodal embedding model and fine-tune it. They apply 10-fold cross-validation and evaluate their approach on their collected dataset. Results show that the pre-trained model outperforms the non-pre-trained model. Besides, their approach outperforms an information retrieval model based on Lucene. They also discover that the combination of an information retrieval-based approach followed by a Transformer achieves the best results.

4.2.3 *Code Translation*

Code translation has been extensively studied for natural languages, but has received little attention in the context of programming languages. code translation for programming languages refers to the use of data from a source language to improve the performance of a model on a target language. transfer learning with LLMs improved the performance of various software engineering tasks, including code summarization, code search, clone detection, code refinement and so on.

In 2023, Zhang *et al.* [77] design Codeditor, an LLM proposed for the task of cross-lingual code editing, incorporating a Transformer-based encoder-decoder framework. Codeditor's parameters are initialized using CoditT5. Subsequently, to adapt to multi-language collaborative editing tasks, Codeditor is fine-tuned, focusing on the model's input context and output format. For training and evaluation, Zhang *et al.* extract the first dataset containing paired code changes from 8 open-source Java projects and their corresponding C# projects' commit histories on GitHub. This dataset encompasses parallel code modifications in two programming languages. Evaluation occurs in two directions: updating C# methods based on Java modifications (source language: Java, target language: C#), and updating Java methods based on C# modifications (source language: C#, target language: Java). The experimental results demonstrate that Codeditor outperforms all existing models across all selected automatic metrics. This includes fine-tuned LLMs like Codex under few-shot settings and ChatGPT under zero-shot settings. In the task of updating C# code based on Java modifications, Codeditor achieves a CodeBLEU score of 96 (out of 100), exceeding by over 25% the performance of LLMs fine-tuned specifically for this task. Furthermore, Codeditor and generative models complement each other: Codeditor excels in updating longer code snippets, while generative models perform better with shorter ones. By combining these two models, based on the input code's size, Codeditor's accuracy in exact matching is further enhanced by 6%.

At the same time, Baltaji *et al.* [78] extensively explore code translation learning using transformer-based LLMs across 41 programming languages in four tasks (classification and generation tasks). They conduct experiments using six common source languages (Javascript, C, Kotlin, Java, C++, and Python) to identify the best source language. Baltaji *et al.* calculate the average scores for each language in every task across all target languages and rank these languages based on the average scores. Their experimental framework is based on CodeT5-base (220M) and open-source datasets CodeNet and XCodeEval, incorporating 58 fine-tuned models. The experimental results indicate that cross-lingual learning outperforms LLM zero-shot learning significantly. In this context, C++ and Python are comparatively less ideal as either source or target languages, while Kotlin exhibits the best transfer results, serving as a strong source language for multiple target languages and tasks.

4.2.4 *Code Comment Completion*

Code comment completion refers to the process by which a computer program or model automatically writes code comments for programmers. In this process, based on the code that has already been written, the program or model automatically generates corresponding comments to explain the function and purpose of the code. It aims to provide real-time suggestions and assistance to programmers while writing code comments, similar to the concept of code auto-completion. This process does not create comments from scratch but rather assists programmers in completing comments more quickly based on the partial comments or code snippets they input. This autocomplete capability helps reduce repetitive work during code writing, improves the quality of code documentation, and offers better code readability.

In 2021, Mastropaolo *et al.* [79]address the issue of code comments using T5 and n-gram models. The study compares a simple n-gram model and the T5 model in supporting code comment completion. The findings indicate that the T5 model performs better, although the n-gram model remains competitive. The research experiments with a dataset containing a large number of Java methods and their associated comments. Results show that, firstly, the T5 model outperforms the n-gram model in all the tested code comment completion scenarios. Secondly, despite the T5 model's higher performance, it requires developers to have already written a portion of the comment (similar to the n-gram model) and to provide the relevant code context associated with the comment. Lastly, the simplicity of the n-gram model and its wider applicability suggest potential complementarity between the two models in implementing a code comment completion tool.

4.2.5 *Code Summarization*

Code summarization takes as input a code snippet provided by the developer and automatically generates natural language summaries for it.

Code summarization is the technology that employs computing systems to provide summaries of code snippets, typically presented in natural language form. These summaries are usually utilized to enhance the comprehension of software systems, thereby facilitating their maintenance. The output of code summarization can be used to offer a higher-level explanation of software systems. Combining LLMs with code summarization can provide more accurate and natural summaries of source code, as LLMs can leverage their training on vast amounts of textual data to more accurately infer and generate natural language summaries of source code.

For example, Rukmono *et al.* [80] propose a method that combines LLMs, code summarization, and achievements in the field of static code analysis to generate component-level summaries of software systems. Particularly, they address the unreliability of LLMs in reasoning by applying a ”chain-of-thought prompting” strategy. This method demonstrates its feasibility through a bottom-up approach, starting with the comprehension of low-level software abstractions (such as functions) and then composing these findings to comprehend higher-level abstractions (such as classes and components). They demonstrate this method for generating software component-level summaries on the JHotDraw 5.1 project. They establish the abstraction levels as operations, modules, and components. Leveraging GPT-3.5 model and utilizing the Chat Completion API, they execute the summarization process. In the operation-level summaries, the model is prompted to provide brief documentation comments. In the class-level summaries, the model is instructed to generate documentation comments for classes and organize methods into groups.

Besides, Sun *et al.* [170] focus on evaluating the performance of ChatGPT on code summarization and comparing it with several code summarization models (*i.e.*,, NCS, CodeBERT, and CodeT5). They first design an appropriate prompt to guide ChatGPT to generate in-distribution comments. Then, they use such a prompt to ask ChatGPT to generate comments for all code snippets in the CSN-Python dataset. They apply three metrics: BLEU, METEOR, and ROUGE-L. Results show that ChatGPT obtains the lowest scores of 10.28 in BLEU and 20.81 in ROUGE-L, worse than all three baselines. They also find that comments generated by CodeT5 are significantly better than that generated by ChatGPT. Their further analysis shows that ChatGPT can better understand the inner meaning of the code without being interfered by the vocabulary of the code. Besides, ChatGPT tends to describe more detailed behavior of the code in the comments generated. However, they notice that compared with CodeT5, ChatGPT is less sensitive to the features or keywords of a programming language and ChatGPT still has some difficulty understanding the deep logic of the code.

4.2.6 *Code Completion*

Code completion aims at speeding up code writing by predicting the next code token(s) the developer is likely to write. Researchers usually focus on improving the accuracy of the generated predictions.

In 2023, Nie *et al.* [86] present a deep learning LLM TeCo, which leverages the context of the prior statement and the code under test to generate the next statement in a test method. They leverage the pre-trained CodeT5 model for test completion tasks. They first collect a Java dataset from CodeSearchNet with some filtering procedure for fine-tuning. Then, they extract six kinds of code semantics from the source code, including three types of execution results and three types of execution context. For generated plausible outputs, they propose to use reranking by test execution to improve the quality of the generated statements. They evaluate fine-tuned CodeT5 on the collected dataset. Results show that TECO significantly outperforms all baseline models (inluding CodeGPT and CodeT5) on all automatic metrics (including top-10 accuracy, BLEU and ROGUE). Besides, TECO can generate runnable statements for 28.63% of the time, and compilable statements for 76.22% of the time, much higher than the best baseline model. They conclude that using code semantics and reranking by execution can greatly improve the deep learning model’s performance on test completion. Their further analysis shows that different kinds of code semantics provide complementary information for test completion.

At the same time, Li *et al.* [84] introduce the CCTEST framework, aiming to test and enhance code completion systems. The framework utilizes a series of program structure-consistent mutation strategies to generate mutated code snippets with similar or identical program structures. Subsequently, it identifies anomalies in the code completion outputs and selects the output closest to the ”average” appearance to enhance the completed code. CCTEST operates as a black-box for code completion systems, requiring no assumptions about the specific implementation details of the systems. The study tests eight widely-used LLM code completion systems, including the commercial tool Copilot and seven other systems maintained by non-profit organizations (such as CodeParrot) or industrial companies (such as EleutherAI’s GPT-Neo and Salesforce’s Codegen). The results reveal 33,540 programs triggering errors in code completion, and the enhancement method significantly improves the accuracy of code completion systems.

In addition to the aforementioned academic-level work, there are also some industrial-grade products widely applied in IDEs, such as GitHub Copilot, OpenAI’s Codex, Tabnine, and Jurassic-1, which are LLMs. These LLMs leverage vast amounts of unstructured text, including websites, books, and open-source code, to generate code completion suggestions. Improvements related to code completion systems encompass both the security of these systems and the testing and enhancement of code completion systems.

Empirical Evaluation of LLMs. There are also many empirical studies now analyzing the actual performance of LLMs in code completion.

For example, Ciniselli *et al.* [85] demonstrate the exploration of BERT models’ capabilities for code completion at various granularity levels. They primarily utilize the RoBERTa model for code completion, training a BPE model to address the out-of-vocabulary issue. The study involves fine-tuning using Java and Android datasets obtained from

CodeSearchNet and the AndroidTimeMachine dataset on GitHub. Subsequently, three versions of each dataset are created by employing different masking techniques-token masking, construct masking, and block masking. After obtaining twelve sets of (training, evaluation, test) data, they train twelve distinct RoBERTa models, comparing their performance against each other and against an n-gram model. The results reveal that RoBERTa can accurately predict only ”small” masked blocks, exhibiting better performance in the Android dataset compared to the Java dataset. Abstracted code outperforms raw code, and RoBERTa demonstrates superior performance across all experimented datasets and code representations over the n-gram model.

In another study, Ciniselli *et al.* [82] conduct a large-scale exploration of Transformer models' code completion capabilities at different granularity levels. They primarily employ three models—RoBERTa, T5-small, and n-gram—and train a BPE model for each to handle out-of-vocabulary concerns. The study involves pre-training data collection from Java repositories on GitHub and fine-tuning using datasets from CodeSearchNet and AndroidTimeMachine for Java and Android datasets. By employing various masking processes, three versions are created—token masking, construct masking, and block masking. Ultimately, the models are evaluated on six datasets, showcasing T5's superior performance to RoBERTa in Java and Android datasets, while the n-gram model displays competitive results against RoBERTa. Furthermore, it is noted that the deep learning-based models fail to generate accurate predictions as the number of masked tokens increases, while the T5 model delivers satisfactory results. The study also highlights the positive impact of pre-training and multi-task fine-tuning on the T5 model's performance.

Besides, Van *et al.* [83] analyze the effects of three recent LLMs (UniXcoder, CodeGPT, and InCoder) on code completion, evaluating at two levels of granularity: token and line completion. They investigate the impact of type annotations and comments as common forms of additional contextual information that often aids developers in better understanding code. The study aims to explore whether using contextual data to make code more understandable improves the performance of LLMs in the task of code completion. The experimental results show that different types of contextual information significantly impact the performance of code completion models. Removing type annotations contributes to enhancing model performance, while multi-line comments positively affect the performance. Preserving all comments or only multi-line comments also boosts performance. However, doc-block comments have a minor impact on performance improvement, and single-line comments exhibit relatively weaker effects, varying across different models. This emphasizes the potential significance of providing diverse contextual information for enhancing the performance of code completion models.

4.2.7 Program Synthesis

Program synthesis refers to the automated process of generating computer programs based on high-level specifications or requirements. The objective of this process is to automate typically complex and time-consuming manual software development tasks by allowing machines to generate code based on specifications provided by users. The emphasis of program synthesis is to enhance the capabilities of LLMs, such as GPT-3 and Codex, enabling them to generate code from natural language specifications of programmer intent. This is achieved by introducing post-processing steps based on program analysis and synthesis techniques to address the limitations of LLMs in understanding program syntax and semantics.

For example, Jain *et al.* [87] discuss the development of LLMs such as GPT-3, Codex, and Google's LLM in generating code from natural language descriptions of programmer intent. To address the issue that LLMs do not understand program semantics and cannot provide guarantees about the quality of suggested code, the authors propose augmenting LLMs with program analysis and synthesis techniques to understand program syntax and semantics. They build and evaluate Jigsaw, aimed at synthesizing code for Python Pandas API using multi-modal inputs. Jigsaw is a multi-modal program synthesis system that accepts natural language strings and test cases or input-output examples as input and generates code snippets as output. Its architecture comprises pre-processing and post-processing modules. The pre-processing module translates natural language intent into a customized query, which is then sent to LLMs. The post-processing module conducts syntactic and semantic checks and performs transformations on the code generated by LLMs to ensure it passes provided test cases and other quality checks. The authors elaborate on common error types such as referencing errors, argument errors, and semantic errors, and how Jigsaw corrects them. Jigsaw can also learn from user feedback to enhance the quality of its pre-processing and post-processing modules. The authors evaluate Jigsaw's performance by creating two datasets for Python Pandas API and demonstrate its performance compared to other baseline and state-of-the-art code synthesis frameworks on these datasets.

To demonstrate the ongoing advancements in the application of LLMs in automatic coding, Liventsev *et al.* [88] propose to apply a Synthesize, Execute, Debug (SED) approach to LLMs to achieve automatic programming. Thus, they present a framework Synthesize, xecute, Instruct, Debug and Rank (SEIDR) which leverages Codex-edit and GPT-3 as the LLMs. In the Synthesize step, they apply a basic template to generate initial draft solutions to programming tasks to be repaired in the later stages of SEIDR. In the Execute step, the programs are compiled and launched using the standard tools for the programming language. In the Instruct step, they generate instructions using template engines that replace placeholders in files or strings with input values and return a formatted string. They design two different instruction generation methods: static and LLM. The static method uses a fixed input template and substitutes placeholders for input and output with the corresponding strings of the first failing test case. The LLM method uses the failing I/O pair in the LLM for text completion, thereby prompting the GPT-3 to produce the bug summary. Then in the Debug step, the Codex-edit generates multiple candidate solutions. Finally, the Rank step ranks all programs in the candidate population by their score calculated in EXECUTE. They then evaluate SEDIR on the PSB2 benchmark. Results show that SEIDR outperforms the

PushGP baseline in Python and performs on par with it in C++ experiments. They also find that program synthesis in C++ with SEIDR achieves better performance in the repair-only setting with both GPT-assisted prompts that summarize bugs in code and static templates which describe failing I/O cases. The best-performing C++ instruction is obtained with GPT-3 for text completion that contains the word "obviously". Overall, SEIDR performance is stable with different debugging prompts submitted to Codex-edit.

4.2.8 Code Edits Prediction

Code edit prediction refers to the task of predicting changes or modifications that need to be made to a piece of code to transition from one version to another. It involves predicting the edits a developer will make to refactor code from one version (e.g., $\mathbf{v_1}$) to another version (e.g., $\mathbf{v_2}$ or $\mathbf{v_3}$). This task is essential in software development, especially during code refactoring or feature additions.

For example, Gupta *et al.* [89] propose Grace to address the challenges of code editing by leveraging the generative capabilities of pre-trained LLMs on previously related edits. In experimental validations with codex and CodeT5, Grace significantly enhances their ability to generate correct edited code in zero-shot and fine-tuning settings, achieving performance improvements of 29% and 54%, respectively, compared to current state-of-the-art symbolic and neural methods. This approach holds significant importance for addressing practical problems in predicting code edits in software engineering and opens up new possibilities for exploring the use of LLMs in solving other software engineering challenges in the future.

4.3 Software Testing

Software testing is the process of executing a program or system with the intent of finding errors, or any activity aimed at evaluating an attribute or capability of a program or system to ensure it meets its required results.

4.3.1 Fault Localization

Fault localization is a critical process in software engineering that involves identifying the specific locations or elements in a software system where faults or bugs are present. By pinpointing the exact location of a fault, developers can more quickly perform software debugging and bug-fixing, leading to more efficient software development and maintenance. In the literature, researchers have conducted several studies to explore the performance of LLMs in fault localization, where two types of LLMs are involved, *i.e.,* BERT-like and GPT-like models. Table 6 presents a summary of existing LLM-based fault localizations. In the following, we discuss these LLM-based fault localization techniques according to the types of employed models.

BERT-based Localization. As early as 2021, Zhu *et al.* [94] propose TroBo, a CodeBERT-based cross-project bug localization approach by leveraging both bug reports and source code. TroBo first extracts the summary and description parts of bug reports, which are considered as a sequence of words and represented as word embedding by CodeBERT. TroBo then treats source code as sequences of code tokens and establishes a dedicated project-aware classifier to enhance the embedding layer's sensitivity to different projects. Besides, to learn the correlation patterns between bug reports and related source code, TroBo constructs a shared relevance predictor based on distance calculations between embeddings.

In 2022, Ciborowska *et al.* [90] discuss how to optimize BERT for changeset-based bug localization. The author explores various design choices for applying BERT, including how to encode code changes and match error reports to specific code changes to enhance accuracy. The study compares their BERT-based model to a non-contextual vector space model and existing BERT-based architectures in software engineering, showing significant advantages, particularly for error reports that do not provide clear indications about related code elements. Furthermore, the research conducts experimental evaluations through two research questions (RQ1 and RQ2). RQ1 focuses on comparing the effectiveness of FBL-BERT with other techniques, while RQ2 investigates different code change encoding strategies. The findings from these questions provide strong support for applying BERT to error localization in software engineering and hold promise for significant advancements in practical applications.

To address the limitation of time and accuracy of previous approaches, in 2023, Mohsen *et al.* [91] propose a three-stage-based bug localization approach based on BERT. First, the raw data preparation stage processes information from bug reports and source code, such as summary, description, and stack trace. Second, the package classification stage aims to identify software packages that might contain the source code requiring modification. Third, the source code recommendation stage returns a list of source code files sorted in descending order based on a text similarity calculated by BERT.

GPT-based Localization. Wu *et al.* [92] conduct a comprehensive empirical study on the large-scale open-source program Defects4J, evaluating the potential of LLMs (ChatGPT-3.5 and ChatGPT-4) in fault localization research. Experimental results indicate that, with function-level context and supplemented by test cases and error logs, ChatGPT-4 achieves an average score of 23.13 under the TOP-1 evaluation metric, outperforming the existing SmartFL baseline by an average of 46.9%. The study assesses the consistency of ChatGPT in fault localization. Findings reveal that incorporating dynamic execution information into ChatGPT leads to more consistent fault localization results, reducing the average variance to 12.00% in the TOP-1 metric, the lowest among ChatGPT-related methods. Furthermore, excluding any component leads to a decline in ChatGPT-4's performance as measured by the TOP-1 metric. Among these, excluding the error log has the most significant impact, resulting in a 25.6% decrease in accuracy. To address concerns about potential overfitting of ChatGPT to Defects4J, Wu *et al.* introduce a new, more recent dataset named "StuDefects." An extended analysis of this new dataset aligns with observations from Defects4J, showing over 52.9% superiority in the TOP-1 metric compared to existing techniques.

Besides, Kang *et al.* [93] introduce AutoFL, enabling it to explore select parts of the code repository and apply a post-processing step to match the language model's answers with

TABLE 6
A summary and comparison of existing LLM-based fault localization studies

Year	Study	LLMs	Type	Repository
2022	Ciborowska *et al.* [90]	BERT	BERT-based Localization	https://anonymous.4open.science/r/fbl-bert-700C
2023	Mohsen *et al.* [91]	BERT	BERT-based Localization	N.A.
2023	Wu *et al.* [92]	GPT-4	GPT-based Localization	N.A.
2023	Kang *et al.* [93]	ChatGPT	GPT-based Localization	N.A.
2021	Zhu *et al.* [94]	CodeBERT	BERT-based Localization	N.A.

actual code elements. This significantly reduces the requirement for prompts and enables the language model to effectively locate faults. Specifically, given a test and its failed stack, AutoFL allows the language model to browse the source code, invoke functions that return coverage classes, coverage functions, and any available Javadoc information for those covered functions. After the prompt, the language model calls the function sequences to gather pertinent details and deduce the manner in which the error occurred. The method ultimately outputs the most probable method responsible for the error, considered as the fault localization result. They formulate a post-processing step, collecting outputs from multiple language models, processing these results to align text predictions with existing code elements, thereby enhancing ranking performance. AutoFL undergoes evaluation on the Defects4J debugging benchmark. In 353 instances, AutoFL successfully identifies the actual error position as the most defective position in 149 cases, solely based on the failed test. Further comparison against the next best-performing independent technique (Spectrum-based Fault Localization) shows that AutoFL's performance exceeds Spectrum-based FL by 338% when only failed tests are used. The experimental outcomes indicate that when appropriately employed, LLMs can serve as effective fault localizers.

4.3.2 Code Decompilation

Decompilation refers to the reverse engineering process aimed at extracting a binary executable's code into a form that closely resembles its original source code and is comprehensible to humans. Within this process, one of the primary challenges is the recovery of variable names. Research in this field explores methods that leverage the combination of LLMs and program analysis to enhance the recovery of variable names. This procedure finds critical applications in security fields (such as malware analysis, vulnerability detection) and software engineering (including code reuse and software supply chain analysis).

To address a crucial issue in decompilation – the recovery of variable names within binary executable files, Xu *et al.* [95] integrate the training of ChatGPT on extensive corpora of natural and programming languages. They introduce an innovative name recovery technique that combines pre-trained LLMs with program analysis. By employing an iterative algorithm that combines LLM queries and program analysis, they progressively enhance the performance of LLM in the name recovery process. They implement a prototype system named LmPa and evaluate it on 1258 functions across six prominent binary analysis benchmarks. Experimental outcomes demonstrate that 75% of the names recovered by LmPa are considered satisfactory, whereas the state-of-the-art name recovery method DIRTY [171] only achieves 6%. According to an automated metric based on name similarity, LmPa exhibits substantially higher precision and recall, standing at 33.85% and 31.12%, respectively, in contrast to DIRTY's 17.31% and 10.89%. On average, it takes eight LLM queries to recover variables within a function. The total cost for the experiments is a mere $30. Ablation studies reveal that conducting LLM queries without program analysis would lead to decreased precision and recall, at 31.04% and 18.21% respectively. They employ an iterative algorithm to progressively enhance the performance of LLM in the name recovery process, whereas Wong *et al.*.'s research also adopts an iterative design to gradually improve the decompiled output, making it compatible with standard C/C++ compilation

Besides, Wong *et al.* [96] introduces DecGPT, a method for automatically repairing decompiled outputs to achieve recompilability. It follows a two-stage, iterative process using a LLM to fix syntax, inference, and memory errors in decompiled output, making it compatible for standard C/C++ compilation. Through static and dynamic repair steps, DecGPT employs LLM model interactions with LLM-Turbo, gradually improving the decompiled output to eventually allow successful executable file generation by the compiler. DecGPT is primarily applied to C/C++ decompilers, using IDA-Pro as an example. The model selected GPT-3.5 instead of GPT-4 to reduce costs and enhance processing speed. The authors utilize the Code Contest dataset, comprising program submissions from five online assessment platforms, to validate the correctness of repairs. The main evaluation metric is the decompilation success chain length, assessed with values of C = 1, 5, 10, and 15. Experimental results show that compared to DecRule and methods solely based on LLMs, DecGPT shows superior performance in automatically rectifying decompiler output errors, significantly improving the success rate of repairs. DecGPT exhibits a 75% success rate in addressing regularized repair challenges, whereas DecRule achieves only 8%. Additionally, compared to using only LLMs, DecGPT achieves approximately 30% higher repair rates. Its iterative design effectively enhances repair efficiency for longer context inputs, although it still encounters challenges in rectifying some memory errors. Overall, DecGPT holds significant promise for automatically rectifying decompiled outputs, yet specific functional and memory error repairs remain challenging.

4.3.3 Vulnerability Detection

Automated vulnerability detection (AVD), also referred to as vulnerability prediction, aims to identify potential security

TABLE 7
A summary and comparison of existing LLM-bssed vulnerability detection studies

Year	Study	LLMs	Type	Repository
2021	Fu *et al.* [97]	CodeBERT	Fine-tuning Dectection	https://github.com/awsm-research/LineVul
2022	Hanif *et al.* [99]	RoBERTa	Fine-tuning Dectection	https://github.com/ICL-ml4csec/VulBERTa
2023	Steenhoek *et al.* [98]	CodeBERT, VulBERTa, PLBART, LineVul	Empirical Study	https://doi.org/10.6084/m9.figshare.20791240
2023	Ahmad *et al.* [101]	Codex, GPT-3.5	Zero-shot Dectection	https://zenodo.org/records/8012211
2023	Noever *et al.* [102]	GPT-4	Empirical Study	N.A.

bugs in software systems. AVD is critical for protecting security-critical software systems from malicious attacks, providing the foundation for timely patching reported security vulnerabilities before they may be exploited (discussed in Section 4.4.2). In the literature, learning-based AVD approaches [172], [173] have been proposed to detect security vulnerabilities by extracting meaningful features and performing predictions automatically. For example, Li *et al.* [173] propose IVDetect, to perform fine-grained vulnerability prediction based on a FA-GCN model and GNNExplainer. However, such learning-based AVD approaches are limited by the amount of training data, resulting in capturing a suboptimal vector representation of source code. Table 7 presents a summary of existing vulnerability detection studies with LLMs, which are discussed in detail as follows.

①**Fine-tuning LLM-based Detection.**

To address the above-mentioned issue of learning-based AVD, in 2022, Fu *et al.* [97] present LineVul, a Transformer-based line-level vulnerability prediction approach based on CodeBERT. LineVul offers three key enhancements over IVDetect. Firstly, IVDetect's training process is constrained by project-specific datasets while LineVul employs a CodeBERT pre-trained language model to generate vector representations of source code. Secondly, IVDetect's RNN-based architecture falls short in handling long-term dependencies and capturing the semantics of buggy code effectively. In contrast, LineVul utilizes a BERT architecture with self-attention layers, powered by dot-product operations, to address these challenges when dealing with lengthy code sequences. Lastly, IVDetect's sub-graph interpretation remains relatively coarse-grained. LineVul takes a step forward by harnessing the attention mechanism of the BERT architecture to pinpoint vulnerable lines, offering a finer-grained solution compared to IVDetect. The experimental results on a large-scale real-world C/C++ dataset with 188k+ C/C++ functions demonstrate that LineVul outperforms previous learning-based AVD approaches, such as IVDetect [173], on both function-level and line-level predictions. In additional to CodeBERT, BERT is leveraged in vulnerability detection by fine-tuning on domain datasets [100].

In parallel to LineVul, Hanif *et al.* [99] propose VulBERTa, an encoder-only Transformer-based AVD approach based on RoBERTa model architecture. VulBERTa is first pre-trained with the MLM objective of learning an informative general representation of C/C++ code across different software projects. VulBERTa is then fine-tuned to support the vulnerability detection task across multiple vulnerability detection datasets. The experimental results on binary and multi-class vulnerability detection tasks across several datasets show that VulBERTa achieves state-of-the-art performance, outperforming existing AVD approaches across different datasets.

②**Zero-shot LLM-based Detection**

In 2023, Ahmad *et al.* [101] propose FLAG, utilizing LLMs for automated code review to aid in identifying defects and anomalies within code. FLAG is based on the lexical capabilities of LLMs, taking an input code file, extracting and regenerating each line within it for self-comparison. By comparing the original code with LLM-generated alternatives, notable differences are flagged as anomalies for further inspection, supported by features such as distance from comments and LLM confidence to assist in classification. Unlike other automated methods in this domain, FLAG is language-agnostic, capable of handling incomplete or non-compiling code without the need for creating security properties, functional tests, or defining rules. The study explores the features of LLMs in this classification and evaluates FLAG's performance on known defects. The authors conduct experiments using two state-of-the-art LLMs, OpenAI's code-davinci-002 and gpt-3.5-turbo, across 121 benchmarks in C, Python, and Verilog. The results demonstrate that FLAG can identify 101 defects and reduce the search space to 12-17% of the source code. Furthermore, the study finds that gpt-3.5-turbo exhibits better defect detection capabilities but with a higher false positive rate (FPR). FLAG slightly outperforms in functional defects compared to security-related defects and exhibits performance differences across different programming languages.

③**Empirical Evaluation of LLM-based Detection.**

In 2023, Steenhoek *et al.* [98] conduct an empirical study to investigate the performance of learning models in detecting software vulnerabilities. They survey and reproduce nine learning-based AVD approaches, including the above-mentioned two LLM-based approaches, *i.e.,* LineVul and VulBERTa, and two off-the-shelf LLMs, *i.e.,* CodeBERT and PLBART. The experiments are conducted from three aspects, *i.e.,* model capabilities, training data, and model interpretation on two vulnerability detection datasets, *i.e.,* Devign and MSR. This research provides valuable insights into the current capabilities and performance of LLMs in the realm of vulnerability detection.

④**Exploration of ChatGPT.**

In 2023, Zhang *et al.* [103] explore the performance of ChatGPT in software vulnerability detection. Researchers aim to evaluate the efficacy of ChatGPT in identifying software vulnerabilities by designing various prompts. Leveraging ChatGPT's multi-turn conversational memory, the study integrates the structural and sequential information

TABLE 8
A summary and comparison of LLM-based test generation studies

Year	Study	LLMs	Type	Repository
2020	Tufano *et al.* [108]	BART	Fine-tuning Generation	https://github.com/microsoft/methods2test
2023	Schafer *et al.* [106]	GPT-3.5	Zero-shot Generation	https://doi.org/10.6084/m9.figshare.23653371
2023	Dakhel *et al.* [174]	Codex, Llama-2	Mutation-guided Generation	https://github.com/ExpertiseModel/MuTAP
2023	Xie *et al.* [107]	ChatGPT	Conversation-driven Generation	https://github.com/ZJU-ACES-ISE/ChatUniTest
2023	Tang *et al.* [109]	ChatGPT	Empirical Study	https://sites.google.com/view/chatgpt-sbst
2023	Lemieux *et al.* [110]	Codex	Empirical Study	https://github.com/microsoft/codamosa

of source code into prompts to enhance ChatGPT's capability in identifying vulnerabilities. Additionally, extensive experiments on two vulnerability datasets showcase the effectiveness of using ChatGPT for enhanced prompt-based vulnerability detection. The paper presents an analysis of the pros and cons of using ChatGPT for software vulnerability detection. RQ1 explains that ChatGPT surpasses baseline models when using basic prompts, performing better in Java programs but having limited comprehension in C/C++ programs. ChatGPT is employed for software vulnerability detection. RQ2 indicates that the inclusion of API calls aids vulnerability detection in Java programs, while data flow information contributes slightly in C/C++ programs. RQ3 demonstrates that chain-of-thought prompts perform excellently in the C/C++ dataset but poorly in the Java dataset. RQ4 emphasizes the significance of prompt placement in vulnerability detection, suggesting that positioning API calls before the code and data flow information after the code enhances ChatGPT's accuracy. Finally, RQ5 summarizes ChatGPT's excellent performance in specific types of vulnerabilities, particularly in grammar-related or boundary-related types, but with lower performance in contexts that demand deeper comprehension.

Meanwhile, Noever *et al.* [102] evaluate the capabilities of LLMs, particularly GPT-4, in detecting software vulnerabilities, comparing their performance against traditional static code analysis tools like Snyk and Fortify. The analysis covers numerous repositories, including those from NASA and the Department of Defense. GPT-4 identifies approximately four times the number of vulnerabilities compared to its counterparts. Furthermore, it provides viable fixes for each vulnerability, demonstrating a low rate of false positives. The tests encompass 129 code samples across eight programming languages, revealing the most severe vulnerabilities in PHP and JavaScript. GPT-4's code corrections result in a 90% reduction in vulnerabilities, requiring only an 11% increase in code lines. LLMs exhibit the ability to self-audit, offering repair suggestions for identified vulnerabilities and emphasizing their precision. Future research should explore system-level vulnerabilities and integrate multiple static code analysis tools to comprehensively understand the potential of LLMs.

4.3.4 *Unit Test Generation*

Test generation is the process of creating a set of test data or test cases for testing the adequacy of new or revised software programs. Unit test generation is a specialized domain within test generation, focusing on creating test cases for individual code units. Table 8 presents existing test generation techniques equipped with LLMs. We summarize these advanced studies that employ LLMs as follows.

①**Fine-tuning LLM-based Generation.** As early as 2020, Tufano *et al.* [108] propose AthenaTest, an approach that aims to generate unit test cases by leveraging the BART model. They first perform semi-supervised pre-training on a large corpus of English text to help BART learn the semantic and statistical properties of natural language. They then pre-train BART on abundant Java source code crawled from GitHub with a similar pre-training strategy to English pre-training. After that, they fine-tune it on a Java dataset with different granularity of focal context. By fine-tuning previous BART models, they finally obtain four different models: BART_Scratch, BART_English, BART_Code and BART_English+Code. They then compare their approach with EvoSuite and GPT-3 in terms of code coverage and explore the impact of different pre-trainings on the assert generation performances. Results show that pre-training on both English and source code has a significant positive effect on the task of generating Test Cases and the model BART_English+Code achieves the best validation loss. Besides, their approach generates test cases that accurately test the focal methods and obtain comparable test coverage w.r.t. EvoSuite, as well as outperforming GPT-3. They also find that focal context improves the performance of the model. Their further analysis shows that AthenaTest generates syntactically correct test cases that conform to the Unit Test Case standards and it is able to generate correct test cases for different defects4j projects.

②**Zero-shot LLM-based Generation.** Schäfer *et al.* [106] present an LLM-based end-to-end adaptive test generation technique TestPilot for JavaScript. TestPilot consists of five main components: API explorer, documentation miner, prompt generator, test validator and prompt refiner. The API explorer takes the package under test (PUT) as input and outputs a list of functions described by their access paths, signatures, and definitions. The documentation miner extracts code snippets and comments from documentation included with the PUT and associates them with the API functions they pertain to. The remaining three components work together to generate and validate tests for all API functions identified by the API explorer, using the information provided by the documentation miner. The prompt generator constructs the initial prompt and then these prompts are sent to the LLM. They leverage the off-the-shelf Codex model of Cushman-size and it generates candidate tests. The test validator tries to fix simple syntactic errors and then parses the resulting code to check whether it is syntactically valid. If not, the test is immediately marked as failed. Otherwise, it is run using the Mocha test runner to determine whether it passes or fails. Finally, the prompt refiner applies a number of strategies to generate additional prompts to use for querying the model. They mainly apply four strategies: 1) Include the definition of a function if the prompt does not contain it. 2) Include the doc comment if the prompt does not contain it. 3) Include the usage snippets if the prompt does not contain them. 4) Include the text of the failing test followed by a comment if a test failed. They evaluate TestPilot on 25 npm packages from diverse domains and use Istanbul/nyc to measure statement coverage. Results show that the generated tests achieve up to 93.1% statement coverage. Besides, they discover that re-prompting the model with the error message of failing tests allows TESTPILOT to produce a consequent passing test in 16.3% of the cases. They also notice that a refiner always improves the results and trying all combinations of the various prompt components is the best for performance improvement.

③**Mutation-guided Generation.** Building on the progress, Dakhel *et al.* [174] propose MuTAP, which enhances the quality of test cases using LLMs, particularly in detecting software bugs. It guides LLMs in generating test cases by combining zero-shot and few-shot learning techniques and assesses their effectiveness through Mutation Testing on the generated test cases. MuTAP also involves steps for syntax fixing and intended behavior repair to ensure the generated test cases are syntactically correct and reflect the expected behavior of the program. If the generated test cases fail to entirely kill all mutants, MuTAP introduces surviving mutants to prompt LLMs to generate more effective test cases. Finally, MuTAP incorporates an Oracle Minimization step utilizing a greedy algorithm to reduce redundant assertions, aiming to enhance the effectiveness of test cases by minimizing redundant assertions and thereby improving their efficiency.

④**Conversation-driven Generation with ChatGPT.** In 2023, ChatGPT is also being applied in unit test generation. Xie *et al.* [107] propose ChatUniTest, an automated unit test generation tool based on ChatGPT, falling under the "Generation-Validation-Repair" framework. It parses the project, extracts vital information, and creates an adaptive focal context, including the focal method and its dependencies within the predefined maximum prompt token limit. This context is incorporated into a prompt and then submitted to ChatGPT. Upon receiving a response from ChatGPT, ChatUniTest extracts the raw test from the response. Subsequently, it validates the test and employs rule-based repair to fix syntax and simple compile errors, followed by using ChatGPT for addressing challenging errors. In rigorous evaluations, ChatUniTest excels, surpassing EvoSuite in branch and line coverage and outperforming AthenaTest and A3Test in focal method coverage. ChatUniTest effectively generates assertions while utilizing mock objects and reflection to achieve test objectives.

⑤**Empirical Evaluations of LLM-based Test Generation.** To enhance test coverage and improve the efficiency and effectiveness of automated testing, Lemieux *et al.* [110] propose to leverage LLMs for SBST. They present CODAMOSA which integrates Codex with SBST to generate Python test cases. In each mutation iteration, CODAMOSA keeps track of the coverage achieved by the current archive and keeps a copy of the current population. If the coverage stall length does not surpass the 'maxStallLen' they set, CODAMOSA behaves just like SBST, *i.e.*,, creating a set of mutated test cases, and updating the archive and population with these mutants. If the coverage stall has reached the maximum length, CODAMOSA calls the target generation step, which invokes the model to generate test cases targeting low-coverage test objects. In the target generation step, they first compute the coverage for each of the test objects and then query the model. They use the source code of the module under test and an appropriate form of a function header as the prompt. With the test cases Codex generated, they further deserialize the generated test cases into the search algorithm's internal representation. Then, the union of the generated test cases and their mutants is used to update the archive and population, as in a regular SBST. They evaluate CODAMOSA on Pynguin and BugsInPy datasets. Results show that CODAMOSA achieves significantly higher coverage on many more of the benchmarks (173 vs MOSA, 279 vs Codex_ONLY) than it reduces coverage on (10 vs MOSA, 4 vs CODEXONLY). They also explore different design decisions and find that sampling Codex with a higher temperature has the most consistently positive effect on achieved coverage. They notice that more complex prompting yielded better results in some cases, but was less consistent than the simple prompting. They conclude that further prompt engineering could improve results.

Meanwhile, in 2023, Tang *et al.* [109] compare ChatGPT and the advanced SBST tool EvoSuite comprehensively. In terms of correctness and execution capability, ChatGPT successfully generates unit test cases for all 207 Java classes, with 69.6% of test cases compiling and executing without manual intervention. However, some test cases have compilation errors, mainly because ChatGPT doesn't fully understand the entire project, SpotBugs detection reveals 403 potential defects among 204 test cases, with most being low-priority issues. Regarding readability, the test suite generated by ChatGPT violates coding style conventions, especially in terms of indentation. The study suggests improving code consistency and maintainability. In code coverage, EvoSuite outperforms ChatGPT with an average statement coverage of 74.2%, compared to ChatGPT's 55.4%. Although EvoSuite generally excels, ChatGPT demonstrates advantages in 37 cases, highlighting its strengths in generating diverse test data and understanding semantics. In defect detection, assertions in test cases generated by ChatGPT are deemed potentially unreliable, introducing some misleading aspects. However, in an experimental comparison using the Defects4J project, ChatGPT successfully identifies 44 defects, slightly fewer than EvoSuite's 55. Overall, the study recommends EvoSuite as the preferred tool due to its robust performance in code coverage and defect detection. Nevertheless, ChatGPT retains value, particularly in generating diverse test data and semantic understanding, positioning it as a viable entry-level testing tool or alternative choice. Combining search-based software testing techniques with NLP modules holds promise for enhancing the performance of SBST tools, offering developers a more efficient and reliable software testing experience.

4.3.5 Assertion Generation

Assertion Generation refers to the process of automatically inferring or creating expected outcomes or conditions against which the output of a software system or program is validated. These oracles serve as a standard or benchmark against which the system's behavior is tested, allowing for the automatic detection of bugs, discrepancies, or unexpected behaviors within the software. It involves using various methods, ranging from hard-coded patterns, natural language processing, neural networks, and machine learning models to derive or predict the expected outcomes, assertions, or exceptional behaviors of a given code snippet, function, or method. The goal is to generate these oracles accurately to enable automated testing and the identification of potential flaws or issues within the software.

In 2022, based on AthenaTest [108], Tufano *et al.* [105] propose to leverage the BART model to generate accurate assert statements in unit test cases. They first perform semi-supervised pre-training on a large corpus of English text to help BART learn the semantic and statistical properties of natural language. They then pre-train BART on abundant Java source code crawled from GitHub with a similar pre-training strategy to English pre-training. After that, they fine-tune it on a dataset mined from more than 9 thousand open-source GitHub projects containing unit test cases defined with JUnit. By fine-tuning previous BART models, they finally obtain four different models: BART_Scratch, BART_English, BART_Code and BART_English+Code. They then compare their approach with the previous RNN-based approach ATLAS and explore the impact of different pre-trainings on the assertion generation performances. Results show that their approach outperforms ATLAS with a relative improvement of 80% on top-1 accuracy. Besides, pre-training on English text boosts the performances of 23-25%, while further pre-training on source code can yield an additional 2% improvement. Thus, they conclude that pre-training has a significant positive effect on downstream performances and their models can generate common assert statements as well as complex ones involving method calls, parameters, and unusual variables.

Despite promising, the previous approach [105] struggles to find real-world bugs. In 2022, Dinella *et al.* [111] propose a transformer-based approach TOGA to infer both exceptional and assertion test oracles based on the context of the focal method. TOGA contains two key components: the Exceptional Oracle Classifier and the Assertion Oracle Ranker. The Exceptional Oracle Classifier leverages the CodeBERT model trained on both natural language and code-masked language modeling. They fine-tune CodeBERT on the task of exceptional oracle inference using a supervised Methods2Test dataset. The Assertion Oracle Ranker is also based on the pre-trained CodeBERT model. They fine-tune it on the task of assertion oracle inference using the supervised Atlas dataset. Based on the return value of the unit under test, they iteratively construct a set of candidate assertions. Besides, they apply EvoSuite to obtain a high-quality test prefix. In combination with a large set of prefixes that attempt to cover the entirety of the unit, TOGA is able to generate functional test oracles that find real-world bugs They then evaluate TOGA on two fine-tuning datasets, the Atlas dataset as well as the Defects4J dataset. Results show that the assertion oracle inference model achieves over 69% accuracy compared to 62% accuracy from existing approaches and the exceptional inference model achieves 86% accuracy. Besides, TOGA finds 57 bugs in real-world Java projects, 30 of which are not found by any other method in the evaluation. They also notice that 82% of the developer-written assertions in the ATLAS dataset are in their grammar, and many other assertions are semantically equivalent to assertions expressed in their grammar.

Unlike previous studies fine-tuning LLMs for assertion generation, Nashid *et al.* [17] introduce Cedar, a few-shot learning method that utilizes a prompt-based approach for both test assertion generation and program repair. Cedar employs neural search and frequency-based retrieval to select relevant demonstrations, then generates prompts using task-specific templates. For test assertion generation, the prompt includes code demonstrations and the query with unit test context and instructions. Each demonstration features the focal method, test method, and expected assertion. For program repair, the prompt mirrors this structure but uses error code snippets, error contexts, and fixed code snippets. Cedar employs the Codex model for code generation, and its performance is evaluated on the ATLAS dataset for assert generation and the TFix dataset for program repair. The results show that Cedar significantly outperforms both fine-tuned and task-specific models on these tasks, achieving notable accuracy without the need for pre-training or fine-tuning and using a smaller set of examples.

4.3.6 Test Suite Minimization

Test suite minimization aims at improving the efficiency of software testing by removing redundant test cases, thus reducing testing time and resources while maintaining the effectiveness of the test suite.

Since previous test suite minimization approaches that rely on test code (black-box) are rather time-consuming and resource-consuming, Pan *et al.* [112] propose LTM, a black-box similarity-based approach that leverages LLM to address the scalability problem. They explore three off-the-shelf pre-trained models for similarity measurements: CodeBERT, GraphCodeBERT and UniXcoder. These models take the source code of test cases as input to generate numeric vectors. Then, they employ two similarity measures for calculating the similarity between test method embeddings: Cosine Similarity and Euclidean Distance. Cosine similarity measures the angle between two vectors, whereas Euclidean distance calculates the straight-line distance between them. They evaluate test suite minimization on the same Java dataset as ATM [175] and compare it to ATM. Results show that UniXcoder/Cosine is the best LTM configuration when considering both effectiveness and efficiency. Besides, LTM outperforms ATM by achieving significantly higher fault detection rate results. They also notice that LTM is running much faster than ATM in terms of both preparation time and minimization time.

4.3.7 Fuzzing

Fuzzing is an automated software testing method that injects invalid, malformed, or unexpected inputs into a system to reveal software defects and vulnerabilities. A fuzzing tool

TABLE 9
A summary and comparison of LLM-based fuzzing studies

Study	LLMs	Type	Repository
Dakhama *et al.* [113]	ChatGPT	General-purpose Fuzzing	https://github.com/karineek/SearchGEM5/
Hu *et al.* [114]	ChatGPT	General-purpose Fuzzing	N.A.
Yang *et al.* [115]	GPT4, StarCoder	Compiler Fuzzing	https://github.com/ise-uiuc/WhiteFox
Menglarge *et al.* [116]	ChatGPT	Protocol Fuzzing	https://github.com/ChatAFLndss/ChatAFL
Zhang *et al.* [117]	ChatGPT	Fuzz Driver Generation	https://sites.google.com/view/llm4fdg/home
Deng *et al.* [118]	ChatGPT	DL Library Fuzzing	https://github.com/ise-uiuc/TitanFuzz
Deng *et al.* [121]	ChatGPT	DL Library Fuzzing	https://github.com/ise-uiuc/FuzzGPT
Xia *et al.* [119]	GPT4	General-purpose Fuzzing	https://fuzz4all.github.io/

injects these inputs into the system and then monitors for exceptions such as crashes or information leakage. Table 9 presents a summary of existing fuzzing studies empowered with LLMs. In the following, we discuss these individual studies in detail.

DL Library Fuzzing. DL libraries like TensorFlow and PyTorch, foundational to the burgeoning field of DL, play a pivotal role in our daily lives due to the widespread adoption of DL systems. It is increasingly crucial to detect potential bugs in these DL libraries, so as to ensure the reliability and safety of these DL systems in some safety-critical scenarios, such as autonomous driving. Since traditional fuzzing techniques cannot satisfy both the input language semantics (*e.g.*, Python) and the DL API input constraints for tensor computations, in 2023, Deng *et al.* [118] propose TitanFuzz, the first approach that directly leverages LLMs to generate input programs for fuzzing DL libraries. Given any target API, they first apply an LLM to generate a list of high-quality seed programs for fuzzing, They query the Codex model with a step-by-step prompt and sample multiple completions. They instruct the model to perform three tasks sequentially: (1) import the target library; (2) generate input data; and (3) invoke the target API. Then, they apply an evolutionary fuzzing algorithm with generated seeds to generate new code snippets iteratively. In each iteration, they select a seed program according to its fitness score and apply different mutation operators to replace parts of the selected seed with masked tokens. After that, they leverage another LLM InCoder to perform code filling to replace those masked tokens. For each generated mutant, they filter out any execution failures and leverage a fitness function they designed to score each mutant. These mutants are placed into the seed bank for future mutations. Finally, they execute all generated programs using differential testing oracle to identify potential bugs. They evaluate TitanFuzz on two of the most popular DL libraries, PyTorch and TensorFlow. Results show that TitanFuzz outperforms API-level fuzzing techniques (FreeFuzz [176], and DeepREL [177]) and model-level techniques (LEMON [24], and Muffin [178]). They also conduct the ablation study to prove that each component in TitanFuzz makes great contributions.

However, due to the nature of LLMs, TitanFuzz tends to generate ordinary human-like DL programs, which can only cover a limited range of program patterns, thereby limiting the exploration of diverse behaviors in DL libraries. Deng *et al.* [121] propose FuzzGPT, which utilizes LLMs to generate unusual programs based on historically bug-triggering programs. FuzzGPT comprises three variants: few-shot learning, zero-shot learning, and fine-tuning. These variants are based on different LLMs, including Codex and CodeGen. Additionally, FuzzGPT can harness the directive-following ability of ChatGPT to generate atypical programs. The experimental results show that FuzzGPT detected 76 errors in PyTorch and TensorFlow, 49 of which were previously unknown, encompassing 11 high-priority errors or security vulnerabilities. This technology is not limited to deep learning libraries and is applicable to other domains and programming languages. Comparisons among the three FuzzGPT variants in PyTorch and TensorFlow reveal that FuzzGPT-FS covers the most APIs and effective programs in both libraries. Its richer contextual information aids in learning various bug patterns and APIs. FuzzGPT-ZS, while triggering more interesting interactions, exhibits lower effectiveness due to the need for improving existing partial programs, limiting the search space. FuzzGPT-FT achieves comparable code coverage in both libraries, demonstrating fine-tuning as an effective fuzz testing method.

Compiler Fuzzing Compilers are the foundation of modern software systems by translating high-level source code written by programmers into machine code, a lower-level language that a computer can execute. Thus, the correctness of a compiler is crucial as it ensures the accurate and efficient execution of the intended functionality of softwares. In 2023, Yang *et al.* [115] introduce WhiteFox, the first white-box compiler fuzzing tool using LLMs in conjunction with source code information to test compiler optimizations. WhiteFox utilizes a dual-model framework, where one analysis LLM examines low-level optimization source code and generates high-level test program requirements that can trigger optimizations, while another generation LLM produces test programs based on the summarized requirements. Evaluated across four different domain-specific compilers, the results demonstrate that WhiteFox can generate high-quality test programs, practicing up to 8 times more optimizations than other fuzzing tools. Before the publication of this paper, WhiteFox has discovered 96 bugs in compiler testing, with 80 confirmed as previously unknown vulnerabilities, and 51 of them have already been fixed. Additionally, WhiteFox can be adapted for white-box fuzzing of complex real-world software systems in general.

Protocol Fuzzing. Protocol implementations are the practical realizations of communication protocols in software or hardware, where correctness is crucial to ensure reliable and secure data transmission across different systems and networks. In 2023, Meng *et al.* [116] explore the opportunity of systematically interacting with pre-trained

LLMs, which have ingested millions of pages of human-readable protocol specifications, to extract machine-readable information about the protocol for use in protocol fuzz testing. They develop CHATAFL, an LLM-guided protocol implementation fuzzing engine. CHATAFL achieves structure-aware mutations concerning the state machine and input structure of the protocol by querying LLMs for information about the protocol's state machine and input structure. The CHATAFL prototype serves as an extended grey-box fuzzing algorithm. The experimental results demonstrate that the LLM-guided stateful fuzzer, CHATAFL, is significantly more effective than the baseline tools AFLNET and NSFUZZ in terms of protocol state space and protocol implementation code coverage. Additionally, CHATAFL uncovers nine previously unknown security vulnerabilities while the baseline tools detect only three or four of these.

General-purpose Fuzzing. Different from previous studies targeting specific scenarios, general-purpose fuzzing (*e.g.,* AFL) is unaware of the programs and focuses on byte-level transformations. In 2023, Xia *et al.* [119] introduce Fuzz4All, the first universal fuzzer to support various software systems based on the multi-lingual capabilities of LLMs. It utilizes auto-prompting techniques to generate effective LLM prompts for fuzzing and iteratively updates prompts to generate diversified fuzzy inputs. Fuzz4All has been evaluated on nine systems across six different languages (*i.e.,* C, C++, SMT, Go, Java, and Python), demonstrating a significant improvement in code coverage compared to previous fuzzers. Furthermore, Fuzz4All is able to discover 76 bugs in widely used systems, with 47 of them confirmed by developers as previously unknown vulnerabilities.

Besides, considering that traditional fuzzers (*e.g.,* AFL) struggle to generate structured test inputs efficiently and at scale, in 2023, Hu *et al.* [114] introduce CHATFUZZ, a grey-box fuzzing tool leveraging generative AI through collaboration with the ChatGPT generation model to enhance test input quality and effectiveness. CHATFUZZ selects a seed from the fuzzer's pool and prompts the ChatGPT generation model to obtain variant inputs conforming better to specific formats. The authors conduct extensive experiments to explore the best practices using generative LLMs. The experimental results demonstrate that CHATFUZZ increases edge coverage by 12.77% compared to the state-of-the-art grey-box fuzzing tool (AFL++) across 12 target programs from three well-tested benchmarks. For vulnerability detection, CHATFUZZ performs well for programs with explicit syntax, but is less effective than AFL++ for programs with more complex syntax. The contributions of this work include identifying the limitations of traditional mutation-based and generative methods in generating structured test inputs effectively. Additionally, CHATFUZZ proposes integrating generative artificial intelligence into grey-box fuzzing and validates the effectiveness of this approach through experiments.

In parallel to CHATFUZZ, Dakhama *et al.* [113] introduce an innovative approach that combines LLM and search-based fuzzing for automated software testing, specifically targeting the gem5 system. The technique leverages ChatGPT to parameterize C programs, compiles the resulting code snippets, and feeds them to the SearchGEM5 extension of the AFL++ fuzzer, utilizing custom mutation operators. Through this system, the authors substantially increase gem5's test coverage by over 1000 lines and discover 244 instances where gem5's simulation behavior differs from the expected behavior of the binary. The gem5 software, which simulates execution on various architectures, presents a significant challenge in creating a test suite due to its large codebase and diverse inputs. The proposed testing method involves using ChatGPT to automatically generate parameterized C programs and AFL++ with custom mutations to diversify the test inputs. Test inputs are generated and evaluated, and the testing process includes coverage-guided mutation-based fuzz testing using AFL++, differential testing, and the creation of a corpus of programs. The experiments conducted involve training LLM to generate test inputs, measuring test generation capabilities, evaluating test coverage, bug finding in gem5, and exploring portability issues. The results show significant coverage improvements, identification of various bugs within gem5, and discussions on the potential portability of the approach to different software systems and languages.

Fuzz Driver Generation. A fuzz driver is a piece of code written to accept inputs from fuzzers and execute the program accordingly. It is labor-intensive and time-consuming for human experts to manually write high-quality fuzz drivers. In 2023, Zhang *et al.* [117] conduct an empirical study to explore the fundamental issues of effective fuzz driver generation using LLMs. This framework includes a quiz with 86 driver generation questions collected from 30 popular C projects and a set of criteria for precise driver effectiveness validation. In total, 189,628 fuzz drivers using 0.22 billion tokens are generated and evaluated. The research results indicate that enhanced query strategies and iterative methods can significantly improve the accuracy and efficiency of generating fuzz drivers.

4.3.8 *Penetration Testing*

Penetration Testing refers to the systematic process of actively assessing an organization's, company's, or system's security defenses by simulating real-world cyberattacks. This method involves identifying potential vulnerabilities and testing the system's resilience against exploitation, thereby providing insights into the system's security gaps and weaknesses.

In 2023, Deng *et al.* [124] propose PentestGPT, an LLM-empowered automatic penetration testing tool that leverages the abundant domain knowledge inherent in LLMs. PentestGPT consists of three core modules: (1) the reasoning module leads overseeing the penetration testing task from a macro perspective; the generation module translates specific sub-tasks from the reasoning module into concrete commands or instructions; and the parsing module operates as a supportive interface between the user and the other two core modules. The experimental results show that PentestGPT significantly improves task completion compared to GPT-3.5 and effectively tackles real-world penetration testing challenges. The system is open-source on GitHub and garners substantial attention in academic and industrial domains.

Meanwhile, Happe *et al.* [123] explores the use of LLMs like GPT-3.5 to enhance penetration testing. The research covers two primary scenarios: high-level task planning and low-level vulnerability discovery. In the latter, Happe *et al.* implements a closed feedback loop, combining LLM-generated operations with a vulnerable virtual machine to analyze machine status and execute attacks. They discuss preliminary findings and delves into potential areas for improvement and ethical considerations. The focus of the study is to explore the potential of LLMs in automating security testing. Using the guidance provided by MITRE ATT&CK, They conduct multiple experiments, showcasing high-level and low-level guidance. They prompt LLM to design penetration tests for generic scenarios and specific target organizations. Additionally, they integrate GPT-3.5 with a vulnerable virtual machine to identify vulnerabilities and attack vectors. Drawing from their experience, the researchers discuss the outcomes and potential future enhancements.

4.3.9 Property-based Testing

Property-based testing (PBT) aims to verify whether the program properties are satisfied by generating a large number of random input data. In comparison to traditional unit testing, PBT emphasizes the program's properties and behaviors rather than singular predefined test cases. This testing method was initially popularized by the QuickCheck library in the Haskell language. The main steps of PBT include property definition, random data generation, and property validation.

Traditional PBT methods are not widely applied in actual software development because crafting diverse random input generators and meaningful test properties poses a challenge. However, developers tend to be more inclined towards documentation writing, and library API documentation contains valuable natural language specifications for PBT. Vikram *et al.* [125] propose PBT-GPT, utilizing LLM to generate random inputs and test properties from API documentation. This study explores three different LLM prompting strategies, revealing various failure modes in PBT-GPT and outlining an evaluation methodology for generator and property quality. Preliminary research reports the results of using PBT-GPT on three Python library APIs. The experimental findings demonstrate the design and evaluation framework of PBT-GPT. In the design phase, researchers introduce three distinct LLM prompting strategies to generate critical components of property-based tests. The evaluation section thoroughly analyzes the quality of the generated generators and properties, encompassing metrics such as validity, diversity, and strength. Additionally, strategies to address potential issues are presented, providing effective pathways for enhancing test quality. Overall, the experimental results offer valuable insights into synthesizing property-based tests using LLM, despite some quality issues. The proposed mitigation strategies and evaluation framework pave the way for subsequent enhancements and improvements.

4.3.10 Failure-Inducing Testing

Failure-inducing testing (FIT) is a software testing approach aimed at identifying test cases that can trigger software errors or faults. This method employs test inputs to provoke specific behaviors or anomalies within a program, thereby detecting and addressing errors in software.

For example, Li *et al.* [122] propose Differential Prompting, a methodology that utilizes ChatGPT to infer program intentions, generate program versions, and conduct differential testing, effectively identifying test cases that trigger software errors. The research finds that ChatGPT's ability to infer program intentions enables it to bypass subtle differences in code, thus identifying the correct program intention. By leveraging this characteristic, Differential Prompting successfully identifies test cases that trigger software errors. Differential Prompting comprises three main steps: program intention inference, program generation, and differential testing. In experiments conducted on different program sets such as QuixBugs and Codeforces, Differential Prompting demonstrates significant advantages in identifying failure-inducing test cases, far surpassing existing baseline methods.

4.3.11 Mutation Testing

Mutation testing represents an established fault-based testing technique. It introduces faults into the programs under examination and requires developers to write tests that uncover these faults. These tests possess the potential to reveal numerous faults, particularly those associated with the introduced faults.

In 2023, Khanfir *et al.* [126] introduce μBERT, which is a mutation testing tool designed to generate natural mutants using CodeBERT. Unlike traditional mutation testing methods, μBERT does not rely on predefined syntactic transformation rules. Instead, it masks tokens in the input expression and utilizes CodeBERT to predict and generate mutants. Research findings demonstrate that μBERT exhibits higher fault detection capabilities than state-of-the-art mutation testing techniques like PiTest. The generated tests have up to 17% higher potential for fault detection compared to PiTest. Moreover, μBERT complements PiTest by detecting 47 faults that PiTest misses, while PiTest identifies 13 faults that μBERT misses.

By incorporating the natural mutant generation approach provided by μBERT, Ibrahimzada *et al.* [127] propose BugFarm, which transforms arbitrary code into multiple complex bugs, achieving efficient code mutation, offering a powerful tool for generating complex defects that are challenging to detect and repair. BugFarm leverages LLMs to mutate code in multiple locations (hard to repair). To ensure that multiple modifications do not notably change the code's representation, BugFarm analyzes the underlying model's attention and instructs LLMs to only change the least attended locations (hard to detect). BugFarm undergoes a comprehensive evaluation, generating over 2.5 million mutations, resulting in 320,000 bugs. The experimental results indicate that BugFarm's superiority in generating bugs that are hard to be detected by learning-based bug prediction methods (up to a 41% higher false alarm rate, and accuracy, precision, recall, and F1 scores are respectively 11%, 6%, 29%, and 21% lower compared to LEAM and μBERT bugs). Additionally, these bugs are hard to be repaired by state-of-the-art learning-based program repair techniques (with a repair success rate lowered by 22% compared to LEAM and μBERT bugs, at 34% and 49%, respectively). BugFarm demonstrates efficiency by mutating code within nine seconds without requiring any prior training time.

Besides, Nong *et al.* [128] propose an injection-based vulnerability-generation technique VulGen, which is not limited not a particular class of vulnerabilities. They first collect C vulnerability fixing examples and mine the patterns for vulnerability injection. Then, they leverage pre-trained CodeT5 to learn the localization of injection. Finally, they apply the extracted vulnerability injection patterns to the localization, which CodeT5 points out. They then evaluate VulGen on a dataset including vulnerabilities from Devign, BigVul, ReVeal, PatchDB and CVEFixes. Results show that a semantic-aware model for injection localization is necessary. Besides, the combination of a pattern-based approach and CodeT5-based localization is promising.

4.3.12 GUI Testing

Graphical user interface (GUI) testing ensures the accuracy and reliability of a mobile application's visual interface. It involves verifying that interactions with components like buttons and text boxes yield expected outcomes. The aim is to confirm the correct information display and intended user interactions. GUI testing can be manual or automated, often using metrics like error detection and code coverage for assessing performance. Its primary goal is to guarantee the stability and reliability of an application's visual and interactive elements.

In 2022, to recognize the diverse and semantic requirements for valid inputs in GUI testing, Liu *et al.* [129] propose an approach named QTypist based on LLMs for intelligently generating semantic input text according to the GUI context. To boost the performance of LLMs on text input in mobile GUI, they develop a prompt-based data construction and tuning method to automatically extract the prompts and answers for model tuning. They leverage the pre-trained GPT-3 model and fine-tune it with the tuning method. For GUI testing, a context-aware input generation method generates the prompt and feeds it into the GPT-3 model. They evaluate QTypist on 106 apps from Google Play. Results show that the passing rate of QTypist is 87%, which is 93% higher than the best baseline RNNInput. They also find that QTypist with the automated GUI testing tools can cover 42% more app activities and 52% more pages compared with the raw tool. They conduct further experiments to demonstrate the value of automated generated tuning data for effective input text generation.

Unlike QTypist [129] focusing on text input generation, in 2023, Liu *et al.* [130] propose GPTDroid, framing the mobile GUI testing problem as a question-answering task, utilizing LLM as a human tester. GPTDroid facilitates the passing of the application's GUI information to LLM to initiate testing scripts and receive and iterate execution results. This framework incorporates a functionality-aware memory prompting mechanism, equipping the LLM to retain testing knowledge and engage in long-term, functionality-based reasoning to guide exploration. Evaluating GPTDroid on 93 apps from Google Play reveals impressive results, showing a 32% improvement in activity coverage compared to the best baseline. GPTDroid also identifies 31% more bugs at a faster rate, even discovering 53 new bugs on Google Play, 35 of which were confirmed and rectified.

4.3.13 NLP Testing

NLP testing refers to the process of assessing and evaluating the performance, accuracy, and robustness of natural language processing systems, including but not limited to machine translation, text generation, language understanding, and other NLP-related tasks. The objective is to verify the effectiveness and reliability of these systems in processing and understanding human language.

As early as 2020, He *et al.* [132] introduce SIT, a metamorphic testing approach leveraging BERT to validate machine translation software empowered by BERT. SIT hinges on the principle that translations of similar source sentences should maintain comparable sentence structures SIT uses BERT to create a set of similar sentences by altering a single word in the source sentence, aiding in testing the translation's structural consistency. Similarly, motivated by the notion that sentences with distinct meanings should yield different translations, Cupta *et al.* [133] propose PatInv, which generates syntactically similar but semantically different sentences using BERT. In 2022, Sun *et al.* [131] introduce CAT, a BERT-driven word-replacement method for enhancing machine translation. CAT utilizes isotopic replacement in two stages: it initially generates a set of candidate words for each word in the original sentence using BERT. Then, leveraging BERT again, it computes a context-aware semantic representation to filter inappropriate words, forming the final word set.

In addition to the aforementioned machine translation studies, researchers have explored the usage of LLMs, particularly BERT, in other NLP testing scenarios, such as named entity recognition software testing [134], textual content moderation software testing [135], dialogue system testing [136], question answering system [137].

4.4 Software Maintenance

Software maintenance is one of the foundational aspects of software engineering, and encompasses the ongoing process of post-delivery software modification, aiming to rectify errors and meet emerging requirements.

4.4.1 Program Repair

Automated program repair aims to generate correct patches for a detected buggy code snippet automatically and plays a crucial role during software maintenance. A typical repair technique usually contains three steps: (1) applying off-the-shelf fault localization techniques to recognize the suspicious code elements; (2) modifying these elements based on a set of transformation rules to generate candidate patches; (3) adopting test suites to verify all candidate patches.

Table 10 presents a summary of existing LLM-based repair techniques to generate correct patches automatically in the literature. The first and second columns list the repair technique and the time of publication. The third and fourth columns list the utilized LLMs and the public repositories. Overall, we summarize the existing program repair work involving LLMs into five stages. Initially, program repair is directly used as a downstream task to evaluate the capability of LLMs when such LLMs are designed and proposed in their original paper, such as CodeT5 and CodeBERT. Subsequently, there exist explorations in the SE field using

TABLE 10
A summary and comparison of LLM-based APR studies

Year	Study	LLMs	Repository
2021	Jiang *et al.* [138]	GPT	https://github.com/lin-tan/CURE
2021	Berabi *et al.* [141]	T5	https://github.com/eth-sri/Tfix
2021	Drain *et al.* [145]	BART	N.A.
2022	Li *et al.* [139]	BERT	https://github.com/AutomatedProgramRepair-2021/dear-auto-fix
2022	Yuan *et al.* [16]	T5	https://github.com/2022CIRCLE/CIRCLE
2022	Xia *et al.* [13]	CodeBERT	https://zenodo.org/record/6819444
2022	Fan *et al.* [148]	Codex	https://github.com/zhiyufan/apr4codex
2022	Kim *et al.* [179]	TFix	N.A.
2023	Jin *et al.* [140]	Codex	N.A.
2023	Wei *et al.* [142]	CodeT5 and InCoder	https://github.com/ise-uiuc/Repilot
2023	Xia *et al.* [143]	CodeT5	https://zenodo.org/records/8244813
2023	Xia *et al.* [11]	GPT-Neo, GPT-J, GPT-NeoX, Codex, CodeT5, INCODER	https://zenodo.org/records/7592886
2023	Jiang *et al.* [144]	PLBART, CodeT5, CodeGen, InCoder	https://github.com/lin-tan/clm
2023	Xia *et al.* [146]	ChatGPT	N.A.
2023	Sobania *et al.* [147]	ChatGPT	https://gitlab.rlp.net/dsobania/chatgpt-apr
2023	Cao *et al.* [149]	ChatGPT	N.A.
2023	Wang *et al.* [150]	CodeT5	N.A.
2023	Zhang *et al.* [18]	UniXcoder, CodeBERT, ChatGPT	https://github.com/iSEngLab/GAMMA
2023	Peng *et al.* [151]	CodeT5	https://github.com/JohnnyPeng18/TypeFix

LLMs as a component in existing repair workflow, such as CURE. Then comes the fine-tuning of LLMs as repair models on historical bug-fixing datasets, which is also the most widely researched topic in the literature. Later, zero-shot learning is utilized to better leverage LLMs, which also indicates a shift in the repair paradigm, *i.e.*, from an NMT task to a close test task in a fill-in-the-blank format. Recently, there have been attempts to combine LLMs with traditional repair techniques to address inherent problems that are difficult for traditional techniques to solve. At the same time, there is a substantial amount of empirical research in the field exploring the actual performances of LLMs in program repair. Now, we list and summarize the existing LLM-based repair techniques as follows.

Repair as a downstream task of LLMs. Since the inception of LLMs, some researchers usually employ program repair as a downstream task (also referred to as code refinement) to evaluate the models' capabilities. In this scenario, program repair is viewed as a general code-code generation task, which translates the buggy code snippet to a correct code snippet on top of natural machine translation. Table 11 presents the comparison results of some LLMs from the summarized papers. The preliminary evaluations have demonstrated the remarkable performance of LLMs in understanding code semantics and learning bug-fixing code changes, as well as fostering subsequent, deeper work introducing such LLMs into the field of program repair.

LLM as a Repair Component. In the SE domain, the earliest exploration is to use LLMs to enhance certain aspects of existing traditional repair techniques.

For example, in 2021, on top of CoCoNut, Jiang *et al.* [138] propose a new NMT-based APR technique CURE empowered with GPT. First, CURE extracts millions of methods from open-source Java projects and use subword tokenization to tokenize these methods. Second, CURE pre-trains GPT on the extracted dataset and fine-tunes it on CoCoNuT's training dataset. Third, CURE applies a new

TABLE 11
The comparison results of representative LLMs on program repair

Model	BFP-Small		BFP-Medium	
	BLEU	Accuracy	BLEU	Accuracy
CodeBERTER	78.26%	17.75%	N.A.	N.A.
CodeT5	77.43%	21.62%	87.64%	13.96%
CodeBERT	77.42%	16.40%	91.07%	5.16%
GraphCodeBERT	80.02%	17.30%	91.31%	0.09%
PLBART	77.02%	19.21%	88.50%	8.98%
RoBERTa	77.30%	15.90%	90.07%	4.10%

code-aware beam-search strategy to improve patch ranking and generate more correct patches. Finally, CURE combines the fine-tuned GPT with CoCoNuT as the full APR pipeline and trains it for the patch generation task. The experimental results show that CURE fixes the most number of bugs, 57 and 26, respectively, on Defects4J and QuixBugs benchmarks. Besides, the compilable patch rates show the effectiveness of GPT and the context-aware search strategy in generating more compilable patches. Importantly, it demonstrates the unique capabilities of combining a GPT PL model and an NMT model to learn both developer-like code and fix patterns to fix more bugs. Besides, in 2022, Li *et al.* [139] propose DEAR, a learning-based APR for multi-hunk, multi-statement bugs empowered with BERT. DEAR fine-tunes BERT to learn the fixing-together relationships among statements, *i.e.*, whether two statements are needed to be fixed together.

Fine-tuning LLMs as Repairers. Inspired by the successful application of program repair as a downstream task for LLMs, more research has delved into exploring the performance of fine-tuning such models in the repair domain.

As early as 2021, Drain *et al.* [145] present a data-driven APR approach DeepDebug which learns to fix bugs in Java methods. They view bug repair as a Seq2Seq learning task consisting of two steps: 1) denoising pre-training, and

2) fine-tuning on the target translation task. They train a BPE tokenizer and use the span-masking objective for pre-training. They consider three pre-training experiments: pre-training on Java only; fine-tuning directly from the baseline BART; and further pre-train on Java with a warm start from BART. They evaluate the approach on the Java dataset BFP small and BFP medium. Results show that pre-training on source code programs improves the number of patches found by 33% as compared to supervised training from scratch. They also find that pre-training on English boosts the performance of program repair and pre-training helps especially for medium methods.

At the same time, unlike Drain *et al.* [145] considering BART, Berabi *et al.* [141] present a learning-based approach TFix, which leverages the pre-trained T5 model to fix JavaScript code errors. First, they apply a code analysis tool named Detector to output the error reports, each including an error type, an error message, and one location. Then, they create a fine-tuning dataset by extracting a large-scale dataset of 100k aligned pairs of coding errors and fixes from GitHub. TFix takes the error reports as input and the fine-tuned T5 model outputs the fixed code. They evaluate TFix on the same dataset and investigate the performance of different T5 models (including T5-base, T5-small, and T5-large). Results show that the accuracy of the T5-large-no-pre-train model trails that of the pre-trained model, which demonstrates that the natural language pre-training is the key for TFix to generate correct fixes. Besides, TFix achieves a significantly 13% higher accuracy than T5-large-fine-tune-per-type on all metrics, which demonstrates the importance of fine-tuning all error types together and the existence of knowledge transfer between different types of errors. Their further analysis shows that model size is an important factor and they pick T5-large as the model for TFix mainly because T5-large achieves the highest accuracy on exact match.

In 2022, Yuan *et al.* [16] propose CIRCLE, a T5-based APR framework equipped with continual learning ability across multiple programming languages. CIRCLE first utilizes a pre-trained model as the repair skeleton, and then incorporates a prompt template to bridge the gap between pre-trained tasks and program repair. To perform the multilingual repair, CIRCLE designs a re-repairing mechanism to eliminate incorrectly generated patches caused by crossing languages. To further strengthen CIRCLE's continual learning ability, CIRCLE applies a difficulty-based rehearsal method to achieve lifelong learning without access to the full historical data and an elastic regularization to prevent catastrophic forgetting. Finally, a simple but effective re-repairing method is adopted to revise generated errors caused by crossing multiple programming languages.

Zero-shot LLM-based Repair. Despite promising, the repair performance of fine-tuned LLMs is usually constrained by the quality and quantity of labeled bug-fixing pairs, similar to previous learning-based repair techniques. Therefore, some researchers attempt to transform the repair problem into a cloze test task under a zero-shot setting, where LLMs are queried to directly predict the correct code tokens based on context information (*i.e.,* buggy methods) without any fine-tuning on historical datasets.

For example, Xia *et al.* [13] propose AlphaRepair as the first cloze-style APR approach that leverages LLMs without any fine-tuning. They implement AlphaRepair based on the CodeBERT model. First, they separate the context and the buggy line of the code snippet. They encode the context in the programming language input and encode the buggy line as a comment in the natural language input for CodeBERT. Then, they generate the buggy line with multiple templates. They would replace the entire buggy line with a line containing only mask tokens or reuse partial code from the buggy line and only mask the remaining. They also implemented several template-based mask line generation strategies targeting conditional and method invocation statements. After that, they will query CodeBERT to generate candidate patches iteratively. Finally, CodeBERT would provide patch ranking and they would validate each candidate patch. They evaluate AlphaRepair on Defects4J and QuixBugs benchmarks. Results show that learning-based APR without any history of bug fixes can substantially outperform previous APR techniques like Recoder, CURE, CoCoNut and DLFix. Besides, AlphaRepair shows a high multilingual repair ability on Java and Python.

In 2023, Wei *et al.* [142] introduce Repilot, with the fundamental concept of treating the autoregressive token generation process of LLMs in a manner similar to human code composition. Repilot utilizes a completion engine to offer real-time feedback, ensuring the validity of the generated code. The completion engine has the capability to understand the code's structure and semantics, thus assisting LLMs in producing better patches. Repilot has two instances, one using CodeT5-large, an encoder-decoder LLM with 7.7 billion parameters, and the other using InCoder-6.7B, a decoder-only LLM with 6.7 billion parameters. Both LLMs are capable of code completion based on context. Additionally, Repilot implements a Java completion engine that can perform semantics-based code analysis. The evaluation of Repilot is based on a subset of the well-established Defects4J 1.2 and 2.0 datasets. The results show that Repilot outperforms other state-of-the-art APR tools, fixing more bugs and generating patches with higher compilation rates. This indicates that Repilot's framework effectively enhances the capabilities of automated program repair.

At the same time, Xia *et al.* [143] introduce FitRepair, a method primarily aimed at LLM-based APR, especially in the context of cloze-style APR. This method seeks to enhance the performance of automated repair by combining the direct utilization of LLMs and the "beautification assumption" knowledge. It employs three main strategies, including Knowledge-Intensified Fine-tuning, Repair-Oriented Fine-tuning, and Relevant-Identifier Prompting. In experiments, the FitRepair approach uses the LLM model CodeT5 to generate repair patches in four different ways: the original CodeT5 model, two fine-tuned models (knowledge-enhanced and repair-oriented), and a CodeT5 model with prompts. These generated repair patches are ranked to identify the most likely correct repair.

Combination of LLMs and traditional APR. Most LLM-based repair techniques utilize LLMs as an end-to-end learning-based patch generator and are developed separately from mature traditional techniques. Inspired by the fact that learning-based APR is complementary to traditional repair techniques in terms of fixed bugs, Zhang *et al.* [18] propose GAMMA to combine the advance of recent

LLMs and state-of-the-art template-based repair. Instead of retrieving donor code in the local buggy file, GAMMA directly predicts the correct code tokens based on the context code snippets and repair patterns by a cloze task. Particularly, GAMMA first summarizes a variety of fix templates from the template-based repair literature and transforms them into mask patterns, and then adopts LLMs to predict the correct code for masked code as a fill-in-the-blank task. The experimental results demonstrate that equipped with UniXcoder, GAMMA is able to generate correct patches for 82 bugs on Defects4J-v1.2 and 45 bugs on Defects4J-v2.0. More importantly, GAMMA highlights the promising future of adopting LLMs to generate correct patches on top of fix patterns in practice.

At the same time, Peng *et al.* [151] propose a domain-aware prompt-based approach TypeFix to repair Python type errors with fix templates. TypeFix contains two main phases: the template mining phase and the patch generation phase. In the template mining phase, they first define a set of fix templates containing three parts: fix pattern, internal context and external context. Then, they parse all type error fixes into specific fix templates. After that, they abstract and merge parsed specific fix templates into general fix templates via a hierarchical clustering algorithm they designed. In the patch generation phase, TypeFix first selects matched fix templates on clustering trees via Breadth-First Search and then ranks fix templates with frequency and abstraction ratio. Then, TypeFix applies ranked fix templates on the buggy program and generates code prompts for the CodeT5-base model, which will fill the masks in code prompts and generate candidate patches. They evaluate TypeFix's performance on BugsInPy and TypeBugs benchmarks with PyTER, AlphaRepair, CoCoNuT and Codex. Results show that TypeFix successfully fixes 55 and 26 bugs in two benchmarks, outperforming baseline approaches by at least 14 bugs and 9 bugs, respectively. Meanwhile, TypeFix obtains the most unique type error fixes in two benchmarks. They also find that TypeFix achieves a template coverage of about 75% on both benchmarks and ablation results demonstrate the usefulness of fix templates mined by TypeFix under each category. However, they also notice that TypeFix sometimes fails to fix type errors due to the limited performance of pre-trained code models and a few cases (25%) that mined fix templates cannot cover.

Empirical Evaluation of LLMs. In conjunction with the aforementioned repair strategies tackling certain technical obstacles, there has been a concurrent surge in empirical studies meticulously examining the development and nuances of these methodologies. These empirical studies systematically explore the actual performance of LLMs during the repair workflow, with the sim to furnish insights for forthcoming program repair endeavors A summary of the existing empirical studies is presented below.

To explore the true performance of LLMs in program repair, in 2022, Xia *et al.* [11] conduct an extensive study on the application of LLMs in real-world bug fixing, using nine advanced models of varying sizes. The study explores different LLMs' effectiveness in APR across various tasks and settings, assesses their scalability, speed, and compilation rates. It also examines the influence of different architectures, parameter settings, task types, domains, and programming languages on the LLMs' APR performance. The research highlights the effectiveness of infilling-style APR and LLMs' ability to recognize correct fixes, suggesting larger sample sizes and fix templates as ways to enhance LLM-based APR.

In 2023, Jiang *et al.* [144] propose to evaluate several code-related LLMs on different APR benchmarks to study the effectiveness of fine-tuned LLMs for the APR task. They evaluate 10 LLMs: PLBART, CodeT5, CodeGen and InCoder with different sizes. For benchmarks, they use Defects4J, QuixBugs and a manually created dataset HumanEval-Java which is used to overcome potential data leaking. They run developer-written test cases to validate the correctness of generated patches to evaluate and compare the ten LLMs. Results show that the best CLM is able to fix 72% more bugs than the state-of-the-art DL-based APR techniques without fine-tuing. Besides, fine-tuning improves LLMs' fixing capabilities by 31%–1,267%, and fine-tuned LLMs outperform DL-based APR techniques significantly, by 46%–164%. However, they also notice that sometimes fine-tuning makes LLMs over-rely on the buggy lines, and thus fail to fix some bugs that can be fixed without fine-tuning. They also find that CodeT5 and InCoder models have the best size efficiency, suggesting developing larger CodeT5 or InCoder models is the most promising.

Unlike previous studies focusing on software bugs [11], [144], in 2022, Fan *et al.* [148] conduct a systematic study to explore whether APR techniques can fix the incorrect solutions produced by language models in LeetCode contests. They use the pre-trained Codex code-davinci-002 and Codex-e code-davinci-edit-001 models to produce solutions. They build a dataset LMDefects with 113 programming tasks in LeetCode which contains 60 easy and 53 medium-level programming tasks. Among them, 46 tasks have been successfully solved by Codex and 67 of them remain unsolved. They then compare the performance of TBar, Recoder and Codex edit mode. Codex edit mode takes a program and a natural language instruction as inputs and outputs an edited program based on the instruction. The results show that existing pattern-based and learning-based APR are ineffective at fixing auto-generated code. They also find that the effectiveness of Codex edit mode with a given specific fault location is nearly comparable to its effectiveness without any location guidance. Besides, by using patch ingredients extracted from Codex edit mode's patches and multiple generated solutions by Codex, they successfully identify the required patch ingredients of more incorrect solutions.

Exploration of ChatGPT. Previous research has focused on using LLMs to generate patches directly but is still constrained by the Generate and Validate paradigm, leading to repetitions and missing information. Xia *et al.* [146] propose ChatRepair, a conversational APR approach using ChatGPT. It leverages interactive conversations, utilizes test failure information for patch generation, and iteratively learns from conversation history to enhance patch accuracy. ChatRepair initially inputs the defective project and error information into ChatGPT, then utilizes detailed test failure information for patch generation. It engages in a dialogue between generating and validating patches until a reasonable fix is identified. ChatRepair is evaluated in Defects4J and

QuixBugs benchmarks. The experimental results demonstrate that ChatRepair successfully generates 114 and 48 correct fixes on Defects4J v1 and v2, respectively, and fixes all 40 errors in QuixBugs. This performance significantly surpasses other learning-based and LLM-based APR tools, such as AlphaRepair, CodexRepair, SelfAPR, RewardRepair, and Recoder, highlighting ChatRepair's outstanding capabilities in conversational-style patch generation and validation.

To analyze the performance of deep learning methods on bug fixing, Sobania *et al.* [147] evaluate ChatGPT on the standard bug fixing benchmark QuixBugs. For each erroneous Python code snippet, they remove contained comments and ask ChatGPT whether the code contains a bug and how it can be fixed. They replace the buggy line with an empty line and query ChatGPT to fix it. Then, they compare the performance of ChatGPT with other APR approaches such as Codex and CoCoNut. They also provide ChatGPT with one hint in each bug-fixing task to evaluate ChatGPT. Results show that ChatGPT performs better than standard APR approaches and ChatGPT fixes about the same number of problems as Codex and CoCoNut without prompting. Besides, adding a hint to ChatGPT vastly improves its performance, with 31 out of 40 problems solved in QuixBugs.

To explore how to utilize ChatGPT to repair deep learning programs, address their unique challenges, and present perspectives on prompt design and ChatGPT capabilities. Cao *et al.* [149] explore ChatGPT's ability to repair deep learning programs. Their study focuses on three aspects: ChatGPT's effectiveness in repairing deep learning programs, improving its performance through optimized prompts, and the role of dialogue in the repair process. They evaluate ChatGPT against other tools like AutoTrainer [180] and DeepFD on a benchmark from StackOverflow and GitHub. The study finds that ChatGPT, while effective in detecting bugs in deep learning programs, benefits significantly from enhanced prompts and multi-round dialogues. However, it faces challenges in certain cases due to issues like catastrophic forgetting or misinterpretation of code intentions. The research offers insights into the strengths and limitations of ChatGPT in deep learning program repair.

Domain Repair. In the LLM-based APR field, researchers have paid considerable attention to semantic and syntax bugs, which represent the most common application of the repair techniques discussed above. Unlike traditional APR, LLMs are pre-trained from a wide range of datasets to learn general language knowledge and can be applied to a wider range of repair scenarios with the buggy and corresponding fixed pairs. For example, in 2023, Jin *et al.* [140] propose InferFix, a Transformer-based approach based on Codex, to automatically fix both critical security and performance bugs detected by the static analysis tool Infer. InferFix first pre-trains a transformer encoder model with the contrastive learning objective, serving as a retriever to search for semantically equivalent bugs and corresponding fixes over a database. InferFix then fine-tuns an off-the-shelf LLM (*i.e.,* 12 billion parameters Codex Cushman model) on supervised bug-fixing pairs with designed prompts augmented from the retriever. The experimental results on the InferredBugs dataset demonstrate the exceptional performance of InferFix in fix three common types of bugs (*i.e.,* Null Pointer Dereference, Resource Leak, and Thread Safety Violation) with a top-1 accuracy of 65.6% for generating fixes in C# and 76.8% in Java. Importantly, InferFix's integration into Microsoft's Azure DevOps and GitHub CI pipelines has markedly enhanced the developer workflow, underscoring its practical utility in real-world software development environments.

4.4.2 *Security Vulnerability Repair*

Software vulnerability predominantly pertains to the weaknesses or flaws found within the software's code or design, which can potentially be exploited to compromise the security or functionality of the hardware or network it operates on. These vulnerabilities do not exclusively reside in the concrete implementation of software, but may also manifest in associated protocols and integrations, putting crucial modern software at risk when not addressed promptly and effectively. Unlike common software bugs focused on by most repair work, securities are more damaging and require more urgent fixes, making it critical to automate vulnerability fixes. Security researchers have to spend a huge amount of effort to manually fix such vulnerable functions, resulting in delays in vulnerability remediation and providing opportunities for attacks. With the successful application of LLMs in program repair, researchers have begun to apply LLMs to help under-resourced security researchers fix software vulnerabilities automatically. Table 12 presents existing vulnerability repair studies empowered with LLMs, which are summarized as follows.

Fine-tuning LLM-based Vulnerability Repair. In 2022, Fu *et al.* [152] propose VulRepair, a T5-based automated software vulnerability repair approach to address the technical limitations of prior work. To handle the out-of-vocabulary problem, they employ a BPE tokenizer and a T5 architecture that considers the relative position information in the self-attention mechanism. They leverage the pre-trained CodeT5 to generate a more meaningful vector representation. The encoder and decoder of VulRepair are similar to the original Transformer architecture. They then evaluate VulRepair on the CVEFixes dataset. Results show that VulRepair achieves a perfect prediction of 44%, which outperforms the baseline approaches (*i.e.*,, VRepair and CodeBERT). They also find that the pre-training corpus improves the percentage of perfect predictions by 30%-38% for vulnerability repair approaches. Besides, the T5 architecture employed by VulRepair still outperforms the BERT architecture when using the same pre-training corpus. Meanwhile, they conduct further experiments to show that the pre-training component of VulRepair is the most important component.

Zero-shot LLM-based Vulnerability Repair. In 2023, Pearce *et al.* [120] examine a zero-shot vulnerability repair approach, assessing the potential of large language models such as OpenAI's Codex and AI21's Jurassic J-1. The primary research challenge lies in designing prompts that prompt LLMs to generate corrected versions of insecure code. The study emphasizes the use of commercially available black-box LLMs, as well as open-source models and locally trained models, for extensive research experiments on synthetic, handcrafted, and real-world security vulnerability scenarios. RQ1 involves parameter adjustments of the Codex model on synthetic programs, revealing its performance in fixing CWE-787 and CWE-89 vulnerabilities. The

TABLE 12
A summary and comparison of LLM-based vulnerability repair studies

Year	Study	LLMs	Description	Repository
2022	Fu *et al.* [152]	CodeT5	proposing a CodeT5-based vulnerability repair approach by fine-tuning	https://github.com/awsm-research/VulRepair
2023	Pearce *et al.* [120]	Codex, Jurassic-1, Polycoder, GPT2	empirically examining use of LLMs for zero-shot vulnerability repair	https://zenodo.org/records/7199939
2023	Zhang *et al.* [12]	CodeBERT, GraphCodeBERT, CodeT5, UniXcoder, CodeGPT	empirically the performance of five LLMs in fixing security vulnerabilities	https://github.com/iSEngLab/LLM4VulFix
2023	Tol *et al.* [104]	GPT, PaLM, LLaMA	exploring the use of LLMs in generating patches for vulnerable code with microarchitectural side-channel leakages.	N.A.
2023	Wu *et al.* [15]	Codex, CodeGen, CodeT5, PLBART and InCoder	evaluate five LLMs on two real-world Java vulnerability benchmarks	https://github.com/lin-tan/llm-vul

results show approximately 2.2% repair for CWE-787 and around 29.6% for CWE-89. RQ2 expands the experimental scope, demonstrating a higher success rate in fixing high-level vulnerabilities. RQ3 explores the LLMs' capabilities in addressing 12 real vulnerabilities, finding that while some fixes may have logical issues, most repairs pass the tests. In RQ4, it is pointed out that although LLMs show significant success in simple local fixes, their performance is inadequate in handling complex contexts. Overall, the study comprehensively evaluates the repair capabilities of LLMs under zero-shot settings, revealing both strengths and limitations in handling actual vulnerability repairs.

Empirical Study of LLM-based Vulnerability Repair. In 2023, Zhang *et al.* [12] conduct the first extensive empirical study to investigate the actual performance of various LLMs on automated vulnerability repair, involving more than 100 fine-tuned LLMs. First, they demonstrate that through simple fine-tuning, LLMs are able to outperform state-of-the-art vulnerability repair techniques with a prediction accuracy of 32.94%∼44.96%. Second, they delved into studying the impact of LLMs on the repair workflow, including data pre-processing, model training, and repair inference phrases. Third, they develop a straightforward vulnerability repair strategy, leveraging transfer learning from bug-fixing, and demonstrate that such a simplified approach further enhances the prediction accuracy of LLMs. Furthermore, they offer additional insights by discussing various aspects, such as code representation and a preliminary study with ChatGPT, to illuminate the capabilities and limitations of LLM-based vulnerability repair approaches. Finally, they precisely identify several practical guidelines, such as enhancing fine-tuning, to advance LLM-based vulnerability repair in the imminent future. Overall, this study underscores the bright prospects of utilizing LLMs to fix real-world security vulnerabilities, holding significant implications for the field of security vulnerabilities. Similarly, Wu *et al.* [15] compare the performance of LLMs with existing learning-based repair techniques, including Codex, CodeGen, CodeT5, PLBART and InCoder.

In response to the growing challenge of microarchitectural vulnerabilities in software like OpenSSL, in 2023, Tol *et al.* [104] evaluate advanced LLMs like OpenAI GPT, Google PaLM2, and Meta LLaMA for generating microarchitectural vulnerability patches, focusing on non-constant time code and Spectre-v1 gadgets. They find that prompt organization is crucial for effective patch generation. GPT4 shows superior performance in fixing leaks and vulnerabilities compared to other LLMs and its predecessor GPT3.5. The study introduces the ZeroLeak framework for testing and patching side-channel leaks in binary files using tools like Microwalk and Spectector. ZeroLeak proves efficient in patching leaks in various programming languages and security libraries, demonstrating the promising capabilities of LLMs in enhancing code security.

4.4.3 Patch Correctness Assessment

It is a common practice for the majority of extant program repair methodologies to predominantly utilize developer-constructed test suites as the program specification, serving to evaluate the accuracy of the patches produced. Nevertheless, such an existing test suite represents an inherently partial specification, delineating only a segment of the program's behavioral domain. Thus, repair approaches may suffer from the patch overfitting issue (*i.e.,* patches passing the available test suites fail to generalize to other potential test suites), limiting the value and deployment of such repair approaches in real-world scenarios. Patch correctness is a crucial phase for developers to further filter out overfitting patches after patch generation (detailed in Section 4.4.1), so as to improve the quality of returned patches. Table 13 summarizes these APCA studies involving LLMs. Considering that the patch overfitting issue is a long-standing challenge in the program repair community and some independent studies have been conducted to address it, we summarize these works separately in this section.

LLMs as Feature Extractor. As early as 2020, Tian *et al.* [153] conduct the first empirical study that involves LLMs for patch correctness assessment. The empirical study aims to investigate different representation learning models for code changes to derive embeddings that are amenable to similarity computations. They apply four different embedding techniques, including re-trained (*i.e.,* Doc2vec, code2vec and CC2vec) and pre-trained models (*i.e.,* BERT). They leverage a pre-trained 24-layer BERT model, which was trained on a Wikipedia corpus. They evaluate these techniques on five literature benchmarks, namely Bugs.jar, Bears, Defects4J, QuixBugs and ManySStuBs4J. Results show that the pre-trained BERT natural language model captures more similarity variations than the CC2Vec model, which is specialized for code changes. Meanwhile, they notice that although BERT achieves the highest recall of filtering incorrect patches, it produces embeddings that

TABLE 13
A summary and comparison of existing APCA techniques involving LLMs

Year	Study	LLMs	Description	Repository
2020	Tian *et al.* [153]	BERT	the first APCA work involving LLMs as code representation tools	https://github.com/TruX-DTF/DL4PatchCorrectness
2022	Tian *et al.* [154]	BERT	transforming APCA into a question answering problem	https://github.com/Trustworthy-Software/Quatrain
2023	Tian *et al.* [155]	BERT	an extended version of prior work [153]	https://github.com/HaoyeTianCoder/Panther
2023	Zhang *et al.* [156]	BERT, CodeBERT, GraphCodeBERT	the first fine-tuning APCA approach	N.A.
2023	Zhou *et al.* [157]	BLOOM, CodeParrot	the first zero-shot APCA approach	N.A.

lead to a lower recall (at 5.5%) at identifying correct patches. Another study result shows that An ML classifier trained using Logistic Regression with BERT embeddings yield very promising performance on patch correctness prediction. In 2023, Tian *et al.* [155] build upon their earlier research [153] by conducting a more comprehensive analysis. In this later study, they assess the effectiveness of code representation, engineered features, and their combined impact on accurately predicting the correctness of patches.

In 2022, Tian *et al.* [154] propose Quatrain, which leverages commit message generation models to produce the relevant inputs associated with each generated patch. They leverage the pre-trained BERT model to embed the natural language text of the bug report and patch descriptions of the patch. The implementation of Quatrain utilizes a pre-trained BERT model to embed bug and patch descriptions, which are then fed into the QA model. Additionally, Quatrain employs CodeTrans to generate patch descriptions. The model used is a pre-trained LLM trained on case-sensitive English text with 24 layers and 1024 embedding dimensions. After representing the text in vector space, numerical computations can be performed on them, such as calculating text similarity or correlation metrics. They compare Quatrain with static and dynamic methods (pure classification based on patch embeddings, BATS, and dynamic method PATCH-SIM). The experimental results show that Quatrain achieves an AUC of 0.886 in predicting patch correctness, while also recalling 93% of correct patches and filtering out 62% of incorrect patches.

Fine-tuning LLM-based APCA. Zhang *et al.* [156] propose APPT, an automated patch correctness assessment technique based on LLMs, aiming to address the overfitting issue in the field of program repair. APPT utilizes BERT as the encoder stack to extract features from source code tokens. Subsequently, it employs bidirectional LSTM layers to capture dependency information between source and repaired code snippets, ultimately constructing a deep learning classifier to predict if patches suffer from overfitting. Unlike traditional approaches, APPT relies solely on source code tokens for input, automatically extracting features without the need for complex code-aware features or manual design. They conduct experiments on 1,183 Defects4J patches. The experimental results demonstrate that APPT outperforms existing learning-based and traditional automatic patch correctness assessment techniques in metrics such as accuracy, precision, recall, F1 score, and AUC. In an additional study involving 49,694 real-world patches from five different benchmarks, APPT exhibits over 99% accuracy across all metrics, showcasing exceptional performance in evaluating patch classification techniques. When equipped with GraphCodeBERT, APPT's precision and recall surpass CACHE by 4.2% and 7.2%, respectively.

Zero-shot LLM-based APCA. In order to assess whether it is possible to use labeled patches generated by existing APR tools to predict the correctness of patches generated by a new/unseen APR tool, Zhou *et al.* [157] propose PatchZero, a zero-shot patch correctness assessment LLM. For new/unseen APR tools, PatchZero does not undergo a fine-tuning phase but directly performs inference on the test set. PatchZero is built upon BLOOM and CodeParrot and reformats the format of the automated patch correctness assessment task to match the original pre-training objective of the LLMs[at. PatchZero is evaluated on a dataset consisting of a total of 1,183 patches from the Defects4J benchmark. The experimental results indicate that PatchZero significantly outperforms the prior state-of-the-art technique Cache by a margin of 19.1% to 26.6%. Furthermore, it leads to substantial improvements over the baseline CodeBERT, ranging from 8.1

4.4.4 Commit Message Generation

Commit message generation attempts to create natural language descriptions for code commits [181]. It begins with modified code snippets and employs template-based, retrieval-based, or learning-based models to generate commit messages. The key lies in understanding the context of code modifications and accurately describing these changes in the commit messages. For example, Jung *et al.* [158] introduce CommitBERT, an encoder-only LLM based on the CodeBERT architecture. CommitBERT is a method based on CodeBERT for generating Git commit messages. It processes a dataset of 345K code modifications and corresponding messages, focusing on changes rather than the entire git diff for efficiency. CommitBERT demonstrates the effectiveness of using CodeBERT's initial weights to bridge the gap between programming and natural language. By removing delete tokens from the dataset and using CodeBERT as the initial weight during training, better message generation results are achieved. CommitBERT shows good performance in languages like Python, PHP, JavaScript, Java, Go, and Ruby.

4.4.5 Code Review

Code review is an integral part of the software development lifecycle, aiming to ensure code quality. Modern code review activities necessitate developers to review, understand, and execute code to assess logic, functionality, latency, style, and other factors. The improvements in LLM combined with Code Review mainly focus on leveraging LLMs (primarily the T5 model) to automate the code review process. They predominantly concentrate on enhancing code quality and review efficiency through data collection and preparation, pre-training task design, and evaluation for performance improvement.

Exploration of T5 in Code Review. In 2022, Li *et al.* [160] introduce AUGER, which uses a pre-trained T5 model to automatically generate code review comments. Firstly, it collected 79,344 Java code reviews from GitHub and then built a framework utilizing the T5 model for automatic integration and generation of review comments. This collaborative approach effectively captures the relationship between code and review language, outperforming baseline models such as LSTM, COPYNET, and CodeBERT, and allows for real-time feedback. The model is capable of further training and covers unfamiliar programs more freely than individuals. Additionally, it adopts several criteria from previous studies to assess the generated comments, providing a certain degree of usefulness akin to human review comments. In future work, the paper aims to explore implementation without language differences and extend its widespread application for collaborative review of new systems with professionals.

In parallel to AUGER [160], Tufano *et al.* [161] propose to apply a pre-trained T5 model for code review tasks. They create a pre-training Java dataset including both source code and technical English by collecting from two datasets (*i.e.*,, the official Stack Overflow dump and CodeSearchNet) and filtering out unqualified instances. To perform the pre-training, they randomly mask in each instance 15% of its tokens. They adopt T5-small and pre-train it with the same configuration by Raffel *et al.* [6]. Then, they create the fine-tuning dataset by mining Java open-source projects from GitHub and from the six Gerrit installations which contain code review data. They extract triplets ¡m_s,c_{nl},m_r¿ from both datasets: a method submitted for the review, a review's comment suggesting code changes and the revised version of the code. They use the fine-tuning dataset for three downstream tasks. In the first task (code-to-code), the model takes m_s as input and is expected to generate m_r. In the second task (code&comment-to-code), the model takes both m_s and c_{nl} as input and is expected to generate m_r. In the third task (code-to-comment), the model takes m_s as input and is expected to generate c_{nl}. After evaluating the approach on the fine-tuning dataset, they find that their approach outperforms the baseline model [182] and the non-pre-trained T5 model.

Domain LLM for Code Review. Li *et al.* [159] propose CodeReviewer, which is based on the Transformer architecture, adopts the same structure as the T5 model, and initializes it with parameters from CodeT5. CodeReviewer focuses on how to use pre-training techniques to automate the code review process, ensuring code quality. The authors start by collecting a large dataset from open-source projects on GitHub, covering nine programming languages, consisting of code changes and code reviews. They then use this dataset for pre-training and subsequently for three important tasks: code change quality estimation, review comment generation, and code refinement. To enhance the model's understanding, Li *et al.* design four pre-training tasks, including Diff Tag Prediction (DTP) task, denoising code diff (DCD), denoising review comment (DRC), and review comment generation. These tasks aim to help the model better understand code differences and generate relevant review comments. The authors categorize these tasks into classification and generation tasks, utilizing the pre-trained encoder for the former and the entire encoder-decoder model for the latter. Finally, they evaluate CodeReviewer's performance on both the pre-training dataset and the processed dataset, with results indicating that CodeReviewer outperforms two previously established pre-trained models: T5 and CodeT5-base.

Exploration of ChatGPT in Code Review. Guo *et al.* [162] explore the potential of ChatGPT in automated code refinement tasks through a pioneering empirical study, with a specific focus on code refinement based on code reviews. The authors assess the impact of different ChatGPT configurations on its performance across standard code review benchmarks and a newly collected dataset. In this empirical study, the researchers optimize parameter settings for ChatGPT and find that it performs exceptionally well on both the CodeReview dataset and the newly constructed CodeReview-New dataset, demonstrating superior generalization capabilities. A detailed analysis reveals that ChatGPT exhibits stable performance compared to CodeReviewer, particularly excelling on the high-quality CodeReview-New dataset. However, for cases where CodeReviewer performs less optimally, the researchers identify several key root causes, including accuracy in understanding review content, excessive deletions, additional modifications, and difficulty in grasping foundational truths in code blocks. The study also uncovers challenges faced by ChatGPT in tasks such as document and functionality refinement, primarily stemming from inadequate domain knowledge, unclear location information, and ambiguities in review comments regarding changes. The authors propose potential strategies for improvement, emphasizing the enhancement of review quality and the utilization of more advanced LLMs like GPT-4. Overall, this research provides insights into the potential of ChatGPT in code refinement tasks and lays the foundation for future studies integrating ChatGPT more deeply.

4.4.6 *Bug Report Detection*

Duplicate Bug Report Detection (DBRD) plays a crucial role in software development. Sometimes, when users report software defects in issue tracking systems such as Bugzilla, Jira, or GitHub, duplicate bug reports emerge. The task of DBRD is to automatically identify and label these duplicate bug reports, enabling developers to avoid redundantly dealing with the same issues. Recognizing duplicate bug reports saves time and effort for developers, thereby enhancing the efficiency of software development.

In DBRD, algorithms and methods compare and analyze the similarity between different bug reports to determine if they describe the same defect or problem. These methods typically involve comparing textual content, using natural language processing and machine learning techniques to establish the similarity between reports and categorize them as duplicate or distinct bug reports. Notably, ChatGPT shows significant potential in DBRD.

For example, in 2023, Zhang *et al.* [163] introduce a method named Cupid, which integrates the traditional DBRD approach REP with ChatGPT. Cupid utilizes ChatGPT in a zero-shot setting to extract key information from bug reports, which is then used as input for REP to detect duplicate bug reports. Evaluation across three datasets reveals that Cupid achieves a new state-of-the-art level, with Recall Rate@10 scores ranging from 0.59 to 0.67. Specifically, Cupid improves Recall Rate@10 by 6.7% to 8.7% over previous state-of-the-art methods on these datasets. Moreover, Cupid's performance surpasses that of deep learning methods, reaching up to 79.2%.

Meanwhile, Plein *et al.* [164] conduct empirical research, as outlined in [164], on how to utilize ChatGPT to transform user-provided software defect reports into formal test case specifications. They employ ChatGPT to generate test cases and evaluate the executability and validity of these generated test cases. Experimental results demonstrate the significant potential of ChatGPT in converting informal defect reports into formal test cases, holding crucial implications for automated software testing and defect resolution tasks.

4.4.7 Bug reproduction

Bug reproduction is a part of the software bug-fixing process, aimed at resolving issues within the software by reproducing and locally replicating the environment where the initial bug occurred. This is a highly collaborative process, in which software developers use reproduction steps from users, software screenshots, tracking logs, and other issue descriptions as information sources [183]. Bug reproduction holds significant importance in software maintenance, but the challenge of replicating the client-side context makes it an obstacle in software development. This process requires a human-centric approach for research and comprehension to design tools that can address the challenges faced by developers in the bug reproduction process.

Kang *et al.* [165] introduce a framework named LIBRO, which utilizes LLMs to generate potential test cases from bug reports and subsequently ranks and suggests these generated solutions through post-processing steps. When evaluating LIBRO on the Defects4J benchmark, Kang et al. find that LIBRO produces failure-reproducing test cases for 33% of the studied cases. Moreover, it initially generates a bug-reproducing test for 149 defects. To minimize the possibility of data contamination, they further evaluate 31 bug reports submitted after the collection of LLM training data. LIBRO generates bug-reproducing tests for 32% of the investigated bug reports. Overall, experimental results indicate that LIBRO significantly enhances developer efficiency by automatically generating tests from bug reports.

4.4.8 Bug Reply

Bug replay refers to the process of reproducing or recreating software defects or issues based on the information provided in a bug report. A bug report is a document that describes the problems encountered by users while using the software, detailing what happened, what was expected to happen, and sometimes including the steps to reproduce the issue. The process of bug replay is crucial for software maintenance as it enables developers to understand, replicate, and fix the defects reported.

Traditional bug replay methods often rely on manual steps where developers follow the instructions in the bug report to reproduce the issue. However, researchers have explored automated bug replay methods to expedite and streamline this process. These methods leverage technologies such as NLP and Machine Learning to extract relevant information from bug reports and guide the software in automatically reproducing the reported defects. The use of LLMs represents an advancement in bug replay, enhancing the capabilities of automated approaches.

In 2023, Feng *et al.* [166] propose a lightweight approach AdbGPT to automatically reproduce Andriod bugs from bug reports without any training. AdbGPT contains two main phases: the Steps to Reproduce (S2R) Entity Extraction phase and the Guided Replay phase. During the initial phase, they provides ChatGPT with details regarding entity specifications and available actions, as well as action primitives. Then, they apply chain-of-thought reasoning to instruct ChaGPT to generate a coherent series of intermediate steps that lead to a reasonable output. After that, given a test bug report as the test prompt, they query for the S2R entities and ChaGPT will consistently generate a numeric list to represent the extracted S2R in the same format as the example output. In the second phase, they first transfer the GUIs into domain-specific prompts that ChaGPT can understand by converting the view hierarchy of the GUI screen into HTML text. Then, they prompt ChaGPT which is adopted with chain-of-thought reasoning to generate the target component to operate on, ultimately triggering the bug. They evaluate the performance of AdbGPT with ReCDroid and MaCa. Results show that AdbGPT is significantly better than that of other baselines in all metrics, *i.e.*,, on average 39.3%, and 42.2% more accurate in step extraction and entity extraction compared with the best baseline (MaCa). In addition, they find that applying few-shot learning and chain-of thought which provide examples with intermediate reasons, can endow ChatGPT with a better understanding of the task, resulting in a boost of performance up to 90.8%. Their further analysis shows that chain-of-thought leads to a substantial improvement on guided replay (*i.e.*,, 18.8% boost) in the performance of AdbGPT, indicating that the LLMs can better understand the task by processing it step-by-step.

4.4.9 Test Update

Test update refers to the process of modifying existing test cases to align them with recent changes in the production code. This is a crucial aspect of maintaining the quality and relevance of software tests throughout the software lifecycle. Test updates are necessary because as the production code evolves to incorporate new features, fix bugs, or adapt to new requirements, the corresponding test cases must also be revised to ensure they accurately assess the software's functionality and performance.

Hu *et al.* [167] propose CEPROT, a method based on CodeT5, aimed at automatically identifying and updating test cases that become obsolete due to changes in production code. It includes two stages: identifying outdated test cases and updating these test cases based on changes in the production code. CEPROT uses two datasets for training: one for identifying method-level production-test co-evolution and another for updating obsolete test cases. These datasets are derived from Java projects with a large amount of high-quality unit tests. CEPROT is trained on positive and negative samples, where positive samples are test cases that need updating with production code changes, and negative samples are those that do not require updates. CEPROT is compared with several baseline models, including KNN, SITAR, LSTM, and NMT. The experimental results show that CEPROT significantly outperforms these baseline models in identifying and updating obsolete test cases. Specifically, CEPROT achieves a precision, recall, and F1 score of 98.3%, 90.0%, and 94.0% respectively in the identification of obsolete test cases. In terms of updating, it attains a CodeBLEU score of 63.1% and an accuracy of 12.3%. In the evaluation of its two-stage application, CEPROT also significantly surpasses the KNN and NMT models in the proportion of correctly updated true positive test cases, demonstrating its efficiency and effectiveness.

Answer to RQ2: Overall, LLMs have been employed across various stages of SE research, tackling a diverse array of 43 tasks. On one hand, these tasks align with previous downstream ones (detailed in Section 3.3), yet they are explored more thoroughly, such as program repair. On the other hand, researchers have turned their attention to a greater number of more complex SE tasks. These tasks may (1) include complex processes that LLMs cannot fully handle, such as fuzzing; (2) include other inputs such as test reports, such as GUI testing; and (3) some unique areas of SE, such as patch correctness assessment. Researchers need to devote more effort to addressing these more domain-specific issues, such as designing specific LLMs or embedding them into existing research workflow. Notably, software testing and development have seen more extensive applications of LLMs. This trend may stem from the fact that these areas often serve as foundational downstream tasks for LLMs, where they have shown considerable promise. Besides, these tasks can be addressed naturally by existing LLMs in the form of sequence-to-sequence code generation. However, the application of LLMs in software requirements is still relatively unexplored, suggesting a potential area of focus for future research in this field.

5 RQ3: How are LLMs empirically evaluated in SE research?

5.1 Benchmark

Different from previous learning-based techniques conducted in a traditional training-and-evaluation pipeline, the process of LLM-based SE techniques is three-fold, *i.e.,* (1) a pre-training process with unsupervised learning on large datasets; (2) a fine-tuning process with supervised learning with labeled datasets; and (3) an evaluation process with limited datasets. Benefiting from a large amount of research effort in the community, there are several existing benchmarks to evaluate LLMs in supporting various code-related tasks. For example, such LLMs are usually trained from all possible open-source projects in the wild, thus it is important to ensure there exist no samples in the evaluation benchmark that appear in the pre-training dataset.

Now we discuss the widely adopted datasets in the literature.

HumanEval. HumanEval [42] is released by OpenAI and is initially used to evaluate the code generation capabilities of Codex. HumanEval comprises 164 programming problems written in Python, featuring English natural text in annotations and docstrings. Each problem includes a function signature, docstring, body, and several unit tests, providing a comprehensive resource for code generation models. Importantly, these problems are hand-written to address the data leakage issue as Codex is trained on a large fraction of GitHub, which already contains solutions to problems from a variety of sources.

HumanEval-X. HumanEval-X [51] is an extended version of HumanEval, specifically designed to evaluate the multilingual code generation capabilities of CodeGeeX. Unlike HumanEval only for Python, HumanEval-X comprises 820 high-quality, manually written problems spanning Python, C++, Java, JavaScript, and Go. Each problem in HumanEval-X includes a function declaration, a description, a canonical solution, and several test cases. Besides, unlike HumanEval only for code generation, HumanEval-X is able to support both code generation and code translation tasks due to it contains parallel problems in multiple languages. In the code generation task, LLMs receive a problem description (*e.g.,* "write a factorial function") and return a solution in the chosen languages. In the code translation task, LLMs receive an implementation of a problem in a source language and return an equivalent implementation in a target language.

CodeXGLUE. CodeXGlUE is constructed by Microsoft and is initially used to evaluate the code understanding and generation capabilities of three types of LLMs, *i.e.,* encoder-only, decoder-only, and encoder-decoder models. CodeXGlUE supports a collection of 10 diversified tasks across 14 datasets and 6 programming languages. The supported tasks can be categorized into four types, including code-code (*i.e.,* clone detection, defect detection, cloze test, code completion, code repair, and code-to-code translation), text-code (natural language code search, text-to-code generation), code-text (code summarization) and text-text (documentation translation). More importantly, CodeXGlUE provides a unified platform[3] for the comparison of different LLMs, laying the foundation for the subsequent evaluation studies of LLMs and being widely used in the community. Compared with CodeXGLUE, in 2023, CrossCodeBench [184] provides a larger scale and more diverse code-related tasks (216 tasks and more than 54M data instances).

EvalPlus. To facilitate a more comprehensive discussion of the effectiveness of GPT, Liu *et al.* [185] propose EvalPlus, a benchmarking framework to evaluate the functional correctness of code generated by LLMs. This framework enhances the original HumanEval dataset by generating additional high-quality test inputs using ChatGPT. EvalPlus starts by constructing prompts with ground-truth

3. https://microsoft.github.io/CodeXGLUE/

implementations and test inputs from HumanEval, which ChatGPT uses to create rigorous test inputs. These inputs undergo type-aware mutation to produce a diverse set of new test cases. The framework emphasizes code coverage, mutant killings, and LLM sample killings to refine the quality and reduce the quantity of test suites. EvalPlus successfully augments the benchmark with these high-quality inputs, offering a more comprehensive assessment of 19 different LLMs. The results show that this augmented dataset can uncover a significant number of errors in LLM-generated code, validating the framework's effectiveness in enhancing programming benchmarks for more accurate LLM evaluation.

CrossCodeBench. CodeXGLUE is now widely used to evaluate the performance of pre-trained models of source code on various downstream tasks. Compared to CodeXGLUE, Niu *et al.* [184] introduce CrossCodeBench, a comprehensive benchmark encompassing 216 diverse code-related tasks with over 54 million data instances. The tasks are categorized into seven types: fill in the blank, translation, generation, summarization, type prediction, and question answering. Each task includes meta-information such as task type, description, detailed instructions, and example cases. For evaluation, they select a range of models: two shortcut methods, two off-the-shelf models, three fine-tuned models (including PLBART and CodeT5), and 1 supervised CodeT5-large model. The findings reveal that the supervised fine-tuned CodeT5 significantly outperforms other methods, with fine-tuned models generally excelling over off-the-shelf models. The study also highlights the superiority of domain-specific LLMs compared to cross-domain models.

ClassEval. , Du *et al.* [186] develop ClassEval, a class-level Python code generation benchmark comprising around 100 tasks, to fill a gap in evaluating LLMs in class-level code generation. ClassEval includes various software development topics, with each task designed to create classes with interdependent methods and diverse dependencies. The benchmark assesses 11 LLMs using three generation strategies: holistic, step-by-step, and composite generation. They employ the Pass@k metric for evaluating code correctness and examine models' ability to generate dependent code. Key findings reveal that LLMs perform lower in class-level generation compared to method-level benchmarks like HumanEval. GPT-4 and GPT-3.5 excel in class-level generation, especially with holistic generation, while other models perform better with step-wise strategies.

EvalGPTFix. Zhang *et al.* [187] focus on the application of black-box LLMs in SE, with a specific emphasis on APR tasks. Taking ChatGPT as an example, the authors evaluate ChatGPT's repair capabilities on their constructed APR benchmark, EvalGPTFix, which includes buggy and correct code snippets gathered from the competitive programming website Atcoder. The results demonstrate that ChatGPT can fix 109 out of 151 buggy programs with a basic prompt, outperforming CodeT5 and PLBART. The paper also investigates the impact of three types of prompts (problem description, error feedback, and bug localization) and explores the potential of improving repair performance through dialogue interactions.

5.2 Empirical Studies

Despite an emerging research area, a variety of LLMs have been proposed and have continuously achieved promising results across a variety of SE tasks. In addition to developing new techniques that address technical challenges, the LLM-based SE research field is benefiting from several empirical studies. In Section 4, when introducing specific SE tasks, we have discussed corresponding empirical studies, such as program repair [144] and vulnerability repair [12]. We also notice there exist some empirical studies that systematically explore the capabilities of LLMs from a more comprehensive perspective, such as across multiple LLMs and tasks, providing macroscopic insights into future LLM-based SE work. We summarize these empirical studies in Table 14 and discuss them in detail as follows.

As early as 2021, Lu *et al.* [41] conduct a empirical study to investigate the performance of three Code LLMs in the CodeXGLUE benchmark (discussed in Section 5.1), including the BERT-style, GPT-style, and Encoder-Decoder models. As a pioneering work in this field, this study only reports some preliminary results on a limited set of models. Furthermore, in 2022, Zeng *et al.* [189] conduct a comprehensive study of eight LLMs across seven code-related tasks using the CodeXGLUE dataset. The study covers three types of LLMs, including three encoder-based models (*i.e.,* CodeBERT, GraphCodeBERT, ContraCode), one decoder-only model (*i.e.,* CodeGPT), and four encoder-decoder LLMs (*i.e.,* CodeT5, CodeTrans, CoTexT and PLBART). The selected LLMs are evaluated on three code understanding tasks (*i.e.,* defect detection, clone detection, and code search) and four code generation tasks (*i.e.,* code summarization, code repair, code translation, and code generation). The experimental results demonstrate that LLMs tend to perform better than traditional approaches on multiple tasks, highlighting the potential of such models in addressing code-related tasks.

While the benefits of LLMs have been widely studied in NLP, limited empirical evidence is available when it comes to code-related tasks. In 2022, Mastropaolo *et al.* [188] empirically evaluate the performance of the T5 model on four code-related tasks, including (1) automatic bug-fixing, (2) injection of code mutants, (3) generation of assert statements, and (4) code summarization. They leverage a self-supervised approach to pre-train the T5 model and collect four different fine-tuning datasets for four code tasks. They then evaluate the T5 model with different training strategies and compare the performance with other baseline models. Results show that T5 performs better than existing baselines across all four tasks in its best configuration. Besides, they also notice that pre-training is beneficial, but multi-task fine-tuning does not consistently help improve performance.

Unlike Mastropaolo *et al.* [30] focusing T5 for four tasks, in 2023, Niu *et al.* [190] perform a large systematic study of 19 LLMs of source code on 13 SE tasks. The study results show that the integration of code structure and natural language features significantly boosts model performance in defect detection and mutant generation. The study also points out the effectiveness of pre-training tasks, such as Next Sentence Prediction and Future N-gram Prediction, particularly in tasks like exception type and code completion. Additionally, the research underscores the superior-

TABLE 14
A summary and comparison of existing empirical studies

Year	Study	LLMs	Task	Repository
2021	Lu *et al.* [41]	CodeBERT, CodeGPT	Clone Detection, Defect Detection, Cloze Test, Code Completion, Code Repair, Code Translation	https://github.com/microsoft/CodeXGLUE
2022	Mastropaolo *et al.* [188], [30]	T5	Program repair, Mutant generation , Assertion Generation, Code Summarization	https://github.com/antonio-mastropaolo/TransferLearning4Code
2022	Zeng *et al.* [189]	CodeBERT, CodeGPT, CodeT5, CodeTrans, ContraCode, CoTexT, GraphCodeBERT, PLBART	Defect Detection, Clone Detection, Code Search, Code Summarization, Code Repair, Code Translation, Code Generation	https://github.com/ZZR0/ISSTA22-CodeStudy
2023	Niu *et al.* [190]	CuBERT, CodeBERT, GraphCodeBERT, DOBF, JavaBERT, CodeGPT, T5-learning, PLBART, ProphetNet, CoTexT, CodeT5, SPT-Code, UniXcoder, TreeBERT, CugLM, GPT-C, C-BERT, DeepDebug, SynCoBERT	Defect Detection, Clone Detection, Exception Type, Code-to-Code Retrieval, Code Search, Code Question Answering, Code Translation, Bug Fixing, Code Completion, Mutant Generation, Assert Generation, Code Summarization, Code Generation	https://github.com/NougatCA/FineTuner

ity of transformer-based models in a multitude of tasks, including code-to-code retrieval, code search, code translation, bug-fixing, assert generation, and code summarization. Specifically for code generation, the study highlights the benefits of pre-training on code structure. The study reveals that in the task of code completion, neither natural language processing nor code structure enhancement shows a positive impact, indicating a need for specialized model configurations for this particular task.

5.3 Empirical Exploration of SE Education

In the SE community, the current research on LLMs has primarily focused on various code-related tasks and has achieved notable results. Given the powerful NL capabilities (*e.g.,* ChatGPT) and extensive programming knowledge inherent in LLMs, their interaction with students to complete programming tasks is becoming increasingly promising. The community has engaged in empirical discussions about the potential of LLMs in the SE education scenario from various dimensions. On one hand, it is exciting for LLMs to introduce innovative methods for learning and teaching, facilitating a more interactive and dynamic educational environment. On the other hand, there exists a growing concern regarding the potential misuse of these LLMs by students, such as relying excessively on automated solutions without developing critical problem-solving skills. In the following, we summarize existing studies that explore the potential and concerns of LLMs in the SE education scenario.

To explore ChatGPT's effectiveness in answering software testing questions from a textbook, in 2023, Jalil *et al.* [191] test ChatGPT with 31 questions from five chapters covering topics like software faults, Test Driven Development, and coverage criteria. The study focuses on the accuracy of ChatGPT's answers and explanations under different prompting strategies. The results show that ChatGPT provides correct or partially correct responses about 55.6% of the time and explanations 53.0% of the time. The study also examines ChatGPT's response non-determinism and finds that its self-reported confidence is not a reliable indicator of answer correctness. The study concludes that ChatGPT's responses for software testing questions need improvement in accuracy and reliability.

At the same time, to explore how well ChatGPT can perform in an introductory-level functional language programming course, Geng *et al.* [192] select a second-year undergraduate computer science course to test ChatGPT. They design two modes: unassisted and assisted. In the unassisted mode, they use ChatGPT without prior training or knowledge of engineering. In the assisted mode, they provide ChatGPT with four prompt engineering techniques: 1) paraphrasing the problem, 2) natural language hints, 3) teaching by example, and 4) providing test cases. The experiment results show that ChatGPT can achieve a grade B- and its rank in the class is 155 out of 314 students overall. They find that providing hints and manipulating the input question are the most effective techniques to improve ChatGPT's performance. They conclude that the assisted mode can significantly improve the quality of ChatGPT's response. They also conclude that ChatGPT can offer real-time support and solve problems step by step to improve students' learning experience. Besides, instructors can take advantage of ChatGPT to create exercises/homework/assignments and generate lesson plans.

When LLMs (*e.g.,* ChatGPT) are used by students during their learning processes, despite their potential, there may arise some concerns. For example, students might directly employ LLMs to complete assignments without self-thinking, leading to ineffective learning and even plagiarism concerns. To address such concerns, in 2023, Nguyen *et al.* [193] present an empirical study to investigate the feasibility of automated identification of AI-generated code snippets. They propose a CodeBERT-based classifier called GPTSniffer to detect source code written by AI The experimental results demonstrate that GPTSniffer accurately distinguishes between code generated by ChatGPT and human-written code under different experimental settings. Moreover, it outperforms two previous methods, namely GPTZero and OpenAI Text Classifier, in terms of performance. The study emphasizes the impact of training data sources, and pre-processing settings on GPTSniffer's performance.

Answer to RQ3: Overall, within the expanding arena of LLM-based SE, the community has also seen an increasing number of studies to empirically underline the evolution and nuances of existing studies from different aspects. First, datasets play a pivotal role in shaping the trajectory of research advancements. One typical trend is the construction of human-written datasets to address the data leakage issue, *e.g.,* HumanEval, and EvalGPTFix. The other typical trend is the application of multiple tasks, *e.g.,* CodeXGLUE and CrossCodeBench. Second, researchers have constructed a mass of empirical studies to explore the actual performance of LLMs from different aspects. One common empirical approach involves designing various settings for specific tasks to investigate the performance of LLMs deeply, detailed in Section 4. The other typical approach involves exploring the performance of LLMs across multiple tasks, such as T5Learning. Third, researchers observe the revolutionary impact of LLMs on SE education, and explore how they assist studies to complement programming courses. However, considering that there exist various tasks, and each task can introduce various specific LLMs, the community urgently needs more and deeper empirical studies to illuminate the landscape of LLM-based SE.

6 RQ4: Optimization and Applications

In the rapidly evolving field of SE, LLMs have emerged as pivotal roles, offering unprecedented opportunities for various code-related tasks. However, the effective deployment of LLMs in SE is not without challenges due to the inherent characteristics of LLMs, such as the record-breaking parameters making it difficult to deploy in practical scenarios. This section delves into three crucial aspects of optimizing LLMs for SE, focusing on enhancing their robustness (Section 6.1), efficiency (Section 6.2), and usability (Section 6.3).

6.1 Security Attack

As LLMs have been widely utilized and achieved state-of-the-art performances in various code-related tasks, the security of these models deserves an increasing number of attention. Similar to conventional deep learning models, LLMs have been shown to be vulnerable to adversarial attacks [194], *i.e.,* generating totally different results given two semantically-identical source code snippets. This is particularly alarming given that LLMs are deployed in some mission-critical applications, such as vulnerability detection and code search. For example, an attacker can manipulate LLMs to, (1) in vulnerability detection, output a non-vulnerable label for a piece of code that actually contains vulnerabilities, with the intention of preserving the vulnerabilities in a software system; and (2) in code search, rank the malicious code snippet high in the search results such that it can be adopted in real-world deployed software, such as autonomous driving systems. Such attacks on LLMs may cause serious incidents and have a negative societal impact. In the field of SE, there has been some preliminary exploration of the attacks on LLMs for different SE tasks.

Table 6.1 presents existing attack studies in the SE community. In the following, we summarize existing studies on attacking LLMs for SE, which mainly fall into two categories according to attack strategies.

Adversarial Attack for Code LLMs. An adversarial attack refers to an attempt to deceive models into making incorrect decisions or predictions by inserting subtle, imperceptible alterations to input data. In 2022, Yang *et al.* [194] propose ALERT, an adversarial attack for Code LLMs to ensure that the generated adversarial examples must maintain naturalness while preserving operational semantics to cater to human reviewers' needs. ALERT employs both Greedy-Attack and GA-Attack to search for adversarial examples, followed by conducting a user study to assess whether the substitutes generated by ALERT can produce adversarial examples that appear natural to human evaluators. ALERT conducts adversarial attacks on CodeBERT and GraphCodeBERT across three downstream tasks,*i.e.,* vulnerability prediction, clone detection, and authorship attribution. The experimental results demonstrate that ALERT achieves an average success rate on CodeBERT and GraphCodeBERT, surpassing the baseline model MHM by 14.07% and 18.56%, respectively. Adversarial fine-tuning of victim models using these adversarial examples enhances the robustness of CodeBERT and GraphCodeBERT, resulting in an improvement of 87.59% and 92.32%, respectively. Unlike ALERT focuses on code understanding tasks, like vulnerability detection and clone detection, Jha *et al.* [201] propose CodeAttack, the first work to perform adversarial attacks on different code generation tasks. CodeAttack attempts to generate imperceptible, effective, and minimally perturbed adversarial code samples based on code structure. CodeAttack selects representative LLMs from different categories as victim models to attack, including CodeT5, CodeBERT, GraphCodeBERT, RoBERTa, and generates adversarial samples for different tasks, including code translation, code repair, and code summarization.

Backdoor Attack for Code LLMs. A backdoor attack involves secretly making poison samples embedded with triggers (*e.g.,* a specific word) during training, so that the target model performs normally on inputs without triggers (*i.e.,* clean inputs) from ordinary users, but yields targeted erroneous behaviors on inputs with triggers (*i.e.,* poison inputs) from attackers. By using triggers to activate backdoors, attackers can manipulate the output of poisoned models and lead to severe consequences. Such attacks enable perpetrators to manipulate the output of compromised models, potentially leading to severe consequences. For example, attackers can attack vulnerability detection models to mislabel a vulnerable piece of code as non-vulnerable. As early in 2021, Schuster *et al.* [195] perform a backdoor attack against the code completion model, including GPT-2. In 2022, Wan *et al.* [196] perform the first backdoor attack for code search models, including an encoder-only Code LLM CodeBERT. The attack utilizes two types of a piece of dead code as the backdoor trigger, including a piece of fixed logging code and a grammar trigger generated by the probabilistic context-free grammar. The experimental results reveal that adding a number of backdoor samples to the training data can easily manipulate the ranking of code snippets.

TABLE 15
A summary and comparison of existing attack studies

Year	Study	LLMs	Type	Repository
2022	Yang *et al.* [194]	CodeBERT, GraphCodeBERT	Adversarial Attack	https://github.com/soarsmu/attack-pretrain-models-of-code
2021	Schuster *et al.* [195]	Pythia, GPT-2	Backdoor Attack	N.A.
2022	Wan *et al.* [196]	CodeBERT	Backdoor Attack	https://github.com/CGCL-codes/naturalcc
2023	Sun *et al.* [197]	CodeBERT, CodeT5	Backdoor Attack	https://github.com/wssun/BADCODE
2023	Li *et al.* [198]	CodeBERT	Backdoor Attack	https://github.com/LJ2lijia/CodeDetector
2023	Li *et al.* [199]	PLBART, CodeT5	Backdoor Attack	https://github.com/Lyz1213/Backdoored_PPLM
2023	Li *et al.* [200]	CodeBERT, CodeT5	Imitation Attack	N.A.
2023	Jha *et al.* [201]	CodeBERT	Adversarial Attack	https://github.com/reddy-lab-code-research/CodeAttack

However, the previously designed triggers (*i.e.,* dead code snippets) are very suspicious and can be easily identified by developers [196]. Thus, focusing on more stealthy attack, in 2023, Sun *et al.* [197] propose BADCODE, a backdoor attack approach targeting neural code search models by altering function and variable names, BADCODE mutates function and/or variable names in the original code snippet by adding extensions to existing function/variable names, such as changing "function()" to "function_aux()". BADCODE utilizes LLMs CodeBERT and CodeT5, and fine-tunes them on the CodeSearchNet dataset, using both fixed triggers and grammar triggers (PCFG) as baselines. The experimental results on CodeBERT and CodeT5 demonstrate that BADCODE can make the attack-desired code rank in the top 11%. In terms of attack performance, BADCODE outperforms the existing baseline [196] by 60%, and regarding attack stealthiness, it improves by two times based on the F1 score.

Unlike previous studies designed for specific tasks, Li *et al.* [198] propose CodePoisoner, a general backdoor attack approach for three code-related tasks (*i.e.,* defect detection, clone detection, and code repair) and three models, including an LLM-based one CodeBERT. In parallel to CodePoisoner, Li *et al.* [199] propose a task-agnostic attack approach to train backdoored models during pre-training, so as to support the multi-target downstream tasks. The attack is designed to perform on two Code LLMs (*i.e.,* PLBART and CodeT5) and two code understanding tasks (*i.e.,* defect detection, clone detection) and three code generation tasks (*i.e.,* Code2Code translation, code refinement, and Text2Code generation) from CodeXGLUE.

Imitation Attack for Code LLMs. An imitation attack refers to an attempt to create a local model (also known as an imitation model) that mimics the behavior of a target model without having access to the target's internal architecture or training data. In 2023, Li *et al.* [200] propose the first imitation attack work to explore the feasibility of extracting specialized code abilities from LLMs sing common medium-sized models. The attack employs OpenAI's text-davinci-003 as the target LLM, CodeBERT and CodeT5 as imitation LLMs for training, and considers three code-related tasks, *i.e.,* code synthesis, code translation and code summarization. The attack pipeline is four-fold, including (1) generating queries from LLMs tailored to different code-related tasks and query schemes; (2)designing a rule-based filter to select high-quality responses suitable for training; and (3) fine-tuning medium-sized backbone models with these filtered responses to train the imitation model. The experimental results reveal that training a medium-sized backbone model with a feasible number of queries can effectively replicate specialized code behaviors akin to those of the target LLMs, highlighting the potential to mimic complex code capabilities from LLMs.

6.2 LLMs Tuning

As mentioned in Section 4, the pre-training-and-fine-tuning paradigm has been a crucial means of adapting LLMs to special domains. Typically, LLMs are first pre-trained to learn the general purpose code representations on a large amount of data and then fine-tuned to targeted tasks. While fine-tuning LLMs has proven to be effective, it comes with significant computational and energy costs due to the record-breaking parameter scale. For example, it takes about 2 days to train CodeT5 with the parameter size of 220M on NVIDIA A100-40G GPUs for program repair [150], let alone more advanced LLMs with hundreds of millions or even billions of parameters. Besides, when fine-tuning LLMs on new datasets, it is inevitable to suffer from the catastrophic forgetting problem [16], *i.e.,* forgetting the knowledge learned from previous datasets. In the field of SE, there has been some preliminary exploration of the optimizations on LLMs during the fine-tuning phase.

In the following, we summarize existing studies on fine-tuning LLMs for SE, which mainly fall into two categories according to previous issues.

Efficient Parameter Fine-tuning. Such studies involve efficient training strategies to reduce the time and resource costs during fine-tuning. For example, in 2023, considering that fine-tuning LLMs incurs a large computational cost, Shi *et al.* [202] conduct an extensive study on fine-tuning LLMs like UniXcoder and GraphCodeBERT, focusing on their layer-wise pre-trained representations and encoded code knowledge. They analyze these models on the CodeSearchNet and POJ-104 datasets, discovering that lexical, syntactic, and structural properties of source code are mainly captured by different model layers, with semantic properties spanning the entire model. Despite the high computational cost of fine-tuning, basic code properties in the lower and intermediate layers are preserved. They propose an efficient fine-tuning approach called Telly-K, which involves freezing the bottom K layers' parameters.

TABLE 16
A summary and comparison of existing tuning studies

Year	Study	LLMs	Tasks	Type	Repository
2023	Shi *et al.* [202]	CodeBERT,UniXcoder	Clone detection, Code Search, Code Summarization, Code Completion, Code Generation	Parameter Tuning	https://github.com/DeepSoftwareAnalytics/Telly
2023	Gao *et al.* [203]	CodeBERT,CodeT5	Code Summarization, Vulnerability Detection, Clone detection	Continual Tuning	https://github.com/ReliableCoding/REPEAT
2023	Yuan *et al.* [16]	T5	Program Repair	Continual Tuning	https://github.com/2022CIRCLE/CIRCLE
2023	Weyssow *et al.* [204]	RoBERTa, GPT-2	API Call, API Usage Prediction	Continual Tuning	https://github.com/martin-wey/cl-code-apis
2023	Wang *et al.* [205]	CodeBERT, CodeT5, UniXcoder, PLBART, GraphCodeBERT	Code Search, Code Summarization	Continual Tuning	https://github.com/wangdeze18/Multilingual-Adapter-for-SE

This method is evaluated across five diverse downstream tasks, showing significant reductions in training parameters and time costs, with performance generally improving or remaining stable up to a certain K value before dropping for higher values.

Effective Continual Fine-tuning Such studies involve effective training strategies to address the catastrophic forgetting issue during fine-tuning. As early as 2022, Yuan *et al.* [16] propose a T5-based program repair CIRCLE equipped with continual learning ability across multiple programming languages. The experimental results demonstrate that CIRCLE not only effectively repairs multiple programming languages in continual learning settings, but also achieves state-of-the-art performance on five benchmarks with a single repair model.

Since LLMs can easily forget knowledge learned from previous datasets when learning from the new dataset, in 2023, Gao *et al.* [203] introduce REPEAT, a method to address forgetting issues in LLMs during continual learning. REPEAT incorporates representative exemplars replay, where selected diverse and informative samples from previous datasets are used to retrain the model, preventing memory loss. However, this may lead to overfitting on these samples. To counter this, they apply adaptive regularization based on the Elastic Weight Consolidation method. They evaluate CodeBERT and CodeT5 on CodeSearchNet, Big-Vul, and POJ datasets for tasks like code summarization, vulnerability detection, and clone detection. The results demonstrate that REPEAT effectively alleviates forgetting and improves generalization on unseen projects in continual code intelligence tasks.

Since previous works overlook a crucial problem that the distribution of software data changes over time, in 2023, Weyssow *et al.* [204] address the overlooked issue of the evolving distribution of software data over time, emphasizing the adaptation of pre-trained language models to this dynamic environment. They design a scenario to learn from a stream of Java programs with new APIs, distinguishing between in-distribution data for pre-training and out-of-distribution data for continual fine-tuning. Their experiments with GPT-2 and RoBERTa models on API calls and usage prediction tasks reveal that classical transfer learning alone is inadequate. They demonstrate that continual learning methods, including replay-based and regularization-based approaches, are more effective in reducing catastrophic forgetting and adapting to new data while retaining previous knowledge.

To alleviate the potentially catastrophic forgetting issue in multilingual models, in 2023, Wang *et al.* [205] propose a solution to mitigate catastrophic forgetting in multilingual models by inserting a parameter-efficient structure adapter and fine-tuning it, instead of adjusting all pre-trained model parameters. They test this approach on CodeT5, UnixCoder, CodeBERT, GraphCodeBERT, and PLBART across various tasks like code summarization and code search. The results indicate that adapter tuning is more effective than full-model fine-tuning in multilingual scenarios, particularly for low-resource languages, and even surpasses language-specific fine-tuning. This approach demonstrates the potential of using fewer samples per language for effective multilingual learning.

6.3 LLM Compressing

Once LLMs are well-trained and achieve impressive results in various SE tasks, they need to be further deployed in real-world scenarios. However, such LLMs consume hundreds of megabytes of memory and run slowly on personal devices, which results in an impediment to the wide and fluent adoption of these powerful models in the daily workflow of software developers.

To address the issue, in 2022, Shi *et al.* [206] propose Compressor to compress LLMs of source code into extremely small models with negligible performance sacrifice. In particular, given an LLM, Compressor first tries to search for smaller models with a similar architectural design to the LLM. Then, Compressor applies a genetic algorithm-based strategy to guide the process of model simplification, which means finding the promising architecture-related hyperparameters that can amplify the capacity of searched tiny models as much as possible and fit the given size limit. After that, Compressor adopts knowledge distillation to obtain a well-performing small model. In knowledge distillation, they first query the LLM and obtain its outputs, which are treated as the 'knowledge' of the LLM. Then the distilled knowledge is used to train a smaller model so that the latter can maintain (most of) the behaviors of the LLM but has a smaller size. Finally, they evaluate Compressor with two well-known LLMs (*i.e.,* CodeBERT and GraphCodeBERT) on two important tasks (*i.e.,* vulnerability prediction and clone detection). Results show that Compressor compresses LLMs to a size with 3 MB, which is 160× smaller than the original size, yet they still maintain 96.95% and 98.36% of the accuracy of the original LLMs on both vulnerability prediction and clone detection tasks. Besides, CodeBERT and GraphCodeBERT are 4.31× and 4.15× faster than the original model during inference, respectively.

Answer to RQ4: Overall, although a large amount of research effort has been devoted to how LLMs can be adapted to automate SE tasks more effectively, the literature has also seen some orthogonal works discussing the unique challenges encountered during the process of adaption. First, LLMs can be attacked to generate vulnerable code snippets or return the wrong classifications with different attack strategies, such as adversarial attack, backdoor attack and imitation attack. Second, considering the huge parameter scale of LLMs, it is crucial to design tuning strategies to adapt such LLMs in SE tasks, such as parameter and continual fine-tuning. Third, after LLMs are well-trained, deploying such LLMs within the development workflow necessitates further consideration of factors such as inference time and resource expenditure.

7 Challenges and Opportunities

Our survey reveals that advances in LLMs for SE have a significant impact on both academia and industry. Despite achieving promising progress, there are still numerous challenges that need to be addressed, providing abundant opportunities for further research and applications. We discuss the following important practical guidelines for future LLM-based SE research.

Trade-off between Effectiveness and Model Size. As discussed in Section 3.1, the community tends to introduce the growing size of models, resulting in the recent emergence of LLMs with record-breaking parameters, *e.g.,* from 117M parameters of GPT-1 to 175B parameters of GPT-3. This trend is reasonable, as existing studies have demonstrated that larger models usually yield better performance, *e.g.,* code generation [20] and program repair [11], highlighting the significance of the number of model parameters in performance enhancement. However, models with such a large number of parameters may raise some concerns during training and deployment. First, it is extremely time-consuming and resource-intensive to train such LLMs, especially since the GPU resources required are unaffordable for most researchers in academia and even in the corporate world. For example, *e.g.,* it takes 12 days to train the medium-sized CodeT5-base model (220M parameters) with 16 NVIDIA A100 GPUs. Second, it is difficult to deploy such LLMs in real-world scenarios, as they consume hundreds of megabytes of memory and disk space, *e.g.,* Code Llama-34B takes about 63GB of disk space[4]. Thus, LLMs may run slowly on personal devices, and cannot be deployed on resource-constrained or real-time terminal devices, such as mobile devices and autonomous driving.

We recommend that future work can be conducted in the following three directions. First, it is promising to optimize the size of LLMs without significantly compromising their performance, such as model pruning, quantization, and knowledge distillation [206]. Second, researchers can develop lightweight models tailored for specific applications or techniques for distributed computing, enabling parts of a model to run on different devices.

Exploring Task-oriented Domain LLMs. As discussed in Section 3.3, although LLMs are increasingly being applied in the SE community, the majority of these models are designed with general-purpose training strategies to support multiple downstream tasks. However, there are some concerns with the adoption of such general LLMs. First, LLMs need to learn general knowledge about natural language and different programming languages from extremely large datasets, leading to the model's vast size. Second, LLMs contain a vast array of knowledge, much of which is irrelevant to specific tasks. Third, their pre-training tasks are universal, creating a certain gap with the downstream tasks. For example, existing LLMs are usually trained with a given code snippet and the corresponding description, which can hardly be exploited to learn the code change patterns for some code-editing tasks that involve two code snippets. Thus, employing existing LLMs for such editing tasks will inevitably lead to inconsistent inputs and objectives between pre-training and fine-tuning.

We recommend future work to explore domain LLMs for specific tasks. For example, researchers can design LLMs specifically for unit testing scenarios (*e.g.,* test generation and update), focusing solely on learning domain knowledge relevant to unit testing with specific pre-training objectives.

Clean Evaluation Datasets. LLMs have been gaining increasing attention and demonstrated promising performance across a variety of SE tasks, such as program repair and code generation. However, there exists a potential risk of data leakage since such LLMS are usually trained with all possible public repositories in the wild. As mentioned in Section 5.1, researchers [18] find that some code snippets in Defects4J, the widely-adopted benchmark in the program repair literature, are leaked in CodeSearchNet, which is the most popular dataset to train LLMs, *e.g.,* CodeT5, CodeBERT, and UniXcoder. More importantly, the greater concern arises from the black-box LLMs developed by commercial companies, which often outperform open-source LLMs. It is difficult to ensure whether or not the evaluation dataset has been seen by such LLMs during training as these LLMs are usually closed-source with unknown specific training details, *e.g.,* pre-training datasets. For example, ChatGPT, the latest black-box LLM, has been investigated by numerous recent research studies and has shown impressive performance in various code-related tasks. However, researchers find that ChatGPT can directly provide complete descriptions and the corresponding solution by simply providing it with the number of a programming problem in LeetCode. Considering the fact there exist a quite number of black-box LLMs for which no architecture or training data information has been released. The data

4. The model size is according to Code Llama's checkpoint implemented by HuggingFace in https://huggingface.co/codellama/CodeLlama-34b-hf

leaking on such LLMs is a significant concern when it comes to evaluating their performance in some code-related tasks in the SE community.

We recommend that future work can be carried out from two perspectives. First, the construction of clean datasets is crucial to ensure they have not been contaminated by LLMs. Three potential sources can be utilized for this purpose, (1) The first source comes from manually written programs. Similar to HumanEval [42], researchers can create evaluation programs by hand, so as to provide a unique and uncontaminated benchmark. (2) The second source is the most recently released programs. Similar to EvalGPTFix [187], researchers can seek out the latest programs, such as those from recent competitions or coding challenge websites, as they often contain fresh and diverse problems that are less likely to have been included in the training sets of current LLMs. (3) The third source is closed-source projects. Researchers can evaluate LLMs with some internal projects within the company, which are not previously exposed to public repositories, thereby providing a more authentic evaluation of the models' capabilities in real-world scenarios. In addition to clean dataset construction, it is essential to design some techniques to verify whether LLMs exhibit any form of data memorization for a given testing sample, such as detecting if a model is simply recalling information from its training data rather than genuinely understanding or solving a problem.

Application on More SE Tasks. As discussed in Section 4, we observe a pronounced emphasis on the application of LLMs in software development, testing, and maintenance. These areas have undoubtedly benefited from the capabilities of LLMs, leading to impressive performance in code completion [85], fault detection [97], program repair [18] and so on. Despite its success in these tasks, there are other tasks that have been popularly studied with traditional techniques or machine/deep learning techniques. For example, the current application of LLMs in requirements engineering, software design, and software management remains relatively sparse.

We suggest that future work should concentrate on two aspects to broaden the scope of LLM applications for more SE tasks. First, for complex tasks, integrating LLMs into existing research workflows as a component rather than developing end-to-end solutions appears more pragmatic. For example, existing regression test case prioritization approaches tend to calculate the similarity of selected test cases and candidates based on code coverage, and may fail to consider the semantic similarity between different test cases. Researchers can boost existing similarity-based prioritization techniques via LLMs, which contain generic knowledge pre-trained with millions of code snippets from open-source projects, and provide accurate semantic information for test code. This integration strategy leverages the strengths of LLMs in augmenting and enhancing current approaches, particularly in areas where conventional approaches have reached a plateau in terms of performance. By combining LLMs with established techniques, we can achieve more robust and efficient outcomes in complex scenarios. Second, for rare SE tasks where LLMs may lack rich knowledge, it is promising to design domain-specific LLMs tailored to these underrepresented areas. For example, a variety of medium-sized LLMs are trained with CodeSearchNet without test cases, thus failing to benefit tasks such as unit test generation.

Multi-task and Multi-dimensional Benchmarks. As the development of LLMs progresses, it is crucial to acquire and prepare benchmarks that are more diverse, comprehensive, and realistic to reflect capabilities in real-world scenarios. However, existing datasets may face issues related to data bias, deficiency, quality, and credibility. First, most well-constructed benchmarks are concentrated on some widely-investigated SE tasks (*e.g.,* HumanEval and CoderEval in code generation), while lacking in other less-explored tasks like unit test generation. Second, the majority of existing evaluation dimensions focus primarily on performance metrics (*e.g.,* Pass-1 in code generation), paying little attention to other critical attributes of LLMs, such as time efficiency and robustness.

We recommend that future work can be directed in the following two areas. First, to address the limitation in task scope, researchers can build diversified benchmarks to evaluate LLMs in emerging fields, such as unit test generation. Second, to address the limitations in evaluation dimensions, new metrics and specialized benchmarks should be introduced to assess some crucial aspects, such as robustness and efficiency.

Beyond Text-based LLMs for Vision-based SE. In the realm of SE, a predominant focus has been observed on LLMs that process text-based inputs, *i.e.,* natural language and source code, significantly benefiting a multitude of code-related tasks. However, alternative forms of input play an equally crucial role in SE tasks, notably images in mobile applications. For example, the graphical user interface has emerged as a crucial component of mobile applications, attracting substantial research attention in the area of GUI testing. Recently, with advancements in computer vision technologies, vision-based GUI testing approaches have been developed and have shown promising progress.

We suggest that future works focus on the utilization of multi-modal LLMs in version-based SE tasks. For example, it is potential to combine both text and image understanding capabilities of multi-modal LLMs to better capture the syntax and semantic information about source code, test scripts, and test reports in GUI testing, so as to benefit several tasks, *e.g.,* GUI test generation, GUI test record, and replay.

Explainable LLM-based Research. Existing LLMs usually address SE tasks in a black-box manner due to the inherent limitations of DL and the vast parameters of LLMs. The developers are unaware of why LLMs generate the predictions, thus unsure about the reliability of these results, hindering the adoption of LLM in practice. In the literature, a majority of studies focus on improving the performance of pre-defined metrics (*e.g.,* accuracy and precision), while minor focus on improving the explainability of such LLMs. Traditional rule-based SE approaches rely on pre-defined rules and logic, which makes them more interpretable and offers more transparency.

In the future, advanced explainable techniques can be considered to make the predictions of LLMs more practical, explainable, and actionable. We suggest that future work should concentrate on two aspects to support the understanding of LLMs for SE research. First, it is possible to incorporate XAI techniques to elucidate the decision-making process of LLMs, such as designing strategies to trace back the decision process to specific data points or model components. Second, developing hybrid frameworks that combine the interpretability of traditional rule-based appraoches with the predictive power of LLMs, could help in bridging the gap between traditional SE approaches and advaned LLMs, providing a balance between transparency and performance.

8 Conclusion

Large language models (LLMs) are bringing significant changes to the software engineering (SE) field, with their ability to handle complex code-related tasks poised to fundamentally reshape numerous SE practices and approaches. In this paper, we provide a comprehensive survey of existing LLM-based SE studies from both the LLM and SE perspectives. We summarize 30 representative LLMs of Code and discuss their distinct architectures, pre-training objectives, downstream tasks, and open science. We illustrate the wide range of SE tasks where LLMs have been applied, involving 155 relevant studies for 43 code-related tasks across four crucial SE phases. We discuss the benchmarks, empirical studies, and the exploration of SE education in the LLM-based SE community. We highlight several crucial aspects of the optimization and application for SE research, including security attacks, model tuning, and model compressing. Finally, we point out several challenges (such as the data leakage issue) and provide possible directions for future study. Overall, our work serves as a roadmap for promising future research, and is valuable to both researchers and practitioners, assisting them in leveraging LLMs to improve existing SE practices.

Acknowledgments

This work is supported partially by the National Natural Science Foundation of China (61932012, 62141215).

References

[1] J. Biolchini, P. G. Mian, A. C. C. Natali, and G. H. Travassos, "Systematic review in software engineering," *System engineering and computer science department COPPE/UFRJ, Technical Report ES*, vol. 679, no. 05, p. 45, 2005.

[2] M. V. Zelkowitz, "Perspectives in software engineering," *ACM Computing Surveys (CSUR)*, vol. 10, no. 2, pp. 197–216, 1978.

[3] Y. Yang, X. Xia, D. Lo, and J. Grundy, "A survey on deep learning for software engineering," *ACM Computing Surveys (CSUR)*, vol. 54, no. 10s, pp. 1–73, 2022.

[4] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language model: Survey, landscape, and vision," *arXiv preprint arXiv:2307.07221*, 2023.

[5] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[6] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 5485–5551, 2020.

[7] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[8] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536–1547.

[9] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP'21)*, 2021, pp. 8696–8708.

[10] OpenAI, "Chatgpt: Optimizing language models for dialogue," https://openai.com/blog/chatgpt, 2023.

[11] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*, 2023.

[12] Q. Zhang, C. Fang, B. Yu, W. Sun, T. Zhang, and Z. Chen, "Pre-trained model-based automated software vulnerability repair: How far are we?" *IEEE Transactions on Dependable and Secure Computing*, 2023.

[13] C. S. Xia and L. Zhang, "Less training, more repairing please: revisiting automated program repair via zero-shot learning," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 959–971.

[14] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," *arXiv preprint arXiv:2204.05999*, 2022.

[15] 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," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. Association for Computing Machinery, 2023, p. 1282–1294.

[16] 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.

[17] 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.

[18] Q. Zhang, C. Fang, T. Zhang, B. Yu, W. Sun, and Z. Chen, "Gamma: Revisiting template-based automated program repair via mask prediction," *arXiv preprint arXiv:2309.09308*, 2023.

[19] C. Watson, N. Cooper, D. N. Palacio, K. Moran, and D. Poshyvanyk, "A systematic literature review on the use of deep learning in software engineering research," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 2, pp. 1–58, 2022.

[20] D. Zan, B. Chen, F. Zhang, D. Lu, B. Wu, B. Guan, W. Yongji, and J.-G. Lou, "Large language models meet nl2code: A survey," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2023, pp. 7443–7464.

[21] S. Wang, L. Huang, A. Gao, J. Ge, T. Zhang, H. Feng, I. Satyarth, M. Li, H. Zhang, and V. Ng, "Machine/deep learning for software engineering: A systematic literature review," *IEEE Transactions on Software Engineering (TSE)*, 2022.

[22] H. Zhang, M. A. Babar, and P. Tell, "Identifying relevant studies in software engineering," *Information and Software Technology (IST)*, vol. 53, no. 6, pp. 625–637, 2011.

[23] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[24] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang, "Deep learning library testing via effective model generation," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 788–799.

[25] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A survey of learning-based automated program repair," *ACM Trans. Softw. Eng. Methodol.*, 2023.

[26] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," in *International conference on machine learning*. PMLR, 2020, pp. 5110–5121.

[27] H. Zhang, S. Lu, Z. Li, Z. Jin, L. Ma, Y. Liu, and G. Li, "Codebert-attack: Adversarial attack against source code deep learning models via pre-trained model," *Journal of Software: Evolution and Process*, p. e2571.

[28] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[29] C. B. Clement, D. Drain, J. Timcheck, A. Svyatkovskiy, and N. Sundaresan, "Pymt5: multi-mode translation of natural language and python code with transformers," *arXiv preprint arXiv:2010.03150*, 2020.

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

[31] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," *arXiv preprint arXiv:2103.06333*, 2021.

[32] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2022, pp. 7212–7225.

[33] C. Niu, C. Li, V. Ng, J. Ge, L. Huang, and B. Luo, "Spt-code: sequence-to-sequence pre-training for learning source code representations," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2006–2018.

[34] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi, "Coderl: Mastering code generation through pretrained models and deep reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 35, pp. 21 314–21 328, 2022.

[35] J. Zhang, S. Panthaplackel, P. Nie, J. J. Li, and M. Gligoric, "Coditt5: Pretraining for source code and natural language editing," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[36] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[37] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. Hoi, "Codet5+: Open code large language models for code understanding and generation," *arXiv preprint arXiv:2305.07922*, 2023.

[38] S. Chandel, C. B. Clement, G. Serrato, and N. Sundaresan, "Training and evaluating a jupyter notebook data science assistant," *arXiv preprint arXiv:2201.12901*, 2022.

[39] Y. Chai, S. Wang, C. Pang, Y. Sun, H. Tian, and H. Wu, "Erniecode: Beyond english-centric cross-lingual pretraining for programming languages," *arXiv preprint arXiv:2212.06742*, 2022.

[40] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1433–1443.

[41] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," in *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, J. Vanschoren and S. Yeung, Eds., 2021.

[42] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[43] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 1–10.

[44] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[45] D. Zan, B. Chen, D. Yang, Z. Lin, M. Kim, B. Guan, Y. Wang, W. Chen, and J.-G. Lou, "Cert: Continual pre-training on sketches for library-oriented code generation," *arXiv preprint arXiv:2206.06888*, 2022.

[46] L. B. Allal, R. Li, D. Kocetkov, C. Mou, C. Akiki, C. M. Ferrandis, N. Muennighoff, M. Mishra, A. Gu, M. Dey *et al.*, "Santacoder: don't reach for the stars!" *arXiv preprint arXiv:2301.03988*, 2023.

[47] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[48] F. Christopoulou, G. Lampouras, M. Gritta, G. Zhang, Y. Guo, Z. Li, Q. Zhang, M. Xiao, B. Shen, L. Li *et al.*, "Pangu-coder: Program synthesis with function-level language modeling," *arXiv preprint arXiv:2207.11280*, 2022.

[49] B. Shen, J. Zhang, T. Chen, D. Zan, B. Geng, A. Fu, M. Zeng, A. Yu, J. Ji, J. Zhao *et al.*, "Pangu-coder2: Boosting large language models for code with ranking feedback," *arXiv preprint arXiv:2307.14936*, 2023.

[50] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, "Palm: Scaling language modeling with pathways," *arXiv preprint arXiv:2204.02311*, 2022.

[51] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, Z. Wang, L. Shen, A. Wang, Y. Li *et al.*, "Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x," *arXiv preprint arXiv:2303.17568*, 2023.

[52] E. Nijkamp, H. Hayashi, C. Xiong, S. Savarese, and Y. Zhou, "Codegen2: Lessons for training llms on programming and natural languages," *arXiv preprint arXiv:2305.02309*, 2023.

[53] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[54] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[55] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, "Electra: Pre-training text encoders as discriminators rather than generators," *arXiv preprint arXiv:2003.10555*, 2020.

[56] D. Xie, B. Yoo, N. Jiang, M. Kim, L. Tan, X. Zhang, and J. S. Lee, "Impact of large language models on generating software specifications," *arXiv preprint arXiv:2306.03324*, 2023.

[57] M. R. Hasan, J. Li, I. Ahmed, and H. Bagheri, "Automated repair of declarative software specifications in the era of large language models," *arXiv preprint arXiv:2310.12425*, 2023.

[58] T. Hey, J. Keim, A. Koziolek, and W. F. Tichy, "Norbert: Transfer learning for requirements classification," in *2020 IEEE 28th International Requirements Engineering Conference (RE)*, 2020, pp. 169–179.

[59] A. F. Subahi, "Bert-based approach for greening software requirements engineering through non-functional requirements," *IEEE Access*, 2023.

[60] L. Han, Q. Zhou, and T. Li, "Improving requirements classification models based on explainable requirements concerns," in *2023 IEEE 31st International Requirements Engineering Conference Workshops (REW)*. IEEE, 2023, pp. 95–101.

[61] K. Rahman, A. Ghani, A. Alzahrani, M. U. Tariq, and A. U. Rahman, "Pre-trained model-based nfr classification: Overcoming limited data challenges," *IEEE Access*, 2023.

[62] M. A. Khan, M. S. Khan, I. Khan, S. Ahmad, and S. Huda, "Non functional requirements identification and classification using transfer learning model," *IEEE Access*, 2023.

[63] P. Brie, N. Burny, A. Sluÿters, and J. Vanderdonckt, "Evaluating a large language model on searching for gui layouts," *Proceedings of the ACM on Human-Computer Interaction*, vol. 7, no. EICS, pp. 1–37, 2023.

[64] J. Li, Y. Zhao, Y. Li, G. Li, and Z. Jin, "Acecoder: Utilizing existing code to enhance code generation," *arXiv preprint arXiv:2303.17780*, 2023.

[65] K. Zhang, Z. Li, J. Li, G. Li, and Z. Jin, "Self-edit: Fault-aware code editor for code generation," *arXiv preprint arXiv:2305.04087*, 2023.

[66] G. Yang, Y. Zhou, X. Chen, X. Zhang, Y. Xu, T. Han, and T. Chen, "A syntax-guided multi-task learning approach for turducken-style code generation," *arXiv preprint arXiv:2303.05061*, 2023.

[67] C. Liu, X. Bao, H. Zhang, N. Zhang, H. Hu, X. Zhang, and M. Yan, "Improving chatgpt prompt for code generation," *arXiv preprint arXiv:2305.08360*, 2023.

[68] A. Ni, S. Iyer, D. Radev, V. Stoyanov, W.-t. Yih, S. Wang, and X. V. Lin, "Lever: Learning to verify language-to-code generation with execution," in *International Conference on Machine Learning*. PMLR, 2023, pp. 26 106–26 128.

[69] B. Kou, S. Chen, Z. Wang, L. Ma, and T. Zhang, "Is model attention aligned with human attention? an empirical study on large language models for code generation," *arXiv preprint arXiv:2306.01220*, 2023.

[70] F. Mu, L. Shi, S. Wang, Z. Yu, B. Zhang, C. Wang, S. Liu, and Q. Wang, "Clarifygpt: Empowering llm-based code generation with intention clarification," *arXiv preprint arXiv:2310.10996*, 2023.

[71] Y. Dong, X. Jiang, Z. Jin, and G. Li, "Self-collaboration code generation via chatgpt," *arXiv preprint arXiv:2304.07590*, 2023.

[72] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "Llm is like a box of chocolates: the non-determinism of chatgpt in code generation," *arXiv preprint arXiv:2308.02828*, 2023.

[73] X. Li, Y. Gong, Y. Shen, X. Qiu, H. Zhang, B. Yao, W. Qi, D. Jiang, W. Chen, and N. Duan, "Coderetriever: A large scale contrastive pre-training method for code search," in *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, 2022, pp. 2898–2910.

[74] A. Saieva, S. Chakraborty, and G. Kaiser, "On contrastive learning of semantic similarity forcode to code search," *arXiv preprint arXiv:2305.03843*, 2023.

[75] N. Sorokin, D. Abulkhanov, S. Nikolenko, and V. Malykh, "Cct-code: Cross-consistency training for multilingual clone detection and code search," *arXiv preprint arXiv:2305.11626*, 2023.

[76] P. Salza, C. Schwizer, J. Gu, and H. C. Gall, "On the effectiveness of transfer learning for code search," *IEEE Transactions on Software Engineering*, 2022.

[77] J. Zhang, P. Nie, J. J. Li, and M. Gligoric, "Multilingual code co-evolution using large language models," *arXiv preprint arXiv:2307.14991*, 2023.

[78] R. Baltaji, S. Pujar, L. Mandel, M. Hirzel, L. Buratti, and L. Varshney, "Learning transfers over several programming languages," *arXiv preprint arXiv:2310.16937*, 2023.

[79] A. Mastropaolo, E. Aghajani, L. Pascarella, and G. Bavota, "An empirical study on code comment completion," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 159–170.

[80] S. A. Rukmono, L. Ochoa, and M. R. Chaudron, "Achieving high-level software component summarization via hierarchical chain-of-thought prompting and static code analysis," in *2023 IEEE International Conference on Data and Software Engineering (ICoDSE)*, 2023, pp. 7–12.

[81] Z. Sun, J. M. Zhang, M. Harman, M. Papadakis, and L. Zhang, "Automatic testing and improvement of machine translation," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 974–985.

[82] M. Ciniselli, N. Cooper, L. Pascarella, A. Mastropaolo, E. Aghajani, D. Poshyvanyk, M. Di Penta, and G. Bavota, "An empirical study on the usage of transformer models for code completion," *IEEE Transactions on Software Engineering*, vol. 48, no. 12, pp. 4818–4837, 2021.

[83] T. van Dam, M. Izadi, and A. van Deursen, "Enriching source code with contextual data for code completion models: An empirical study," *arXiv preprint arXiv:2304.12269*, 2023.

[84] Z. Li, C. Wang, Z. Liu, H. Wang, D. Chen, S. Wang, and C. Gao, "Cctest: Testing and repairing code completion systems," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1238–1250.

[85] M. Ciniselli, N. Cooper, L. Pascarella, D. Poshyvanyk, M. Di Penta, and G. Bavota, "An empirical study on the usage of bert models for code completion," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 108–119.

[86] 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.

[87] N. Jain, S. Vaidyanath, A. Iyer, N. Natarajan, S. Parthasarathy, S. Rajamani, and R. Sharma, "Jigsaw: Large language models meet program synthesis," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1219–1231.

[88] V. Liventsev, A. Grishina, A. Härmä, and L. Moonen, "Fully autonomous programming with large language models," *arXiv preprint arXiv:2304.10423*, 2023.

[89] P. Gupta, A. Khare, Y. Bajpai, S. Chakraborty, S. Gulwani, A. Kanade, A. Radhakrishna, G. Soares, and A. Tiwari, "Grace: Generation using associated code edits," *arXiv preprint arXiv:2305.14129*, 2023.

[90] A. Ciborowska and K. Damevski, "Fast changeset-based bug localization with bert," in *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)*, 2022, pp. 946–957.

[91] A. M. Mohsen, H. Hassan, K. Wassif, R. Moawad, and S. Makady, "Enhancing bug localization using phase-based approach," *IEEE Access*, 2023.

[92] Y. Wu, Z. Li, J. M. Zhang, M. Papadakis, M. Harman, and Y. Liu, "Large language models in fault localisation," *arXiv preprint arXiv:2308.15276*, 2023.

[93] S. Kang, G. An, and S. Yoo, "A preliminary evaluation of llm-based fault localization," *arXiv preprint arXiv:2308.05487*, 2023.

[94] Z. Zhu, Y. Wang, and Y. Li, "Trobo: A novel deep transfer model for enhancing cross-project bug localization," in *Knowledge Science, Engineering and Management: 14th International Conference, KSEM 2021, Tokyo, Japan, August 14–16, 2021, Proceedings, Part I*. Springer, 2021, pp. 529–541.

[95] X. Xu, Z. Zhang, S. Feng, Y. Ye, Z. Su, N. Jiang, S. Cheng, L. Tan, and X. Zhang, "Lmpa: Improving decompilation by synergy of large language model and program analysis," *arXiv preprint arXiv:2306.02546*, 2023.

[96] W. K. Wong, H. Wang, Z. Li, Z. Liu, S. Wang, Q. Tang, S. Nie, and S. Wu, "Refining decompiled c code with large language models," *arXiv preprint arXiv:2310.06530*, 2023.

[97] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 608–620.

[98] B. Steenhoek, M. M. Rahman, R. Jiles, and W. Le, "An empirical study of deep learning models for vulnerability detection," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2237–2248.

[99] H. Hanif and S. Maffeis, "Vulberta: Simplified source code pre-training for vulnerability detection," in *2022 International joint conference on neural networks (IJCNN)*. IEEE, 2022, pp. 1–8.

[100] M. Alqarni and A. Azim, "Low level source code vulnerability detection using advanced bert language model," in *Proceedings of the Canadian Conference on Artificial Intelligence-Https://caiac. pubpub. org/pub/gdhb8oq4 (may 27 2022)*, 2022.

[101] B. Ahmad, B. Tan, R. Karri, and H. Pearce, "Flag: Finding line anomalies (in code) with generative ai," *arXiv preprint arXiv:2306.12643*, 2023.

[102] D. Noever, "Can large language models find and fix vulnerable software?" *arXiv preprint arXiv:2308.10345*, 2023.

[103] C. Zhang, H. Liu, J. Zeng, K. Yang, Y. Li, and H. Li, "Prompt-enhanced software vulnerability detection using chatgpt," *arXiv preprint arXiv:2308.12697*, 2023.

[104] M. C. Tol and B. Sunar, "Zeroleak: Using llms for scalable and cost effective side-channel patching," *arXiv preprint arXiv:2308.13062*, 2023.

[105] 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.

[106] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *arXiv preprint arXiv:2302.06527*, 2023.

[107] 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.

[108] 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.

[109] Y. Tang, Z. Liu, Z. Zhou, and X. Luo, "Chatgpt vs sbst: A comparative assessment of unit test suite generation," *arXiv preprint arXiv:2307.00588*, 2023.

[110] 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.

[111] E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri, "Toga: A neural method for test oracle generation," in *Proceedings of the*

44th International Conference on Software Engineering, 2022, pp. 2130–2141.

[112] R. Pan, T. A. Ghaleb, and L. Briand, "Ltm: Scalable and black-box similarity-based test suite minimization based on language models," *arXiv preprint arXiv:2304.01397*, 2023.

[113] A. Dakhama, K. Even-Mendoza, W. B. Langdon, H. D. Menendez, and J. Petke, "Searchgem5: Towards reliable gem5 with search based software testing and large language models," in *15th Symposium on Search Based Software Engineering (SSBSE): Lecture Notes in Computer Science*. Springer, 2023.

[114] J. Hu, Q. Zhang, and H. Yin, "Augmenting greybox fuzzing with generative ai," *arXiv preprint arXiv:2306.06782*, 2023.

[115] C. Yang, Y. Deng, R. Lu, J. Yao, J. Liu, R. Jabbarvand, and L. Zhang, "White-box compiler fuzzing empowered by large language models," *arXiv preprint arXiv:2310.15991*, 2023.

[116] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large language model guided protocol fuzzing." NDSS, 2024.

[117] C. Zhang, M. Bai, Y. Zheng, Y. Li, X. Xie, Y. Li, W. Ma, L. Sun, and Y. Liu, "Understanding large language model based fuzz driver generation," *arXiv preprint arXiv:2307.12469*, 2023.

[118] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*, 2023.

[119] C. S. Xia, M. Paltenghi, J. L. Tian, M. Pradel, and L. Zhang, "Universal fuzzing via large language models," *arXiv preprint arXiv:2308.04748*, 2023.

[120] 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, 2023, pp. 2339–2356.

[121] 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.

[122] 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.

[123] A. Happe and J. Cito, "Getting pwn'd by ai: Penetration testing with large language models," *arXiv preprint arXiv:2308.00121*, 2023.

[124] G. Deng, Y. Liu, V. Mayoral-Vilches, P. Liu, Y. Li, Y. Xu, T. Zhang, Y. Liu, M. Pinzger, and S. Rass, "Pentestgpt: An llm-empowered automatic penetration testing tool," *arXiv preprint arXiv:2308.06782*, 2023.

[125] V. Vikram, C. Lemieux, and R. Padhye, "Can large language models write good property-based tests?" *arXiv preprint arXiv:2307.04346*, 2023.

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

[127] A. R. Ibrahimzada, Y. Chen, R. Rong, and R. Jabbarvand, "Automated bug generation in the era of large language models," *arXiv preprint arXiv:2310.02407*, 2023.

[128] Y. Nong, Y. Ou, M. Pradel, F. Chen, and H. Cai, "Vulgen: Realistic vulnerability generation via pattern mining and deep learning," in *IEEE/ACM 45th International Conference on Software Engineering(ICSE) https://www. software-lab. org/publications/icse2023_VulGen. pdf*, 2023.

[129] 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.

[130] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, X. Che, D. Wang, and Q. Wang, "Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions," *arXiv preprint arXiv:2310.15780*, 2023.

[131] Z. Sun, J. M. Zhang, Y. Xiong, M. Harman, M. Papadakis, and L. Zhang, "Improving machine translation systems via isotopic replacement," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1181–1192.

[132] P. He, C. Meister, and Z. Su, "Structure-invariant testing for machine translation," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 961–973.

[133] S. Gupta, P. He, C. Meister, and Z. Su, "Machine translation testing via pathological invariance," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 863–875.

[134] B. Yu, Y. Hu, Q. Mang, W. Hu, and P. He, "Automated testing and improvement of named entity recognition systems," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 883–894.

[135] W. Wang, J.-t. Huang, W. Wu, J. Zhang, Y. Huang, S. Li, P. He, and M. R. Lyu, "Mttm: Metamorphic testing for textual content moderation software," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2387–2399.

[136] Z. Liu, Y. Feng, and Z. Chen, "Dialtest: automated testing for recurrent-neural-network-driven dialogue systems," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 115–126.

[137] Z. Liu, Y. Feng, Y. Yin, J. Sun, Z. Chen, and B. Xu, "Qatest: A uniform fuzzing framework for question answering systems," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[138] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1161–1173.

[139] Y. Li, S. Wang, and T. N. Nguyen, "Dear: A novel deep learning-based approach for automated program repair," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 511–523.

[140] 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.

[141] B. Berabi, J. He, V. Raychev, and M. Vechev, "Tfix: Learning to fix coding errors with a text-to-text transformer," in *International Conference on Machine Learning*. PMLR, 2021, pp. 780–791.

[142] Y. Wei, C. S. Xia, and L. Zhang, "Copiloting the copilots: Fusing large language models with completion engines for automated program repair," *arXiv preprint arXiv:2309.00608*, 2023.

[143] C. S. Xia, Y. Ding, and L. Zhang, "Revisiting the plastic surgery hypothesis via large language models," *arXiv preprint arXiv:2303.10494*, 2023.

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

[145] D. Drain, C. Wu, A. Svyatkovskiy, and N. Sundaresan, "Generating bug-fixes using pretrained transformers," in *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*, 2021, pp. 1–8.

[146] 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.

[147] 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.

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

[149] 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.

[150] W. Wang, Y. Wang, S. Joty, and S. C. Hoi, "Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair," *arXiv preprint arXiv:2309.06057*, 2023.

[151] 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," *arXiv preprint arXiv:2306.01394*, 2023.

[152] 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.

[153] H. Tian, K. Liu, A. K. Kaboré, A. Koyuncu, L. Li, J. Klein, and T. F. Bissyandé, "Evaluating representation learning of code changes for predicting patch correctness in program repair," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 981–992.

[154] H. Tian, X. Tang, A. Habib, S. Wang, K. Liu, X. Xia, J. Klein, and T. F. Bissyandé, "Is this change the answer to that problem? correlating descriptions of bug and code changes for evaluating patch correctness," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[155] H. Tian, K. Liu, Y. Li, A. K. Kaboré, A. Koyuncu, A. Habib, L. Li, J. Wen, J. Klein, and T. F. Bissyandé, "The best of both worlds: Combining learned embeddings with engineered features for accurate prediction of correct patches," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, pp. 1–34, 2023.

[156] Q. Zhang, C. Fang, W. Sun, Y. Liu, T. He, X. Hao, and Z. Chen, "Boosting automated patch correctness prediction via pre-trained language model," *arXiv preprint arXiv:2301.12453*, 2023.

[157] X. Zhou, B. Xu, K. Kim, D. Han, T. Le-Cong, J. He, B. Le, and D. Lo, "Patchzero: Zero-shot automatic patch correctness assessment," *arXiv preprint arXiv:2303.00202*, 2023.

[158] T. H. Jung, "Commitbert: Commit message generation using pre-trained programming language model," in *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, 2021, pp. 26–33.

[159] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu *et al.*, "Automating code review activities by large-scale pre-training," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1035–1047.

[160] L. Li, L. Yang, H. Jiang, J. Yan, T. Luo, Z. Hua, G. Liang, and C. Zuo, "Auger: automatically generating review comments with pre-training models," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1009–1021.

[161] R. Tufano, S. Masiero, A. Mastropaolo, L. Pascarella, D. Poshyvanyk, and G. Bavota, "Using pre-trained models to boost code review automation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2291–2302.

[162] Q. Guo, J. Cao, X. Xie, S. Liu, X. Li, B. Chen, and X. Peng, "Exploring the potential of chatgpt in automated code refinement: An empirical study," *arXiv preprint arXiv:2309.08221*, 2023.

[163] T. Zhang, I. C. Irsan, F. Thung, and D. Lo, "Cupid: Leveraging chatgpt for more accurate duplicate bug report detection," *arXiv preprint arXiv:2308.10022*, 2023.

[164] L. Plein and T. F. Bissyandé, "Can llms demystify bug reports?" *arXiv preprint arXiv:2310.06310*, 2023.

[165] S. Kang, J. Yoon, and S. Yoo, "Large language models are few-shot testers: Exploring llm-based general bug reproduction," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2312–2323.

[166] S. Feng and C. Chen, "Prompting is all your need: Automated android bug replay with large language models," *arXiv preprint arXiv:2306.01987*, 2023.

[167] X. Hu, Z. Liu, X. Xia, Z. Liu, T. Xu, and X. Yang, "Identify and update test cases when production code changes: A transformer-based approach," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 1111–1122.

[168] Y. Xie, J. Lin, H. Dong, L. Zhang, and Z. Wu, "A survey of deep code search," *arXiv preprint arXiv:2305.05959*, 2023.

[169] N. D. Bui, H. Le, Y. Wang, J. Li, A. D. Gotmare, and S. C. Hoi, "Codetf: One-stop transformer library for state-of-the-art code llm," *arXiv preprint arXiv:2306.00029*, 2023.

[170] W. Sun, C. Fang, Y. You, Y. Miao, Y. Liu, Y. Li, G. Deng, S. Huang, Y. Chen, Q. Zhang *et al.*, "Automatic code summarization via chatgpt: How far are we?" *arXiv preprint arXiv:2305.12865*, 2023.

[171] Q. Chen, J. Lacomis, E. J. Schwartz, C. Le Goues, G. Neubig, and B. Vasilescu, "Augmenting decompiler output with learned variable names and types," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 4327–4343.

[172] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet," *IEEE Transactions on Software Engineering*, 2021.

[173] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 292–303.

[174] A. M. Dakhel, A. Nikanjam, V. Majdinasab, F. Khomh, and M. C. Desmarais, "Effective test generation using pre-trained large language models and mutation testing," *arXiv preprint arXiv:2308.16557*, 2023.

[175] R. Pan, T. A. Ghaleb, and L. Briand, "Atm: Black-box test case minimization based on test code similarity and evolutionary search," *arXiv preprint arXiv:2210.16269*, 2022.

[176] A. Wei, Y. Deng, C. Yang, and L. Zhang, "Free lunch for testing: Fuzzing deep-learning libraries from open source," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 995–1007.

[177] Y. Deng, C. Yang, A. Wei, and L. Zhang, "Fuzzing deep-learning libraries via automated relational api inference," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 44–56.

[178] J. Gu, X. Luo, Y. Zhou, and X. Wang, "Muffin: Testing deep learning libraries via neural architecture fuzzing," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1418–1430.

[179] M. Kim, Y. Kim, H. Jeong, J. Heo, S. Kim, H. Chung, and E. Lee, "An empirical study of deep transfer learning-based program repair for kotlin projects," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1441–1452.

[180] X. Zhang, J. Zhai, S. Ma, and C. Shen, "Autotrainer: An automatic dnn training problem detection and repair system," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 359–371.

[181] L. Wang, X. Tang, Y. He, C. Ren, S. Shi, C. Yan, and Z. Li, "Delving into commit-issue correlation to enhance commit message generation models," *arXiv preprint arXiv:2308.00147*, 2023.

[182] R. Tufano, L. Pascarella, M. Tufano, D. Poshyvanyk, and G. Bavota, "Towards automating code review activities," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 163–174.

[183] D. Vyas, T. Fritz, and D. Shepherd, "Bug reproduction: A collaborative practice within software maintenance activities," in *COOP 2014-Proceedings of the 11th International Conference on the Design of Cooperative Systems, 27-30 May 2014, Nice (France)*. Springer, 2014, pp. 189–207.

[184] C. Niu, C. Li, V. Ng, and B. Luo, "Crosscodebench: Benchmarking cross-task generalization of source code models," *arXiv preprint arXiv:2302.04030*, 2023.

[185] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *arXiv preprint arXiv:2305.01210*, 2023.

[186] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation," *arXiv preprint arXiv:2308.01861*, 2023.

[187] Q. Zhang, T. Zhang, J. Zhai, C. Fang, B. Yu, W. Sun, and Z. Chen, "A critical review of large language model on software engineering: An example from chatgpt and automated program repair," 2023.

[188] A. Mastropaolo, N. Cooper, D. N. Palacio, S. Scalabrino, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Using transfer learning for code-related tasks," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1580–1598, 2022.

[189] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, and L. Zhang, "An extensive study on pre-trained models for program understanding and generation," in *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*, 2022, pp. 39–51.

[190] C. Niu, C. Li, V. Ng, D. Chen, J. Ge, and B. Luo, "An empirical comparison of pre-trained models of source code," *arXiv preprint arXiv:2302.04026*, 2023.

[191] S. Jalil, S. Rafi, T. D. LaToza, K. Moran, and W. Lam, "Chatgpt and software testing education: Promises & perils," in *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2023, pp. 4130–4137.

[192] C. Geng, Z. Yihan, B. Pientka, and X. Si, "Can chatgpt pass an introductory level functional language programming course?" *arXiv preprint arXiv:2305.02230*, 2023.

[193] P. T. Nguyen, J. Di Rocco, C. Di Sipio, R. Rubei, D. Di Ruscio, and M. Di Penta, "Is this snippet written by chatgpt? an empirical study with a codebert-based classifier," *arXiv preprint arXiv:2307.09381*, 2023.

[194] Z. Yang, J. Shi, J. He, and D. Lo, "Natural attack for pre-trained models of code," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1482–1493.

[195] R. Schuster, C. Song, E. Tromer, and V. Shmatikov, "You autocomplete me: Poisoning vulnerabilities in neural code completion," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1559–1575.

[196] Y. Wan, S. Zhang, H. Zhang, Y. Sui, G. Xu, D. Yao, H. Jin, and L. Sun, "You see what i want you to see: poisoning vulnerabilities in neural code search," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1233–1245.

[197] W. Sun, Y. Chen, G. Tao, C. Fang, X. Zhang, Q. Zhang, and B. Luo, "Backdooring neural code search," *arXiv preprint arXiv:2305.17506*, 2023.

[198] J. Li, Z. Li, H. Zhang, G. Li, Z. Jin, X. Hu, and X. Xia, "Poison attack and poison detection on deep source code processing models," *ACM Transactions on Software Engineering and Methodology*, 2023.

[199] Y. Li, S. Liu, K. Chen, X. Xie, T. Zhang, and Y. Liu, "Multi-target backdoor attacks for code pre-trained models," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*. Association for Computational Linguistics, 2023, pp. 7236–7254.

[200] Z. Li, C. Wang, P. Ma, C. Liu, S. Wang, D. Wu, and C. Gao, "On the feasibility of specialized ability stealing for large language code models," *arXiv preprint arXiv:2303.03012*, 2023.

[201] A. Jha and C. K. Reddy, "Codeattack: Code-based adversarial attacks for pre-trained programming language models," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, no. 12, 2023, pp. 14 892–14 900.

[202] E. Shi, Y. Wang, H. Zhang, L. Du, S. Han, D. Zhang, and H. Sun, "Towards efficient fine-tuning of pre-trained code models: An experimental study and beyond," *arXiv preprint arXiv:2304.05216*, 2023.

[203] S. Gao, H. Zhang, C. Gao, and C. Wang, "Keeping pace with ever-increasing data: Towards continual learning of code intelligence models," *arXiv preprint arXiv:2302.03482*, 2023.

[204] M. Weyssow, X. Zhou, K. Kim, D. Lo, and H. Sahraoui, "On the usage of continual learning for out-of-distribution generalization in pre-trained language models of code," *arXiv preprint arXiv:2305.04106*, 2023.

[205] D. Wang, B. Chen, S. Li, W. Luo, S. Peng, W. Dong, and X. Liao, "One adapter for all programming languages? adapter tuning for code search and summarization," *arXiv preprint arXiv:2303.15822*, 2023.

[206] J. Shi, Z. Yang, B. Xu, H. J. Kang, and D. Lo, "Compressing pre-trained models of code into 3 mb," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.