SCIENCE CHINA
Information Sciences

• REVIEW •

April 2026, Vol. 69, Iss. 4, 141102:1–141102:52
https://doi.org/10.1007/s11432-025-4670-0

A survey on large language models for software engineering

Quanjun ZHANG[1], Chunrong FANG[1*], Yang XIE[1], Yaxin ZHANG[1], Shengcheng YU[1], Weisong SUN[2], Yun YANG[3] & Zhenyu CHEN[1*]

[1]*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China*
[2]*College of Computing and Data Science, Nanyang Technological University, Singapore 639798, Singapore*
[3]*Department of Computing Technologies, Swinburne University of Technology, Melbourne 3122, Australia*

Received 23 January 2025/Revised 16 June 2025/Accepted 5 November 2025/Published online 5 March 2026

Abstract Software engineering (SE) is the systematic design, development, maintenance, and management of software applications underpinning the digital infrastructure of our modern world. 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 on 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 62 representative LLMs of 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 926 studies for 112 specific code-related tasks across five crucial phases within the SE workflow. We also discuss several critical aspects during the integration of LLMs into SE, such as empirical evaluation, benchmarking, security and reliability, domain tuning, compressing, and distillation. Finally, we highlight several challenges and potential opportunities in 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 be continuously updated at the living repository https://github.com/iSEngLab/AwesomeLLM4SE.

Keywords software engineering, large language model, AI and software engineering, LLM4SE

Citation Zhang Q J, Fang C R, Xie Y, et al. A survey on large language models for software engineering. Sci China Inf Sci, 2026, 69(4): 141102, https://doi.org/10.1007/s11432-025-4670-0

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].

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 [8–11] (e.g., code summarization and code search) with LLMs (e.g., CodeBERT and CodeT5). 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 [12], 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

* Corresponding author (email: fangchunrong@nju.edu.cn, zychen@nju.edu.cn)

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 it. 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 [13–15] and unit testing [16–18]. These studies encompass different research aspects (e.g., empirical and technical studies [19]), types of LLMs (e.g., open-source or closed-source [14]), mode architectures (e.g., encoder-decoder or encoder-only [20]), model parameters (e.g., CodeT5-60M and InCoder-6B [21]), types of bugs (e.g., semantic bugs and security vulnerabilities [22]), and utilization paradigms (e.g., fine-tuning [23, 24] and few-shot [25]).

In this paper, we summarize existing work and provide a retrospective 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 988 relevant studies empowered with LLMs in terms of publication trends and distribution of venues until August 2024.

• LLM perspective. We summarize 62 representative LLMs of Code for the SE community according to different aspects, such as model architectures, pre-training objectives, downstream tasks, and open science.

• SE perspective. We explore the typical application of leveraging the advances of recent LLMs to automate the SE research, involving 926 relevant studies for 112 code-related tasks across five SE phases, i.e., software requirements and design, software development, software testing, software maintenance and software management.

• Integration perspective. We discuss some crucial aspects when LLMs are integrated into the SE field, such as evaluation, benchmarking, security and reliability, and domain 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. In 2022, Watson et al. [26], Wang et al. [27] and Yang et al. [3] conducted systematic literature reviews on research at the intersection of SE and ML&DL. Such surveys [28] mainly concentrate on the application of ML&DL techniques in SE rather than more powerful and rapidly emerging LLMs. Besides, Niu et al. [29] and Zan et al. [30] presented surveys to summarize 20 and 27 LLMs for natural-language-to-code (NL4Code) tasks. Such surveys are limited to a narrow research scope of NL4Code, thus ignoring the more complex and challenging SE domain. More recently, Chen et al. [31] conducted the first task-oriented survey on deep learning-based SE covering twelve major SE subareas that have been significantly shaped by deep learning techniques. In contrast, our work emphasizes the foundational aspects of recently emerged LLMs within the critical context of SE research, particularly covering 112 specific tasks across five crucial SE phases, i.e., software requirement & design, development, testing, maintenance and management, as well as corresponding integration aspects.

In addition to the aforementioned surveys, we notice there exist several studies that explore the integration of LLMs with SE [4, 32–37]. While these studies are concurrent with our work, they exhibit fundamental differences compared to our work. For example, Wang et al. [4] provided a review of LLMs in software testing, whereas our work targets the whole SE scope rather than a single SE phase. Recent surveys also focus on specific SE tasks, such as program repair [13], unit testing [38], vulnerability detection [39, 40], and code generation [41], whereas our work provides a broader and more comprehensive overview covering the full range of LLM applications in software engineering. Fan et al. [32] and Gao et al. [42] discussed the overall achievements and challenges of LLMs in SE. Hou et al. [33] conducted a systematic literature review on LLM4SE, and He et al. [43] discussed the potential of utilizing LLMs to address complex challenges in SE with multi-agents. Despite their relevance, our work distinguishes itself through a three-fold focus: (1) the foundational aspects of LLMs (i.e., 62 LLMs in Section 3), (2) the broader SE context (i.e., 112 tasks in Section 4), and (3) the integration of LLMs within SE (i.e., Section 5). Unlike the first work, which only provides a bird's-eye view of LLM-based SE advancements, and the second, which delves into complex aspects such as data processing and evaluation metrics, our survey adopts a structured and straightforward review strategy. Moreover, our work provides detailed summaries of representative studies, such as 62 LLMs and 15 pre-training tasks, which are overlooked by previous studies and offer valuable insights for readers, particularly

Table 1 Comparison of related surveys and our work.

Surveys	Year	Scope	Time frame	# Papers included
Watson et al. [26]	2022	DL4SE	2009–2019	128
Wang et al. [27]	2022	ML/DL4SE	2009–2020	1428
Yang et al. [3]	2022	DL4SE	2015–2020	142
Niu et al. [29]	2022	LLMs for Code	–	20
Zan et al. [30]	2023	LLMs for Code	2020–2023	27
Wang et al. [4]	2023	LLM4Testing	2019–2023	102
Fan et al. [32]	2023	LLM4SE	–	–
Hou et al. [33]	2023	LLM4SE	2017–2023	395
Zheng et al. [34]	2024	LLM4SE	2022–2024	123
Chen et al. [31]	2025	DL4SE	2020–2023	601
Our work	2025	LLM4SE	2017–2024	988

for novice researchers entering this field. To further contribute to the community, we have released the first public repository to track the latest developments in this area via crowd-sourcing, fostering continued engagement and collaboration. Finally, our survey incorporates studies published up until August 2024, ensuring a comprehensive and up-to-date perspective. Table 1 presents a detailed comparison of existing surveys and our work.

Paper organization. The remainder of this paper is organized as follows. Section 2 provides a detailed exposition of three 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 the crucial aspects during the integration of LLMs into SE. Section 6 highlights the challenges and promising opportunities for future research. Section 7 draws the conclusion.

Availability. All artifacts of this study are available in the following public repository. The living repository continuously updates the latest research on LLMs for code, LLM4SE, and related studies[1)].

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 (LLM perspective):** What LLMs are designed to support SE 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?

RQ1 aims to provide a taxonomy and analysis of LLMs that have been introduced and adapted for SE applications. To this end, this RQ summarizes LLMs from four perspectives: LLM categories in Section 3.1, pre-training tasks in Section 3.2, downstream tasks in Section 3.3, and open-science in Section 3.4. Particularly, RQ1.1 focuses on identifying LLMs that are explicitly designed or evaluated for SE, along with their release timelines and architectural types (e.g., encoder-only, decoder-only, encoder-decoder). RQ1.2 analyzes the training objectives used during the pre-training phase to understand how LLMs learn from source code. RQ1.3 categorizes the downstream tasks used to evaluate LLM in the original papers, particularly those involving code understanding or generation. RQ1.4 examines the degree to which LLMs and associated artifacts (e.g., models, datasets, benchmarks) are publicly available to support open science.

- **RQ2 (SE perspective):** What SE tasks are facilitated by LLMs?

RQ2 aims to explore the scope of SE tasks to which LLMs have been applied. To this end, this RQ investigates 112 distinct SE tasks that LLMs have supported from five major areas: software requirements & design in Section 4.1, software development in Section 4.2, software testing in Section 4.3, software maintenance in Section 4.4 and software management in Section 4.5.

- **RQ3 (integration perspective):** What are the key factors during the integration of LLMs into SE?

1) https://github.com/iSEngLab/AwesomeLLM4SE.

Table 2 Checklist of quality assessment criteria for LLM-based SE studies.

ID	Quality assessment criteria
QAC1	Is the study primarily focused on SE tasks?
QAC2	Is the study centered on using LLMs to address SE challenges (LLM4SE), rather than leveraging SE techniques to improve LLMs (SE4LLM)?
QAC3	Is the study an original research contribution rather than a secondary study, such as a systematic literature review (SLR), review, or survey?
QAC4	Is the study published in a reputable venue?
QAC5	Does the study provide a clear research motivation?
QAC6	Does the study provide a clear description of the techniques used?
QAC7	Are the experimental setups, including experimental environments and dataset information, described in detail?
QAC8	Are the experimental findings clearly validated and aligned with the research goals?
QAC9	Are the key contributions and limitations of the study discussed?
QAC10	Does the study make a meaningful contribution to the academic or industrial community?

RQ3 aims to analyze challenges and achievements during the integration of LLMs into SE. To this end, this RQ examines four key aspects: i.e., evaluation and benchmarking in Section 5.1, security and reliability in Section 5.2, domain tuning in Section 5.3, and compressing and distillation in Section 5.4.

2.2 Search strategy

Following existing DL for SE surveys [3, 26, 27], 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 number of relevant papers from SE, AI, and NLP communities, following Zhang et al. [44], we attempt to identify some preliminary search keywords from three sources: (1) existing LLM surveys [30] to derive LLM-related keywords; (2) existing SE surveys [26, 27] 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 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 the 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 August 2024, i.e., Google Scholar repository, ACM Digital Library, and IEEE Explorer Digital Library. Finally, we retrieved a total of 32560 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 [45] was proposed in 2017, which is the foundation of LLMs. Second, we automatically filter out any paper less than 7 pages and duplicate papers. Third, we conduct a full-text inspection of the remaining papers manually to determine whether they are relevant to the LLM-based SE field. Following Zhang et al. [13], we design ten questions to assess the relevance and quality of these papers, as presented in Table 2. For example, regarding QAC1 and QAC2, we focus on LLM4SE papers, i.e., studies applying LLMs to SE tasks, while explicitly excluding SE4LLM papers that center on LLM optimization without involving any SE tasks. Regarding QAC4, given the nascent nature of this research area and the fact that many recent papers have not yet undergone peer review, we include papers from arXiv and selectively incorporate high-quality papers through our quality assessment process to ensure that our survey remains comprehensive and up-to-date. The manual inspection is conducted independently by two authors, and any paper with different decisions will be

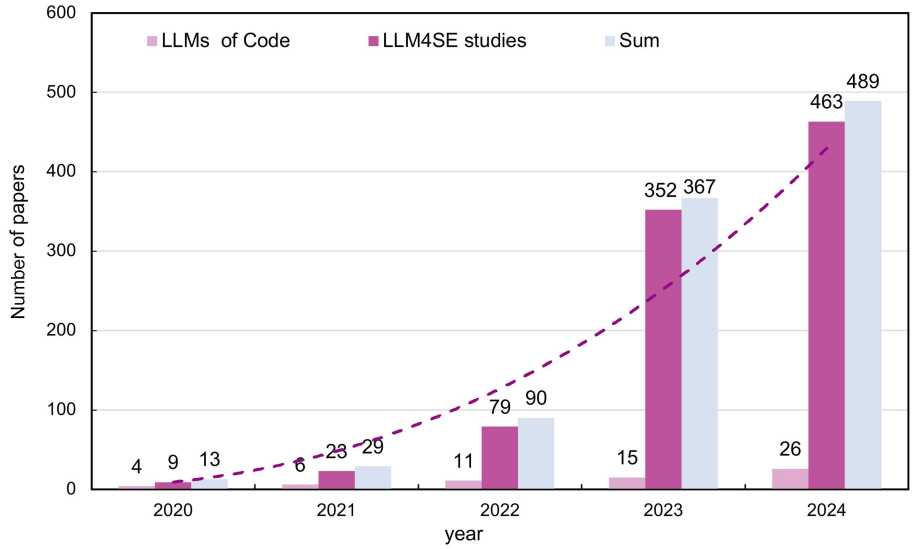

Figure 1 (Color online) Number of collected papers over the years.

handed over to a third author to make the final decision. As a result, we collect 59 papers related to the code LLM research and 891 papers related to the LLM-based SE research.

To mitigate potential omissions in our automated search and to ensure a thorough collection of papers, we further employed a snowballing search strategy [26]. 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 looked at every reference within the collected papers and determined if any of those references are relevant to our study. Through this rigorous manual analysis, we successfully identified three additional papers related to LLMs and 35 papers related to LLM-based SE, thereby enriching our survey with a diverse range of insights.

2.4 Trend observation

We finally obtain 988 relevant research papers after automated searching and manual inspection. Figure 1 shows the number of collected papers from 2020 to 2024, with the dashed line representing the overall growth trend in publications. It can be observed that studies on proposing LLMs and their applications in SE have rapidly increased since 2020, indicating the growing recognition among researchers of LLMs as a viable and promising approach to automating SE tasks. One possible reason is that DL technologies have already shown promising performance in various SE tasks over the past several years [26]. 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 3 further shows the number of collected papers across different venues. The first two columns list the venue and its acronym, the third column indicates whether it is a conference or journal (i.e., abbreviated as C or J), and the final column shows the number of publications. We only present the top-30 venues with the highest number of publications due to page limitations. First, we find that these papers span multiple research fields, including SE, NLP, AI, and Security, which indicates the wide range of attention this direction has received. Second, unlike previous studies [46,47], it can be found that a significant number of papers (381/988) 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 the field of SE. Third, the top five venues are top-tier conferences (ICSE, FSE, ACL, ASE and ISSTA), and 23 venues among the top-30 ones are conferences, indicating a current inclination towards conferences in this field due to the timeliness of conference proceedings.

Table 3 Venue distribution of collected papers.

Acronym	Venues	Type	# Publications
arXiv	–		381
ICSE	International Conference on Software Engineering	C	88
FSE	International Symposium on Foundations of Software Engineering	C	52
ACL	Meeting of the Association for Computational Linguistics	C	31
ASE	International Conference on Automated Software Engineering	C	30
ISSTA	International Symposium on Software Testing and Analysis	C	27
TSE	Transactions on Software Engineering	J	25
TOSEM	Transactions on Software Engineering Methodology	C	20
ICLR	International Conference on Learning Representations	C	20
SANER	IEEE International Conference on Software Analysis, Evolution, and Reengineering	C	17
EMNLP	Empirical Methods in Natural Language Processing	C	16
ICSME	IEEE International Conference on Software Maintenance and Evolution	C	15
MSR	Mining Software Repositories	C	14
ICML	International Conference on Machine Learning	C	14
JSS	Journal of Systems and Software	J	12
AAAI	Association for the Advance of Artificial Intelligence	C	10
ICPC	International Collegiate Programming Contest	C	10
NeurIPS	Neural Information Processing Systems	C	10
IST	Information and Software Technology	J	8
Internetware	Asia-Pacific Symposium on Internetware	C	8
ISSRE	International Symposium on Software Reliability Engineering	C	8
ICST	IEEE International Conference on Software Testing	C	7
APSEC	Asia-Pacific Software Engineering Conference	C	7
USENIX Security	USENIX Security Symposium	C	7
NAACL	North American Association of Computational Linguistics	C	6
AUSE	Automated Software Engineering	J	6
ESEM	International Symposium on Empirical Software Engineering and Measurement	C	6
SCP	Science of Computer Programming	J	5
TMLR	Transactions on Machine Learning Research	J	5
COMPSAC	International Computer Software and Applications Conference	C	5

3 RQ1: LLM perspective

In this section, we provide a four-fold analysis to summarize existing LLMs in the era of SE, including 62 representative LLMs of Code in Section 3.1, 15 pre-training tasks of LLMs in Section 3.2, 16 fine-tuning tasks of LLMs in Section 3.3, and the crucial open science issue related to LLMs in Section 3.4. To provide a structured overview, we also present a detailed taxonomy in Figure 2, which outlines the relationships among the sub-RQs and their corresponding categorizations [48–108].

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 4 presents the summary and comparison of these representative LLMs[2)]. The columns summarize the year of release, model name, the publisher or the conference where the model is introduced, the architecture types, and the initialization method or the base model used for pre-training.

From Table 4, it can be found that these LLMs are usually derived from foundational architectures in the NLP community and trained with some code-aware objectives. 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 are gaining increasing popularity. In

2) We note that some general-purpose LLMs (not limited to source code) have been applied in the field of SE, such as PaLM, Qwen, LLaMA, and Gemini. However, such LLMs have already been extensively reviewed in NLP and AI communities, so they fall outside the scope of our work. For more details, please refer to the relevant studies [109, 110].

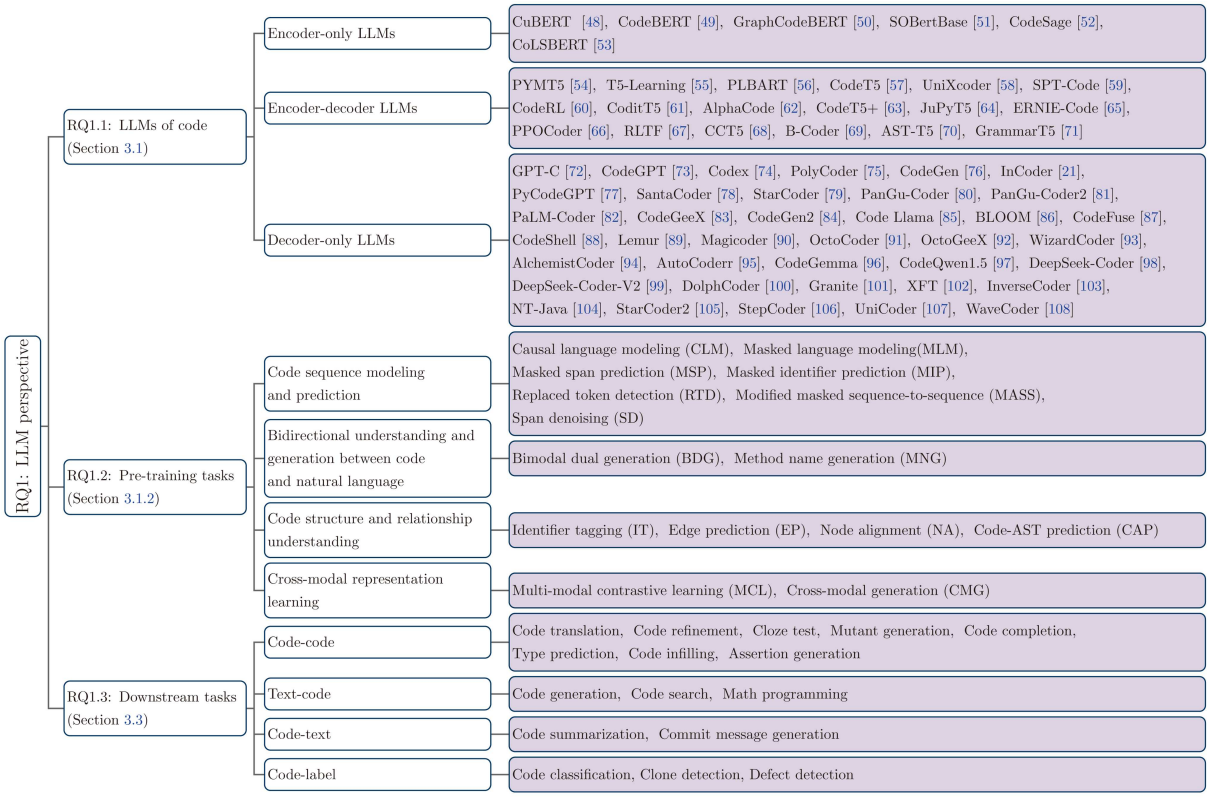

Figure 2 (Color online) Taxonomy of RQ1.

the following, we summarize these LLMs according to their model architectures. More detailed information about these specific LLMs is included in Appendix A.

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 [48–53]. 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 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 [5] 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 studies. For example, CuBERT [48] is the first adaptation of BERT from NLP to the source code domain by replicating the training procedure of BERT on a Python code corpus. CodeBERT [49] is a bimodal variant of BERT that takes into account both natural language and programming language. GraphCodeBERT [50] is a structure-aware extension of CodeBERT that incorporates data flow graphs to capture the structural and semantic relationships within the source code.

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 [54–71]. 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

Table 4 A summary and comparison of LLMs of Code.

Year	Model	Publisher	Architecture	Size	Tokenizer	Init	Organization	Public
2020	GPT-C	FSE	Decoder-only	366M	BPE	GPT-2	Microsoft	No
2020	CodeBERT	EMNLP	Encoder-only	125M	WordPiece	Scratch	HIT, Microsoft	Yes
2020	CuBERT	ICML	Encoder-only	350M	–	BERT	Google, IISC	Yes
2020	PyMT5	EMNLP	Encoder-Decoder	374M	BBPE	GPT-2	Microsoft	Yes
2021	CodeGPT	NeurIPS	Decoder-only	124M	BBPE	GPT-2	PKU, Microsoft	Yes
2021	CodeT5	EMNLP	Encoder-Decoder	60M, 220M	BBPE	Scratch	Salesforce	Yes
2021	Codex	arXiv	Decoder-only	12M, 25M, 42M, 85M, 300M, 679M, 2.5B, 12B	BBPE	GPT-3	OpenAI	No
2021	GraphCodeBERT	ICLR	Encoder-only	125M	WordPiece	Scratch	Microsoft, SYSU	Yes
2021	PLBART	NAACL	Encoder-Decoder	140M, 406M	SentencePiece	Scratch	UC	Yes
2021	T5-Learning	ICSE	Encoder-Decoder	60M	SentencePiece	T5	USI	Yes
2022	AlphaCode	Science	Encoder-Decoder	300M, 3B, 9B, 41B	SentencePiece	Scratch	DeepMind	No
2022	BLOOM	arXiv	Decoder-only	176B	BPE	Scratch	BigScience Workshop	Yes
2022	CodeRL	NeurIPS	Encoder-Decoder	770M	BBPE	CodeT5	Salesforce	Yes
2022	CoditT5	ASE	Encoder-Decoder	220M	BBPE	CodeT5	UTexas	Yes
2022	JuPyT5	arXiv	Encoder-Decoder	300M	BBPE	PyMT5	Microsoft	Yes
2022	PaLM-Coder	JMLR	Decoder-only	8B, 62B, 540B	SentencePiece	Scratch	Google	No
2022	PanGu-Coder	arXiv	Decoder-only	317M, 2.6B	SentencePiece	Scratch	Huawei	No
2022	PolyCoder	ICLR	Decoder-only	41B, 160M, 400M	BPE	GPT-2	CMU	Yes
2022	PyCodeGPT	IJCAI	Decoder-only	110M	BPE	GPT-Neo	Microsoft, CAS	Yes
2022	SPT-Code	ICSE	Encoder-Decoder	262M	BPE	Scratch	Nanjing University	Yes
2022	UnixCoder	ACL	Encoder-Decoder	125M	WordPiece	Scratch	Microsoft, SYSU	Yes
2023	CCT5	FSE	Encoder-Decoder	220M	BBPE	CodeT5	NUDT	Yes
2023	Code Llama	arXiv	Decoder-only	13B, 7B, 34B	SentencePiece	Llama2	Meta	Yes
2023	CodeFuse	ICSE	Decoder-only	13B	BPE	GPT-NeoX	Ant Group	Yes
2023	CodeGen	ICLR	Decoder-only	350M, 2.7B, 6.1B, 16.1B	BPE	Scratch	Salesforce	Yes
2023	CodeGen2	ICLR	Decoder-only	16B, 3.7B, 7B	BPE	Scratch	Salesforce	Yes
2023	CodeShell	arXiv	Decoder-only	7B	BPE	GPT-2	PKU	Yes
2023	CodeT5+	EMNLP	Encoder-Decoder	770M, 2B, 6B, 16B	BBPE	CodeT5	Salesforce	Yes
2023	ERNIE-Code	ACL	Encoder-Decoder	560M	SentencePiece	Scratch	Baidu	Yes
2023	InCoder	ICLR	Decoder-only	6.7B, 1.3B	BBPE	Scratch	CMU, Meta, UW	Yes
2023	PanGu-Coder2	arXiv	Decoder-only	15B	SentencePiece	PanGu-Coder	Huawei	No
2023	PPOCoder	TMLR	Encoder-Decoder	770M, 220M	BBPE	CodeT5	Virginia Tech	Yes
2023	RLTF	TMLR	Encoder-Decoder	770M	–	CodeT5	Tencent	Yes
2023	SantaCoder	ICLR	Decoder-only	1.1B	BPE	Scratch	Hugging Face	Yes
2023	SOBertBase	arXiv	Encoder-only	109M, 762M	BPE	Scratch	CMU	No
2023	StarCoder	TMLR	Decoder-only	15.5B	BBPE	GPT-2	Hugging Face	Yes
2024	AlchemistCoder	arXiv	Decoder-only	7B, 6.7B	–	CodeLlama, Llama2, DeepSeekCoder	Tongji University	Yes
2024	AST-T5	ICML	Encoder-Decoder	277M	BPE	T5	UC, Meta	Yes
2024	AutoCoder	arXiv	Decoder-only	6.7B, 33B	BPE	DeepSeekCoder	UCONN	Yes
2024	B-Coder	ICLR	Encoder-Decoder	770M	BBPE	T5	UIC	No
2024	CodeGeeX	KDD	Decoder-only	13B	BBPE	GPT-2	Tsinghua, Zhipu.AI	Yes
2024	CodeGemma	arXiv	Decoder-only	2B, 7B	BPE	Gemma	Google	Yes
2024	CodeQwen1.5	arXiv	Decoder-only	7B	BPE	Qwen1.5	Alibaba Group	Yes
2024	CodeSage	ICLR	Encoder-only	130M, 356M, 1.3B	SentencePiece	Scratch	AWS AI Labs	Yes
2024	CoLSBERT	arXiv	Encoder-only	354M, 757M, 124M, 1.5B	BPE	Scratch	IDEA	Yes
2024	DeepSeek-Coder	arXiv	Decoder-only	1.3B, 6.7B, 33B	BPE	Scratch	PKU	Yes
2024	DeepSeek-Coder-V2	arXiv	Decoder-only	16B, 236B	BPE	DeepSeek-V2	PKU	Yes
2024	DolphCoder	ACL	Decoder-only	7B, 13B	–	CodeLlama	BUPT	Yes
2024	GrammarT5	ICSE	Encoder-Decoder	60M, 220M	BBPE	CodeT5	PKU	Yes
2024	Granite	arXiv	Decoder-only	3B, 8B, 20B, 34B	BPE	Scratch	IBM Research	Yes
2024	InverseCoder	arXiv	Decoder-only	6.7B, 7B	–	CodeLlama, DeepSeekCoder	UCAS, CAS	Yes
2024	Lemur	ICLR	Decoder-only	70B	BBPE	Llama2	HKU	Yes
2024	Magicoder	ICML	Decoder-only	7B	BPE	CodeLlama, DeepSeekCoder	UIUC	Yes
2024	NT-Java	arXiv	Decoder-only	1.1B	BBPE	StarCoder	Infosys Limited	Yes
2024	OctoCoder	ICLR	Decoder-only	16B	BBPE	StarCoder	Hugging Face	Yes
2024	OctoGeeX	ICLR	Decoder-only	6B	BBPE	CodeGeeX2	Hugging Face	Yes
2024	StarCoder2	arXiv	Decoder-only	3B, 7B, 15B	BBPE	StarCoder	Hugging Face	Yes
2024	StepCoder	ACL	Decoder-only	6.7B	–	DeepSeekCoder	Fudan	Yes
2024	UniCoder	ACL	Decoder-only	6.7B, 7B	BPE	DeepSeekCoder	BUAA	Yes
2024	WaveCoder	ACL	Decoder-only	15B, 7B, 6.7B, 13B	BBPE	StarCoder, CodeLlama, DeepSeekCoder	Microsoft	Yes
2024	WizardCoder	ICLR	Decoder-only	15B	BBPE	StarCoder	Microsoft, HKBU	Yes
2024	XFT	ACL	Decoder-only	1.3B	BPE	DeepSeekCoder	UIUC	Yes

output sequences can be of different lengths and structures, such as code summarization and program repair.

Among existing encoder-decoder LLMs, T5 (Text-to-Text Transfer Transformer) is a significant development in the NLP field and serves as a catalyst for follow-up code-related studies. For example, similar to CuBERT [48] in the encoder-only LLM domain, PYMT5 [54] is the first attempt to apply T5 to source code by replicating the pre-training process of T5 on a code corpus. In parallel with PYMT5 [54], Mastropaolo et al. [55] empirically investigated how T5 performs when pre-trained with CodeSearchNet and fine-tuned to support four code-related tasks. PLABRT [56] is pre-trained with the denoising objective and built on the BART architecture. CodeT5 [49] represents a well-known adaptation of T5 from NLP to the source code domain by leveraging the code semantics from the developer-assigned identifiers. CoditT5 [61] is a variant of CodeT5 particularly trained to tackle code editing tasks, such as code review. Researchers also release some encoder-decoder LLM for specific scenarios, such as AlphaCode [62] is introduced by DeepMind to generate solutions for competitive programming problems and

Table 5 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)
	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)

JuPyT5 [64] for Jupyter Notebook.

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 [72–108]. 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.

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. Early efforts focus on adapting the GPT series models for the code domain, leading to LLMs such as GPT-C [72], CodeGPT [73], PolyCoder [75], and PyCodeGPT [77]. Subsequent advancements have seen the introduction of various specialized LLMs tailored for code generation and understanding tasks with advanced training techniques and large-scale datasets. Examples include CodeGen [76], InCoder [21], SantaCoder [78], StarCoder [79], CodeGeeX [83], and CodeGen2 [84]. Recently, with the development of general-purpose LLMs, several variants have been released specifically in handling code-related tasks, such as CodeLlama [111], PaLM-Coder [82], PanGu-Coder [80], PanGu-Coder2 [81].

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 5 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[3)] 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 $x = (x_1, \ldots, x_n)$ with 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. As mentioned in CodeT5 [57], MSP randomly masks spans with arbitrary lengths and then predicts these masked spans combined with some sentinel tokens at the decoder. The input of LLMs is the original sequence, the mask sequence is processed by the noise function, and the output is the denoised sequence.

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.

Replaced token detection (RTD). Originally proposed by Clark et al. [112], 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 performs 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 $p_{Gw}(w_i|w_{\text{masked}})$ for positions i in m_w, and sampling alternative tokens $\hat{c}_i$ from $p_{Gc}(c_i|c_{\text{masked}})$ 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 $C^{u:v}_{\text{origin}}$ 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, 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

3) 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.

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 $[\text{Input} = C, [\text{SEP}], A, [\text{SEP}], N]$, where C is the code snippet, A is the corresponding AST sequence, and N is a natural language.

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 [57], 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.

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 [50]. The primary motivation behind this task is to encourage the 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 [50]. 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.

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 [59]. 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 [58]. 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 [58], 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.

Cross-modal generation (CMG). CMG attempts to generate comments for code segments to aid LLMs in understanding code semantics. The comment generation process is conditioned on the code, embedding semantic information into the hidden states. Besides, to expose LLMs to diverse contexts, a strategy is applied where the source and target inputs are randomly swapped with a 50% probability.

As shown in Table 5, while these pre-training objectives vary in form, they also differ in their suitability for downstream task types. For example, objectives (e.g., CLM and MASS) are typically more aligned with generation tasks (e.g., code completion, summarization), as they encourage models to learn sequential dependencies and output fluency. In addition, objectives (e.g., MLM, RTD, and MIP) are better suited for classification or understanding tasks (e.g., defect prediction, clone detection), since they improve bidirectional representations and semantic reasoning. Empirical studies [57, 63] have shown that models trained with code-aware objectives like MIP and IT tend to outperform generic language models on software engineering tasks, due to their stronger alignment with

code structure and semantics. However, comprehensive benchmark comparisons across pre-training objectives in SE contexts remain limited, and more systematic empirical evaluations are needed in future work.

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 studies 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 also 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 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 knowledge acquired during pre-training to downstream tasks, requiring LLMs to demonstrate capabilities in code understanding, reasoning, and generation. Following prior studies [29, 73], downstream tasks 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). It is worth noting that this section focuses on downstream tasks as reported in the original LLM papers, which typically do not cover more complex SE tasks (e.g., test generation) and data types (e.g., GUI inputs). We summarize 15 representative downstream tasks that are evaluated by existing LLMs in their original papers according to a well-maintained repository[4)], detailed as follows.

Code-code. Code-code tasks involve the process of transforming one code snippet into another. For example, code translation attempts to convert code from one programming language into another while preserving its functionality. This task has been adopted as a downstream task in LLMs like CodeT5 [57], CodeBERT [49], and CodeT5 [57]. Code refinement (also known as program repair) attempts to refine existing code that might contain errors, and has been explored in LLMs like CodeT5 [57], GraphCodeBERT [50] and SPT-Code [59]. Cloze test aims to predict a missing token in a code snippet and has been adopted in LLMs like CodeGPT [83]. Mutant generation attempts to generate mutants by introducing small artificial faults, such as replacing the `+` operator with `-`, and has been adopted in LLMs like T5-learning [55]. Assert generation generates assert statements to verify the correctness of programs and validate certain assumptions, and has been adopted in LLMs, like T5-learning [55].

Text-code. Text-code tasks involve the process of transforming human language descriptions into code snippets. For example, Code generation attempts to directly produce code snippets based on natural language descriptions, such as docstrings. It has been widely adopted as a downstream task in LLMs, including PyMT5 [54], CodeT5 [57], Codex [74]. Code search refers to the retrieval of relevant code samples from a codebase that matches a given natural language query, and haven been adopted in LLMs like CodeT5+ [63], CodeGPT [73], UnixCoder [58] and SPT-Code [59].

Code-text. Code-text tasks involve the process of transforming code snippets into human language descriptions. For example, 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. It has been explored in various LLMs, including CodeT5 [57], GraphCodeBERT [50], PLBART [56], CodeGPT [73], UnixCoder [58], SPT-Code [59] and ERNIE-Code [65].

Code-label. Code-label tasks involve the process of performing classifications of code snippets. For example, clone detection identifies whether two code snippets are functionally or semantically similar based on similarity analysis and has been adopted in LLMs like CodeBERT [49], CodeT5 [57] and CodeT5+ [63]. Defect detection predicts whether a piece of source code contains bugs that could potentially make software systems vulnerable to attacks and has been adopted in LLMs like CodeT5 [57], PLBART [56], and CodeGPT [73].

Figure 3 illustrates a Sankey diagram mapping encoder-only, decoder-only, and encoder-decoder models to representative downstream tasks. This mapping reveals distinct architectural preferences. For example, decoder-only models (e.g., CodeGen) dominate in autoregressive tasks like code completion and generation. Encoder-only models (e.g., CodeBERT) are frequently used for classification-oriented tasks such as defect detection and clone detection. Encoder-decoder models (e.g., CodeT5, PLBART) are favored in sequence-to-sequence tasks like code translation and summarization. This analysis provides practical insights into selecting model architectures based on the task type and performance requirements. We recommend that a systematic and empirical evaluation of architecture-task alignment offers practical guidance for architecture selection based on task characteristics and remains a valuable

4) https://microsoft.github.io/CodeXGLUE/.

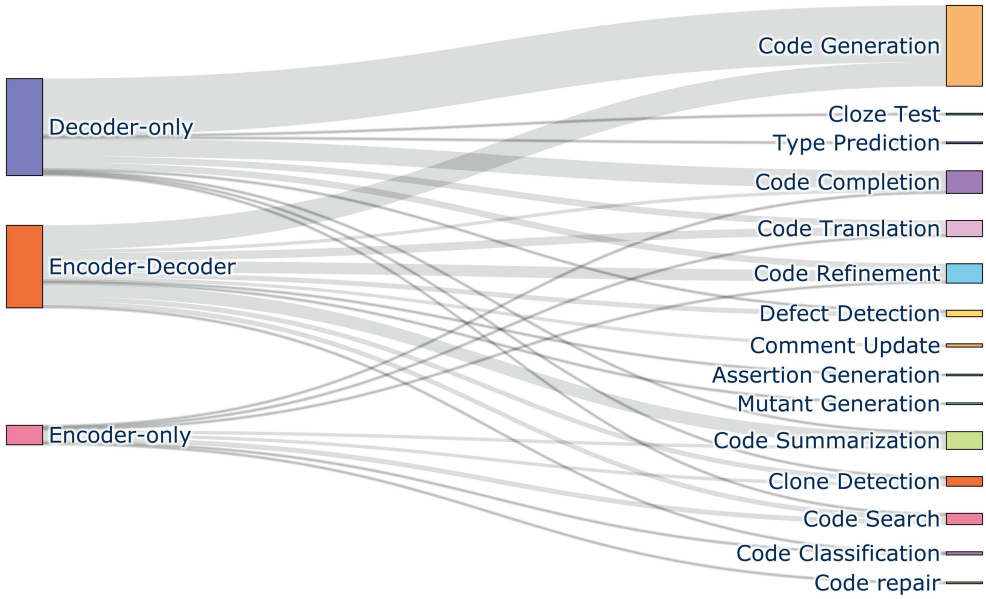

Figure 3 (Color online) Mapping between LLM architecture and downstream tasks.

direction for future work. This analysis provides practical insights into choosing model architectures based on task requirements. We encourage future work to conduct systematic and empirical studies to better understand the relationship between architecture design and downstream SE task performance.

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-text, text-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, as this section focuses on summarizing downstream tasks as reported in the original LLM papers, which typically do not cover more complex SE tasks such as GUI inputs. 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, as 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 [113]. 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 62 investigated LLMs, 54 of them provide the corresponding open-source repositories, which are summarized in Table 6. 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., train-

Table 6 The details of code LLMs availability.

Model	Hosting site	CA	DA	MA	Model site	Public URL
CodeBERT	GitHub	Yes	Yes	Yes	Hugging Face	https://github.com/microsoft/CodeBERT
CuBERT	GitHub	Yes	Yes	Yes	Google Cloud	https://github.com/google-research/google-research
PyMT5	GitHub	Yes	Yes	Yes	PyPi	https://github.com/devcartel/pymt5
CodeGPT	GitHub	No	No	Yes	Hugging Face	https://github.com/microsoft/CodeXGLUE
CodeT5	GitHub	Yes	Yes	Yes	Hugging Face	https://github.com/salesforce/CodeT5
GraphCodeBERT	GitHub	Yes	Yes	Yes	Hugging Face	https://github.com/microsoft/CodeBERT
PLBART	GitHub	Yes	Yes	Yes	Google Drive	https://github.com/wasiahmad/PLBART
T5-Learning	GitHub	Yes	Yes	Yes	Google Drive	https://github.com/antonio-mastropaolo/T5-learning-ICSE_2021
CodeRL	GitHub	Yes	Yes	Yes	Google Cloud	https://github.com/salesforce/CodeRL
CoditT5	GitHub	Yes	Yes	Yes	Hugging Face	https://github.com/engineeringsoftware/coditt5
JuPyT5	GitHub	No	No	No	–	https://github.com/microsoft/DataScienceProblems
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
SPT-Code	GitHub	Yes	Yes	Yes	OneDrive	https://github.com/NougatCA/SPT-Code
UnixCoder	GitHub	No	Yes	Yes	Hugging Face	https://github.com/microsoft/CodeBERT/tree/master/UniXcoder
Code Llama	GitHub	Yes	No	Yes	Hugging Face	https://github.com/facebookresearch/codellama
CodeFuse	GitHub	Yes	No	Yes	Hugging Face	https://github.com/codefuse-ai
CodeGen	GitHub	No	Yes	Yes	Hugging Face	https://github.com/salesforce/CodeGen
CodeGen2	GitHub	No	Yes	Yes	Hugging Face	https://github.com/salesforce/CodeGen2
CodeT5+	GitHub	No	Yes	Yes	Hugging Face	https://github.com/salesforce/CodeT5/tree/main/CodeT5%2B
ERNIE-Code	GitHub	Yes	No	No	–	https://github.com/PaddlePaddle/PaddleNLP/
InCoder	GitHub	No	Yes	Yes	Hugging Face	https://sites.google.com/view/incoder-code-models
PPOCoder	GitHub	Yes	Yes	Yes	–	https://github.com/reddy-lab-code-research/PPOCoder
RLTF	GitHub	Yes	Yes		Hugging Face	https://github.com/Zyq-scut/RLTF
SantaCoder	Hugging Face	No	Yes	Yes	Hugging Face	https://huggingface.co/bigcode/santacoder
StarCoder	GitHub	No	No	No	–	https://github.com/bigcode-project/starcoder
AlchemistCoder	GitHub	Yes	Yes	Yes	Hugging Face	https://github.com/InternLM/AlchemistCoder
AST-T5	GitHub	Yes	Yes	Yes	Hugging Face	https://github.com/gonglinyuan/ast_t5
AutoCoder	GitHub	No	No	Yes	Hugging Face	https://github.com/bin123apple/AutoCoder
CodeGeeX	GitHub	No	No	Yes	–	https://github.com/THUDM/CodeGeeX
CodeGemma	Hugging Face	No	No	No	Hugging Face	https://huggingface.co/blog/codegemma
CodeQwen1.5	GitHub	No	No	Yes	Hugging Face	https://github.com/QwenLM/CodeQwen1.5
CodeSage	GitHub	Yes	Yes	Yes	Hugging Face	https://github.com/amazon-science/CodeSage
CoLSBERT	GitHub	Yes	Yes	Yes	–	https://github.com/stanford-futuredata/ColBERT
DeepSeek-Coder	GitHub	Yes	Yes	Yes	Hugging Face	https://github.com/deepseek-ai/DeepSeek-Coder
DeepSeek-Coder-V2	GitHub	No	Yes	Yes	Hugging Face	https://github.com/deepseek-ai/DeepSeek-Coder-V2
DolphCoder	GitHub	No	No	No	–	https://github.com/pris-nlp/DolphCoder
CCT5	GitHub	Yes	No	Yes	Zenodo	https://github.com/Ringbo/CCT5
BLOOM	Hugging Face	Yes	Yes	Yes	Hugging Face	https://huggingface.co/bigscience/bloom
CoLSBERT	GitHub	Yes	Yes	Yes	–	https://github.com/stanford-futuredata/ColBERT
GrammarT5	GitHub	Yes	Yes	Yes	–	https://github.com/pkuzqh/GrammarT5
Granite	GitHub	No	Yes	Yes	Hugging Face	https://github.com/ibm-granite/granite-code-models
InverseCoder	GitHub	Yes	Yes	Yes	Hugging Face	https://github.com/wyt2000/InverseCoder
Lemur	GitHub	Yes	Yes	Yes	Hugging Face	https://github.com/OpenLemur/Lemur
Magicoder	GitHub	Yes	Yes	Yes	Hugging Face	https://github.com/ise-uiuc/magicoder
NT-Java	Hugging Face	No	Yes	Yes	Hugging Face	https://huggingface.co/infosys/NT-Java-1.1B
OctoCoder	Hugging Face	Yes	Yes	Yes	Hugging Face	https://huggingface.co/bigcode/octocoder
OctoGeeX	Hugging Face	Yes	Yes	Yes	Hugging Face	https://huggingface.co/bigcode/octocoder
StarCoder2	GitHub	Yes	Yes	Yes	Hugging Face	https://github.com/bigcode-project/starcoder2
StepCoder	GitHub	No	Yes	No	–	https://github.com/Ablustrund/APPS_Plus
UniCoder	GitHub	Yes	Yes	Yes	Google Drive	https://github.com/microsoft/Unicoder
WaveCoder	GitHub	Yes	Yes	Yes	Hugging Face	https://github.com/microsoft/WaveCoder
WizardCoder	GitHub	Yes	Yes	Yes	Hugging Face	https://github.com/nlpxucan/WizardLM
XFT	GitHub	Yes	Yes	Yes	–	https://github.com/ise-uiuc/xft

ing 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 54 of 62 LLMs have made their artifacts available to the public. Almost all studies upload their artifacts to 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 [63], perhaps due to commercial reasons.

Answer to RQ1.4. Overall, compared with traditional DL studies, the demand for high-quality and accessible artifacts in LLM research is even more vital to ensure reproducibility and support future advancements. Although numerous LLMs have been introduced for code-related tasks, there remains a significant gap in their adherence to

open science principles. Our findings highlight that platforms like GitHub and Hugging Face are commonly utilized for sharing models and data; however, there is an urgent need for the research community to standardize practices for sharing reproducible artifacts. Therefore, we encourage researchers to prioritize the release of high-quality open-source code and detailed instructions for convenient reproduction.

4 RQ2: SE perspective

In this section, we summarize existing SE studies empowered with LLMs, which can be categorized into five 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, software maintenance in Section 4.4, and software management in Section 4.5. Each SE phase contains several distinct code-related tasks, such as fault localization and program repair in the software maintenance phase. Table 7 presents the taxonomy of this section, categorizing 926 LLM-based SE studies into 112 distinct SE tasks across five SE phases. It should be noted that a single study may address multiple SE tasks. More detailed information about all specific studies is included in Appendix B. Compared with Section 3.3 focusing on summarizing downstream tasks as reported in the original LLM papers, this section systematically covers more complex SE tasks such as test generation and GUI inputs. This evolving trend clearly illustrates the evolving trend of LLM applications in the era of SE, moving from simpler code-related tasks in the early stages to increasingly complex SE tasks in more recent work. It is important to note that some tasks, such as program repair and fuzzing, have a disproportionately large number of studies. To adequately reflect their diversity and research landscape, we provide a more detailed and structured summary for these tasks, following best practices from prior surveys.

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 *Requirement ambiguity detection*

Ambiguities refer to terms or phrases in requirements that can be interpreted in multiple ways, thus leading to misunderstandings and inconsistencies during the development process. Ambiguity detection attempts to identify such unclear, vague, or imprecise statements early in the requirements engineering phase. Moharil et al. [114, 115] introduced TABASCO to detect ambiguities by leveraging BERT to capture the different meanings a word can have depending on its context within a requirement. Further, Ezzini et al. [116] explored multiple approaches (such as SpanBERT) to handle anaphoric ambiguity by ambiguity detection and anaphora interpretation. Sridhara et al. [117] conducted a preliminary empirical study to explore the potential of ChatGPT in anaphora ambiguity resolution.

4.1.2 *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. Kolthoff et al. [118] fine-tuned BERT to retrieve reusable GUIs from a large-scale GUI repository, which can be adapted to facilitate GUI prototyping. Besides, Brie et al. [119] explored the application of LLMs in GUI layout and introduced instigator, which utilizes LLMs to search and suggest GUI layouts based on textual instructions. Instigator aims to enhance creativity and efficiency in the GUI design process by providing designers with relevant and diverse layout options. This work highlights the potential of LLMs in supporting GUI design tasks, particularly by automating parts of the creative process.

Table 7 Distribution of SE tasks over five SE activities.

SE activity	SE task		Total
Software requirements & design (Section 4.1)	Requirement ambiguity detection (5)	Class diagram derivation (2)	43
	GUI layout (2)	Requirement classification (7)	
	Requirement completeness detection (1)	Requirement elicitation (1)	
	Requirement engineering (7)	Requirement prioritization (1)	
	Requirement summarization (1)	Requirement traceability (2)	
	Requirement quality assurance (4)	Software modeling (5)	
	Software specification generation (3)	Software specification repair (1)	
	Use case generation (1)		
Software development (Section 4.2)	API documentation smell (1)	API inference (4)	436
	API recommendation (6)	Code comment completion (1)	
	Code completion (40)	Code compression (2)	
	Code editing (5)	Code generation (211)	
	Code recommendation (2)	Code representation (7)	
	Code search (24)	Code summarization (57)	
	Code translation (29)	Code understanding (14)	
	Continuous development optimization (2)	Control flow graph generation (1)	
	Data analysis (1)	Data augmentation (2)	
	Identifier normalization (1)	Method name generation (4)	
	Microservice recommendation (1)	Neural architecture search (1)	
	Program synthesis (13)	Project planning (1)	
	SO post title generation (2)	SO question answering (2)	
	Type inference (1)	Unified development (1)	
Software testing (Section 4.3)	Actionable warning identification (1)	Adversarial attack (3)	253
	API misuse detection (1)	API testing (1)	
	Assertion generation (15)	Binary code similarity detection (4)	
	Code decompilation (8)	Code execution (1)	
	Failure-inducing testing (1)	Fault localization (16)	
	Formal verification (4)	Fuzzing (17)	
	GUI testing (6)	Indirect call analysis (1)	
	Inducing testing (1)	Invariant prediction (1)	
	Mutation testing (13)	NLP testing (7)	
	Penetration testing (4)	Program analysis (1)	
	Program reduction (1)	Proof generation (1)	
	Resource leak detection (1)	Simulation testing (1)	
	Static analysis (6)	Static warning validation (2)	
	Taint analysis (1)	Test generation (50)	
	Test suite minimization (1)	Theorem proving (1)	
	Vulnerability detection (81)	Vulnerable dependency alert detection (1)	
Software maintenance (Section 4.4)	Android permission (1)	App review analysis (2)	247
	Bug report detection (5)	Bug report reproduction (4)	
	Bug triaging (3)	Code clone detection (20)	
	Code coverage prediction (1)	Code evolution (2)	
	Code refactoring (6)	Code review (24)	
	Code smell (1)	Commit message generation (6)	
	Compiler optimization (8)	Debugging (2)	
	Exception handling recommendation (1)	Flaky test prediction (1)	
	Incident management (5)	Issue labeling (2)	
	Log analysis (22)	Log anomaly detection (13)	
	Malware tracking (1)	Mobile app crash detection (1)	
	Outage understanding (1)	Patch correctness assessment (6)	
	Privacy policy (1)	Program repair (82)	
	Report severity prediction (3)	Sentiment analysis (4)	
	Tag recommendation (1)	Technical debt management (1)	
	Test update (3)	Traceability link recovery (1)	
	Vulnerability repair (13)		
Software management (Section 4.5)	Developer behavior analysis (2)	Effort estimation (2)	7
	Software repository mining (1)	Software tool configuration (2)	

4.1.3 *Requirement classification*

Requirement classification refers to the process of categorizing software requirements into different classes or types, such as 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. As early as 2020, Hey et al. [120] proposed NoRBERT, a BERT-based requirement classification approach for both functional and non-functional classes by transfer learning. Khan et al. [121] discussed the performance of LLMs in identifying and categorizing non-functional requirements. To address the issue of limited annotated data in non-functional requirements classification, Rahman et al. [122]

proposed to classify non-functional requirements by extracting features from pre-trained word embedding models. Considering that previous work utilizing LLMs in a black-box manner, Han et al. [123] proposed a requirement classification approach based on BERT and an explainable AI framework. They train a concern extraction model to extract concerns from requirement texts, and utilize explainability to generate explanations for the predictions of the requirement classification model, which is then used to fine-tune BERT for requirement classification.

4.1.4 *Requirement quality assurance*

High-quality requirements are fundamental to the success of software development, providing a clear and detailed specification of what the software system should achieve. To assess the quality of requirements, Lubos et al. [124] empirically explored the effectiveness of LLMs (like Llama 2) to enhance the quality assurance process in the requirements engineering phase. Preda et al. [125] presented an initial study to automate the review of coverage between high-level and low-level software requirements by GPT-3.5 and GPT-4. Poudel et al. [126] utilized several BERT-style LLMs to assess whether design elements adequately satisfy software requirements. Ronanki et al. [127] utilized ChatGPT to evaluate the quality of user stories, which are crucial to capture end-user needs and express requirements in agile software development projects. These studies suggest that LLMs can play a crucial role in automating and enhancing the quality assurance processes across various aspects of requirements engineering.

4.1.5 *Software modeling*

Software modeling refers to the process of creating abstract representations of a software system based on requirements. LLMs have recently been explored for automating various modeling tasks, such as generating and completing UML models from natural language descriptions. For example, Ferrari et al. [128] conducted an exploratory study to assess ChatGPT's ability to generate UML sequence diagrams from real-world natural language requirements. Their results show that ChatGPT performs well in terms of understandability and standard compliance, but struggles with correctness and completeness, especially when facing ambiguous or underspecified requirements. At the same time, Wang et al. [129] conducted a controlled study with 45 undergraduate students to investigate how LLMs assist novice analysts in constructing three types of UML models, including use case diagrams, class diagrams, and sequence diagrams. Similarly, Tinnes et al. [130] proposed RAMC, a retrieval-augmented model completion approach that guides LLMs in completing partial UML models with contextualized change graphs.

4.1.6 *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 to automatically extract software specifications from textual information. For example, Xie et al. [131] conducted 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. This work also conducts a comparative diagnosis of failure cases between LLMs and traditional methods to identify their respective strengths and weaknesses. Considering traditional methods relying on pre-defined templates or grammar rules, SpecGen [132] leverages the code comprehension capabilities of LLMs to generate formal program specifications. SpecSyn [133] is a specification synthesis approach that treats this task as a sequence-to-sequence learning problem, directly translating natural language input into formal specifications.

4.1.7 *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. [134] evaluated the potential of ChatGPT for repairing software specifications in the Alloy declarative language. It aims to assess ChatGPT's capabilities in correcting errors and identify challenges in making it a viable solution. This work demonstrates that ChatGPT successfully addresses unique errors that other tools fail to rectify, although it does not consistently surpass them in total repair count.

In addition to the above-mentioned tasks, researchers also integrate LLMs into software requirements and design from other aspects, including class diagram derivation [135], requirement completeness detection [136], requirement elicitation [137], requirement prioritization [138, 139], requirement traceability [140, 141] and and use case generation [142].

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 *API recommendation*

API recommendation aims to assist developers in identifying and utilizing relevant APIs in their code by analyzing context, intent, and historical usage patterns. This task reduces the cognitive load of remembering extensive API documentation and improves software development efficiency. Recent studies demonstrate the value of LLMs in advancing API recommendation systems for both code and natural language-based developer queries. For example, Wei et al. [143] presented CLEAR, which leverages contrastive learning to optimize BERT for method-level and class-level API recommendation from natural language queries. CLEAR trains BERT by embedding full query sentences and differentiating semantically similar but lexically different queries. Li et al. [144] proposed PTM-APIRec, an extensible API recommendation framework that introduces a novel strategy that encodes both context and API candidates using separate vocabularies. Huang et al. [145] presented a knowledge-guided query clarification approach for API recommendation that integrates LLMs with structured API knowledge graphs (KGs). Chen et al. [146] introduced APIGen, a generative API method recommendation approach based on in-context learning. APIGen augments LLMs with two key components: diverse example selection from lexical, syntactic, and semantic dimensions, and guided API recommendation using reasoning prompts that align task intent with API documentation.

4.2.2 *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. In 2021, Mastropaolo et al. [147] addressed 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 the T5 model outperforms the n-gram model in all the tested code comment completion scenarios.

4.2.3 *Code completion*

Code completion aims at speeding up code writing by predicting the next code token the developer is likely to write. Researchers usually focus on improving the accuracy of the generated predictions. For example, TeCo [148] attempts to generate the next statement in a test method by fine-tuning CodeT5 with code semantic information. CCTEST [149] focuses on improving the performance of off-the-shelf code completion systems (such as Copilot and CodeGen) by utilizing a mutation strategy to detect erroneous outputs and a repair mechanism to fix these outputs. Besides, repository-level code completion has always been challenging due to complicated contexts from multiple files in the repository. LLMs demonstrate the potential in this task by integrating advanced strategies, such as information retrieval [150–153], static analysis [154], and reinforcement learning [155]. There are also many empirical studies analyzing the actual performance of LLMs in code completion. For example, Ciniselli et al. [156, 157] investigated the effectiveness of T5 and RoBERTa for code completion at different granularity levels,

from single tokens to entire code blocks. Similarly, van Dam et al. [158] explored the impact of contextual information on three LLMs (UniXcoder, CodeGPT, and InCoder) for both token-level and line-level code completion.

4.2.4 *Code editing*

Code editing 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 [159]. 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, Li et al. [160] designed a pre-training specialized for code editing by rewriting mutated versions of code snippets into their correct form. They then introduce CodeEditor initialized with CodeT5 and fine-tune it on two code editing scenarios, i.e., code-to-code editing and comment&code-to-code editing. Besides, Gupta et al. [161] proposed Grace to address code editing by leveraging the generative capabilities of LLMs on previously related edits. By mirroring the behavior of developers, Liu et al. [162] introduced a hybrid approach SARGAM for automated code editing, which consists of three steps. SARGAM first retrieves similar code patches from a large repository and uses them to guide LLMs (such as PLBART, CoditT5, and NatGen) to generate a new patch, which is further modified to fit the exact context. These studies demonstrate the potential of LLMs to automate complex editing tasks, offering developers an efficient way to handle repetitive and time-consuming code modifications.

4.2.5 *Code generation*

Code generation plays a pivotal role during software development and has always been the primary focus in the application of LLMs, such as AlphaCode [62] and CodeGen [76]. In general, recent advancements in LLM-based code generation focus on requirement-guided generation, execution-guided, and empirical studies. Below, we will introduce these studies in detail.

Requirement-guided code generation. In the early stages of development, LLMs commonly take a natural language description as the input and return the correct code snippet, which is evaluated by corresponding unit tests [163–166]. For example, ArchCode [167] leverages in-context learning to interpret software requirements from textual descriptions for LLM-based code generation. ClarifyGPT [168] attempts to detect ambiguous requirements, and prompt LLMs to specific clarifying questions, which are used to refine the requirements and generate more accurate code solutions. Besides, AceCoder [169] directs LLMs to analyze the given requirement and generate intermediate artifacts for better code generation, such as test cases, that help clarify the requirement. Inspired by the way humans typically use planning to break down complex problems, Jiang et al. [164] introduced a self-planning code generation approach to help LLMs understand complex intents and simplify problem-solving.

Execution-guided code generation. Inspired by the process of human programming, developers utilize the execution results to guide LLMs in refining generated code iteratively [170–172]. For example, Self-Edit [173] validates generated code with available test cases, and the results are fed into LLMs as supplementary comments for further improvements and corrections. Besides, Dong et al. [174] introduced a self-collaboration code generation framework, which assembles a team consisting of three ChaGPT roles (i.e., analyst, coder and tester). Given a requirement, the analyst decomposes it into several manageable subtasks and develops a high-level plan. The coder generates code according to the plan provided by the analyst, and refines code according to the test reports provided by the tester. The tester receives the code generated by the coder and subsequently produces a test report.

Empirical evaluation of LLMs. A mass of empirical studies are conducted to investigate the actual code generation capabilities of LLMs from different aspects, such as robustness [175, 176], efficiency [177], human study [178–180], ChatGPT [181, 182], prompt engineering [183], benchmarks [184, 185], domain-specific generation [186]. For example, Mastropaolo et al. [175] conducted an empirical study to investigate how robust GitHub Copilot is in generating consistent code when provided with semantically equivalent natural language descriptions. Kou et al. [180] empirically explored the attention alignment of LLMs and human programmers during code generation.

Others. Recently, in addition to the above-mentioned studies, there has been an exploration in cutting-edge areas such as repository-level generation [187–189], retrieval-augmented generation [190] and agent-based generation [191–193].

4.2.6 *Code representation*

Code representation focuses on transforming source code into structured, machine-understandable vector formats that preserve its syntactic and semantic properties [194, 195]. Unlike code understanding, which emphasizes rea-

soning over code to extract functional or behavioral insights, code representation serves as the foundational step to facilitate such reasoning. Recent studies typically capture key code features, such as syntax trees, control flow, identifier semantics, or API behavior, thereby enabling LLMs to perform better across a wide range of downstream tasks, such as code summarization and translation. For example, Agarwal et al. [196] introduced a structured code representation method that encodes concrete syntax trees into serialized sequences for LLM adaptation. Their plug-and-play approach preserves structural granularity while reusing existing Transformer architectures, enabling better data efficiency, especially in low-resource scenarios such as code translation and summarization. He et al. [197] evaluated a wide range of LLMs, including general-purpose models like RoBERTa and domain-specific ones such as CodeBERT and GraphCodeBERT, for representing Stack Overflow posts. They find that no single model consistently outperforms others and propose SOBERT, which is obtained by continuing pre-training on Stack Overflow text. For behavior-based malware analysis, Cui et al. [198] proposed API2Vec++, a graph-based API representation framework that captures both intra- and inter-process behavior via temporal API property graphs (TAPG) and temporal process graphs (TPG). API paths extracted from these graphs are embedded using BERT, yielding behavior-sensitive representations that significantly boost malware detection and classification accuracy, particularly for multi-process malware. To address the semantic sparsity of low-frequency identifiers, Lin et al. [199] proposed VarGAN, an adversarial training framework that enhances variable name representations by aligning rare identifiers with their frequent counterparts in the vector space. VarGAN improves similarity and relatedness performance on the IdBench benchmark, and demonstrates consistent gains in code summarization and clone detection tasks. Overall, these studies underline a growing consensus that integrating syntactic structure, execution behavior, and identifier semantics into pre-trained code embeddings of LLMs is essential to achieving expressive and generalizable code representations.

4.2.7 *Code search*

Code search is an essential practice in software development where developers search for specific pieces of source code within extensive codebases. Given a query, this activity is undertaken to find code snippets, functions, classes, or entire files for potential reuse. Overall, existing studies utilizing LLMs for code search can be classified into two categories. First, researchers utilize training objectiveness to help LLMs learn more effective encoding representation, primarily focusing on contrastive learning [200–203]. For example, CodeRetriever [204] learns semantic representations for function-level code search with unimodal and bimodal contrastive learning. 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. Similarly, CoCoSoDa [205] helps LLMs better align code snippets and natural language queries for code search based on multimodal contrastive learning and data augmentation. Second, some empirical studies are conducted to investigate the potential of LLMs in code search. For example, Salza et al. [206] explored how transfer learning can be applied to code search tasks by pre-training and fine-tuning BERT. Chi et al. [207] evaluated existing code search LLMs in industry requirements based on their adaptability, scalability, robustness, and semantic sensitivity.

4.2.8 *Code summarization*

Code summarization takes as input a code snippet provided by the developer and automatically generates a higher-level summary in natural language form. These summaries are usually utilized to enhance the comprehension of software systems, thereby facilitating their maintenance. Integrating LLMs into code summarization can provide more accurate and natural summaries of source code, as LLMs can leverage their general knowledge on vast amounts of textual data to more accurately infer and generate natural language summaries of source code.

Existing LLM-based studies mainly fall into three areas: training optimizations, prompt engineering, and empirical studies. First, for open-source LLMs, researchers utilize pre-training or fine-tuning methods to enhance LLMs' capabilities of the alignment between code and natural language [208]. For example, ESALE [209] introduces three code summarization-specific pre-training tasks (including two general tasks ULM and MLM, and one domain-specific task AWP) to help LLMs learn the code-summary alignment. Other training strategies include adapter tuning [210] and joint training [208]. Second, for commercial LLMs, prompt engineering is usually utilized to provide relevant information in a zero-shot or few-shot manner [211, 212]. For example, Ahmed et al. [213] explored enhancing the code summarization performance of LLMs by explicitly adding semantic information to the prompts, such as parameter names, return types, and simple control flows. Rukmono et al. [214] integrated static code analysis into the chain-of-thought prompting to generate component-level summaries of software systems. Third, empirical studies are conducted to explore the potential of LLMs from different aspects, such as explainability [215], metrics [216], binary code summarization [217]. For example, Sun et al. [218] evaluated the performance of ChatGPT on code

summarization, and Li et al. [215] utilized eye-tracking metrics from human participants to measure whether they concentrate on the same parts of code as LLMs when generating summaries.

4.2.9 *Code translation*

Code translation refers to the process of converting code from a source language into a target language while preserving the original functionality and behavior of the program. This task is crucial in software development, especially when developers want to migrate software systems to a different platform. Yang et al. [219] introduced UniTrans, which introduces test case generation to enhance the accuracy of code translation and provides mechanisms for iterative repair of translation errors. TransMap [220] attempts to detect semantic mistakes in code translated by models like Codex and ChatGPT. Regarding empirical studies, Pan et al. [221] proposed a taxonomy of translation bugs introduced by LLMs, while Jiao et al. [222] provided a detailed analysis of code translation across four levels: token level, syntactic level, library level, and algorithm level.

As a specialized form of code translation, code co-evolution focuses on updating code snippets in a target programming language to reflect changes made in the source language. Instead of generating code from scratch, code co-evolution learns an edit sequence to update existing code snippets. For example, Zhang et al. [223] proposed Codeditor, which fine-tunes CoditT5 to align the code edits for two programming languages (Java and C#) from eight open-source projects.

4.2.10 *Code understanding*

Code understanding refers to the task of analyzing source or binary code to infer its semantic structures, behaviors, and relationships within the codebase. It aims to encode functional and structural semantics into representations usable for diverse downstream SE tasks, such as code summarization, clone detection, and code completion. For example, Nam et al. [224] developed an IDE-integrated conversational UI that leverages ChatGPT towards code understanding. Their approach enables developers to access code explanations, API details, and usage examples directly within the IDE context, resulting in significantly improved task completion rates compared to web searches, as demonstrated by a user study involving 32 participants. Wan et al. [225] conducted a structural analysis of LLMs such as CodeBERT and GraphCodeBERT for source code understanding from three perspectives: attention mechanism analysis, embedding probing, and syntax tree induction. To facilitate binary code understanding, Artuso et al. [226] introduced BinBert, which leverages symbolic execution information to enhance semantic understanding during pre-training and employs fine-tuning techniques for adapting general pre-trained knowledge to task-specific binary analysis tasks. Shi et al. [227] introduced ShellGPT, a specialized LLM tailored for shell language understanding in IT operations. ShellGPT employs domain-specific pre-tokenization and an equivalent command learning strategy to enhance model accuracy in tasks such as command recommendation, command correction, and natural language to shell command translation.

4.2.11 *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. For example, Jain et al. [228] explored the integration of LLMs such as GPT-3 and Codex in generating code from natural language descriptions of programmer intent. Liventsev et al. [229] introduced SED, a framework to enhance the program synthesis capabilities of LLMs like OpenAI Codex. In SEIDR, a draft program is first synthesized based on a high-level description and is then executed to validate its correctness. If the program fails, specific instructions are generated to guide LLMs in fixing the issues until it meets the required specifications. Besides, Vella et al. [230] explored the integration of multiple LLMs within evolutionary algorithms to enhance program synthesis tasks.

In addition to the above-mentioned tasks detailed above, researchers also integrate LLMs into software development from other aspects, including API documentation smells [231], API Inference [232–234], code compression [235, 236], continuous development optimization [237], identifier normalization [238], microservice recommendation [239], neural architecture search [240], performance data synthesize [241], SO post title generation [242], type inference [243], and unified development [244].

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 *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 assertions serve as a part of the solution to test oracle problems, which provide the criteria to determine whether a program behaves as intended.

As early as 2022, Tufano et al. [245] proposed 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. Finally, they fine-tune it on a dataset mined from more than 9 thousand open-source GitHub projects containing unit test cases defined with JUnit. Despite promising, the previous approach [245] struggles to find real-world bugs. In 2022, Dinella et al. [246] proposed a transformer-based approach TOGA to infer both exceptional and assertion test oracles based on the context of the focal method. TOGA fine-tunes CodeBERT to classify exceptional oracles and rank assertion oracles. Unlike previous studies fine-tuning LLMs for assertion generation, Nashid et al. [25] introduced Cedar, a few-shot learning method that utilizes a prompt-based approach for both test assertion generation and program repair. Cedar retrieves relevant demonstrations and conducts prompts to query Codex to generate assertions in a few-shot manner. More recently, Zhang et al. [247] introduced a retrieval-augmented automated assertion generation approach that incorporates a novel hybrid retriever leveraging both lexical and semantic similarity to identify the most relevant assertions from external codebases. Furthermore, Zhang et al. [18] proposed AG-RAG, an end-to-end framework that jointly optimizes the retriever and generator through collaborative learning, thereby enhancing the overall performance of assertion generation.

Regarding empirical studies and benchmarking, He et al. [248] investigated the effectiveness of seven test-to-code traceability techniques in assertion generation and enhanced existing datasets for training and evaluating assertion generation models, such as T5. Besides, Pulavarthi et al. [249] introduced AssertionBench, a benchmark to assess the effectiveness of LLMs in generating assertions for hardware verification. AssertionBench focuses on generating high-quality assertions, which are crucial for detecting and diagnosing design bugs, particularly in complex hardware systems. Endres et al. [250] explored the potential of LLMs, such as GPT-4, to convert informal natural language descriptions of program behavior into formal postconditions that can be used for software verification. Hossain et al. [251] conducted a large-scale investigation of the ability of LLMs to automatically generate test oracles by fine-tuning LLMs (including CodeGPT-110M, CodeParrot-110M, CodeGen-350M, PolyCoder-4B, Phi-1-1.3B, CodeGen-2B and PolyCoder-2.7B) with six different prompts. Recently, Zhang et al. [17] conducted the first extensive study on fine-tuning five LLMs to assertion generation across two independent datasets, and further proposed a simplistic retrieval-and-repair-enhanced LLM-based approach that reformulates assertion generation as a program repair task for retrieved similar assertions.

4.3.2 *Code decompilation*

Decompilation is a reverse engineering process that involves transforming a binary executable into a human-readable form closely resembling its original source code. This process has significant applications in the security domain, such as malware analysis and vulnerability detection, as well as in software development, including code reuse and software supply chain analysis. Although decompilation is utilized across various phases of software engineering, following prior work [3], we have classified it under software testing.

There mainly exist two key challenges during the decompilation process, i.e., recovering variable names within binary executable files and producing human-readable code. To address the first issue, Xu et al. [252] introduced LmPa to improve the recovery of variable names and other high-level information by combining LLMs with program analysis. LmPa utilizes LLMs to provide meaningful names for variables, which are then refined and validated with program analysis techniques, ensuring that the recovered names are contextually appropriate and semantically accurate. To address the second issue, Wong et al. [253] automatically refined the accuracy and quality of decompiled C code by combining LLMs with traditional decompilation techniques. LLMs are utilized to fix syntax, inference, and memory errors in decompiled output, making it compatible with standard C/C++ compilation. A similar work is DeGPT [254], which designs a three-role mechanism to maximize the optimization capabilities of LLMs

on decompiler output. Researchers also make efforts to integrate LLMs into code decompilation from different perspectives, including domain LLMs [255–257], and WebAssembly [258], binary code understanding [259].

4.3.3 *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. [260] proposed 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.

4.3.4 *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 [261, 262]. 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., encoder-like and GPT-like models.

Regarding encoder-based localization, as early as 2021, Zhu et al. [263] proposed TroBo, a CodeBERT-based cross-project bug localization approach by leveraging both bug reports and source code. In 2022, Ciborowska et al. [264] discussed how to optimize BERT for changeset-based bug localization. They explore various design choices for applying BERT, including how to encode code changes and match error reports to specific code changes to enhance accuracy. To support cross-language cross-project bug localization, Chandramohan et al. [265] fine-tuned UniXcoder with a contrastive learning object to enhance the representation of both bug reports and source code. Regarding GPT-based localization, Wu et al. [266] conducted a comprehensive empirical study on the large-scale open-source program Defects4J, evaluating the potential of OpenAI GPT LLMs (i.e., ChatGPT-3.5 and ChatGPT-4) in fault localization research. Besides, to handle large codebases, Kang et al. [267] introduced AutoFL, to select parts of the project and apply a post-processing step to match ChatGPT's answers with actual code elements. Yang et al. [268] introduced LLMAO, the first CodeGen-based approach that locates buggy lines without relying on traditional test coverage information. FuseFL [269] leverages ChatGPT to deliver explainable fault localization by integrating multiple information sources, including test case outcomes and code descriptions.

4.3.5 *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 injects these inputs into the system and then monitors for exceptions such as crashes or information leakage. Given the large number of studies on LLM-based fuzzing, we organize existing studies into six categories based on application scenarios and briefly introduce the context and representative studies for each category.

DL library fuzzing. DL libraries, such as TensorFlow and PyTorch, are foundational to the burgeoning field of DL and play a pivotal role in our daily lives due to the widespread adoption of DL systems. Traditional fuzzing techniques often struggle to satisfy both the input language semantics and the DL API input constraints for tensor computations. To address this, Deng et al. [270] introduced TitanFuzz in 2023, the first approach to leverage LLMs directly for generating input programs for fuzzing DL libraries. For any given target API, TitanFuzz initially uses an LLM to generate a list of high-quality seed programs for fuzzing by querying the Codex model with a step-by-step prompt and sampling multiple completions. 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. Deng et al. [271] proposed FuzzGPT, which utilizes LLMs to generate unusual programs based on historically bug-triggering programs. FuzzGPT features three variants: few-shot learning, zero-shot learning, and fine-tuning, leveraging different LLMs, including Codex and CodeGen. Additionally, FuzzGPT can utilize the directive-following capabilities of ChatGPT to generate atypical programs.

Compiler fuzzing. Compilers form the foundation of modern software systems by translating high-level source code into machine code, a lower-level language that computers can execute. Thus, the correctness of a compiler is crucial as it ensures the accurate and efficient execution of the intended functionality of the software. In 2023, Yang et al. [272] introduced 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. Eom et al. [273] introduced CovRL, a novel approach for fuzzing JavaScript engines by integrating coverage-guided reinforcement learning with LLMs. CovRL leverages coverage feedback to guide the mutation process, aiming to improve the discovery of vulnerabilities while minimizing syntax and semantic errors.

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. [274] explored the opportunity of interacting with 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, to achieve structure-aware mutations concerning the state machine and input structure of the protocol. To fuzz Internet of Things (IoT) devices, LLMIF [275] utilizes an LLM to analyze protocol specifications and generate relevant test cases.

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. [276] introduced 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. Besides, considering that traditional fuzzers (e.g., AFL) struggle to generate structured test inputs efficiently and at scale, in 2023, Hu et al. [277] introduced CHATFUZZ, a grey-box fuzzing tool leveraging ChatGPT to enhance test input quality and effectiveness. In parallel to CHATFUZZ, Dakhama et al. [278] introduced an innovative approach that combines LLM and search-based fuzzing, 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. Similarly, LLAMAFUZZ [279] attempts to enhance traditional greybox fuzzing by leveraging LLMs to generate structured data, which is often a challenge for traditional fuzzing methods that rely on random mutations.

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. [280] conducted 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, 189628 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.

Others. Researchers also integrate LLMs into other fuzzing scenarios, including smart contracts [281], kernel fuzzing [282], and BusyBox [283]. For example, Oliinyk et al. [283] utilized LLMs to generate initial test seeds for fuzzing BusyBox, a widely-used open-source software suite for Linux-based embedded devices, aligning these seeds more closely with the expected inputs of BusyBox's various utilities.

4.3.6 *GUI testing*

Graphical user interface (GUI) testing ensures the accuracy and reliability of a mobile application's visual interface [284]. It involves verifying that interactions with components, such as buttons and text boxes, yield the expected outcomes. The aim is to confirm the correct display of information and intended user interactions. GUI testing can be either manual or automated, often using metrics such as error detection and code coverage to assess performance. Its primary goal is to guarantee the stability and reliability of an application's visual and interactive elements.

In 2022, to address the diverse and semantic requirements for valid inputs in GUI testing, Liu et al. [285] proposed 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. Unlike QTypist [285] focusing on text input generation, in 2023, Liu et al. [286] proposed 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. Besides, VisionDroid [287] is a vision-driven automated GUI testing approach that leverages multimodal LLMs that can understand both visual and textual information to detect non-crash functional bugs in mobile apps. DroidAgent [288] leverages LLMs to simulate user interactions by setting and executing tasks that mirror realistic user behaviors and intents.

4.3.7 *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 particular, μBERT [289] represents the first attempt to integrate LLMs into mutation testing by using CodeBERT to generate natural mutants. 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. Inspired by μBERT, Ibrahimzada et al. [290] proposed BugFarm to transform arbitrary code into complex bugs by querying LLMs to mutate code in multiple locations. VULGEN [291] focuses on generating realistic software vulnerabilities by combining pattern mining with LLMs. It first collects vulnerability-fixing examples to mine the patterns for vulnerability injection, and then leverages CodeT5 to learn the localization of injection. Similarly, Garg et al. [292] investigated the relationship between software vulnerabilities and mutants generated by CodeBERT.

Regarding empirical studies, Wang et al. [293] investigated the performance of LLMs in generating mutations based on various criteria, including usability, fault detection potential, and their relationship with real bugs. Geng et al. [211] empirically investigated the feasibility of utilizing LLMs to generate comments that can reflect multiple developer intents by in-context learning. Besides, Tian et al. [294] explored the effectiveness and efficiency of LLMs in detecting equivalent mutants.

4.3.8 *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. As early as 2020, He et al. [295] introduced SIT, a metamorphic testing approach that leverages BERT to validate machine translation software. SIT uses BERT to generate a set of similar sentences by altering a single word in the source sentence, helping to assess the structural consistency of translations. Similarly, motivated by the idea that sentences with distinct meanings should yield different translations, Gupta et al. [296] proposed PatInv, which generates syntactically similar but semantically different sentences using BERT. In 2022, Sun et al. [297] introduced CAT, a BERT-driven word-replacement method for enhancing machine translation. Beyond these studies on machine translation, researchers have also explored using LLMs, particularly BERT, in other NLP testing scenarios, including named entity recognition software [298], textual content moderation software [299], dialogue systems [300], and question answering systems [301].

4.3.9 *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. PentestGPT [302] is 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 maintains a high-level overview of the testing process; (2) the generation module translates specific subtasks from the reasoning module into concrete instructions; and (3) the parsing module operates as a supportive interface between the user and the other two core modules. PTGroup [303] automates complex penetration testing scenarios by utilizing LLMs to interpret multiple testing strategies simultaneously and designing multiple prompt

chains for different penetration testing tasks. Meanwhile, Happe et al. [304] explored the use of LLMs like GPT-3.5 to enhance penetration testing, focusing on both high-level task planning and low-level attack execution.

4.3.10 *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. [305] proposed 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.

4.3.11 *Static analysis*

Static analysis refers to the process of examining source code without dynamic execution, enabling early detection of software defects or optimization of performance. Traditional static analysis tools, while effective, often suffer from scalability issues, high false positive rates, and the need for extensive manual modeling of code semantics. Recent studies seek to overcome these limitations by either improving the underlying analysis infrastructure or introducing LLMs to augment reasoning and automation capabilities.

For instance, Mohajer et al. [306] introduced SkipAnalyzer, an LLM-powered static analysis tool that integrates bug detection, false positive filtering, and automated patch generation. Built on ChatGPT, SkipAnalyzer significantly improves precision over conventional tools like Infer, especially for null dereference and resource leak bugs. Hao et al. [307] proposed E&V, a novel prompting framework that enables LLMs to perform static analysis by simulating pseudo-code execution. E&V first emulates analysis procedures across multiple languages by converting static analysis logic into human-readable pseudo-code, and incorporates a self-verification mechanism to mitigate hallucination errors. At the same time, Li et al. [308] proposed LLift, a framework that integrates LLMs with static analyzers to identify use-before-initialization (UBI) bugs in the Linux kernel. LLift employs post-constraint guided path analysis to reduce exploration paths in path-sensitive analysis and applies prompt-based reasoning to LLMs to resolve undecidable cases from traditional analyzers. Li et al. [309] explored the use of ChatGPT as an assistant to reduce false positives of a traditional static analyzer, UBITect, which specializes in detecting UBI bugs in the Linux kernel. They employ carefully designed progressive prompts, task decomposition, and chain-of-thought strategies to improve reliability, which helps GPT-4 correctly identify 16 out of 20 false positives. Recently, Fang et al. [310] conducted a systematic evaluation of LLMs' ability to analyze both obfuscated and unobfuscated code, assessing their robustness in defensive and security-focused static analysis.

4.3.12 *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. We summarize these advanced studies that employ LLMs as follows.

Fine-tuning LLM-based generation. AthenaTest [311] represents the first work to apply LLMs in the field of test generation. AthenaTest first pre-trains a BART-like model from scratch on a large corpus of English and Java datasets and fine-tunes it on a labeled Java dataset. A3Test [312] builds upon the PLBART checkpoint, further pre-training it on the Atlas dataset to learn assertion knowledge, followed by fine-tuning for the test case generation

task. Rao et al. [313] introduced CAT-LM, a GPT-style LLM that is specifically trained to learn the mapping between code and its corresponding test cases.

Prompt-based generation. For example, Schäfer et al. [314] introduced TestPilot, an LLM-based end-to-end adaptive test generation technique for JavaScript. TestPilot first constructs a prompt with the information of the function under test and its usage examples. TestPilot then queries LLMs to generate test cases, which are validated through dynamic execution, and refined with new prompts containing failure information. MuTAP [315] utilizes mutation testing to augment the prompts, guiding LLMs to generate test cases that can detect software bugs. SymPrompt [316] enhances the performance of LLMs in generating high-coverage test cases by utilizing code-aware prompts, which are dynamically constructed with the execution paths of the method under test. Pizzorno et al. [317] introduced CoverUp, which integrates coverage analysis into prompts to iteratively generate Python regression tests. HITS [318] decomposes the focal methods into slices and queries LLMs to generate test cases slice by slice.

Conversation-driven generation with ChatGPT. To address issues of invalid test cases and low coverage, Gu et al. [319] introduced ChatGPT-based TestART, which integrates test generation with an iterative template-based repair process. To mitigate common issues such as invalid or incomplete tests generated by LLMs, Karmarkar et al. [320] introduced TestRefineGen to generate tests based on textual descriptions while ensuring confidentiality is maintained. Chen et al. [321] proposed ChatUniTest, an automated unit test generation tool based on ChatGPT under the generation-validation-repair framework. Ni et al. [322] introduced CasModaTest, an end-to-end LLM-based framework, by dividing the unit test generation task into two cascaded ones: test prefix generation and test oracle generation. To enhance test coverage and improve the efficiency and effectiveness of automated testing, Lemieux et al. [323] introduced CODAMOSA, which integrates Codex with SBST to generate Python test cases.

Empirical evaluations of LLM-based test generation. Xiao et al. [324] empirically explored how LLMs can be integrated into traditional search-based unit test generation workflows, including the initial phase, the test generation period, and the test coverage plateaus. Tang et al. [325] provided a comprehensive comparison between ChatGPT and EvoSuite based on several factors: correctness, readability, code coverage, and bug detection capabilities. Similarly, Yang et al. [326] explored the impact of prompt designs, comparison of open-source and commercial LLMs, and in-context learning. More recently, Shang et al. [16] conducted the first multi-task empirical study on fine-tuning LLMs for unit testing, covering three unit testing tasks, five benchmarks, eight evaluation metrics, and 37 widely used LLMs. Their findings highlight the potential of fine-tuning in this domain and provide practical guidelines for developing LLM-based approaches to unit testing. Researchers also conduct more studies from different aspects, such as the investigation of ChatGPT [327,328], prompt engineering [329], human study [330], GUI text input [331], and security tests [332].

4.3.13 *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, Pan et al. [333] proposed 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. Results show that UniXcoder/Cosine is the best LTM configuration when considering both effectiveness and efficiency. Besides, LTM outperforms prior studies by achieving a significantly higher fault detection rate and faster minimization time.

4.3.14 *Vulnerability detection*

Vulnerability detection, also known as vulnerability prediction, aims to identify potential security bugs in software systems. Vulnerability detection is critical for protecting security-critical software systems from malicious attacks, providing the foundation for timely patching of reported security vulnerabilities before they may be exploited (discussed in Section 4.4.13). In the literature, learning-based vulnerability detection approaches [334, 335] have been proposed to detect security vulnerabilities by extracting meaningful features and performing predictions automatically. For example, Li et al. [335] proposed IVDetect, to perform fine-grained vulnerability prediction based on a FA-GCN model and GNNExplainer. However, such learning-based approaches are limited by the amount of training data, resulting in capturing a suboptimal vector representation of source code.

Thus, a mass of vulnerability detection approaches empowered with LLMs have been proposed. For example, LineVul attempts to detect vulnerabilities at the line level by fine-tuning CodeBERT and utilizing its attention mechanism to pinpoint vulnerable lines. In parallel to LineVul, VulBERTa is an encoder-only Transformer-based vulnerability detection approach by pre-training and fine-tuning RoBERTa. Besides, VulLLM [336] utilizes multi-task instruction tuning to adapt LLMs in vulnerability detection, which includes two auxiliary tasks: vulnerability localization to pinpoint the specific vulnerable parts of the code, and vulnerability interpretation to understand the underlying issues. GPTScan [337] attempts to detect smart contract logic vulnerabilities by combining GPT and static analysis. In the literature, researchers also conduct some empirical studies. For example, Steenhoek et al. [338] conducted an empirical study to investigate the performance of DL models in detecting software vulnerabilities from three aspects, i.e., model capabilities, training data, and model interpretation. This study includes 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. In 2023, Zhang et al. [339] explored the performance of ChatGPT in software vulnerability detection with different prompts. Meanwhile, Noever et al. [340] evaluated the capabilities of LLMs, particularly GPT-4, in detecting software vulnerabilities, comparing their performance against traditional static code analysis tools.

In addition to the above-mentioned tasks detailed above, researchers also integrate LLMs into software testing from other aspects, including API testing [341], test oracle [342, 343], code execution [344], vulnerable dependency alert detection [345] theorem proving [346], DL adversarial attack [347], root cause analysis [348, 349], actionable warning identification [350], and program reduction [351].

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 *Bug report detection*

Duplicate bug report detection attempts to automatically identify and label duplicate bug reports in issue tracking systems, enabling developers to avoid redundantly dealing with the same issues. In the report detection process, researchers utilize LLMs to compare and analyze the similarity between different bug reports to determine if they describe the same defect or problem. For example, in 2023, Zhang et al. [352] introduced Cupid, which integrates the traditional detection 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. Meanwhile, Plein et al. [353] conducted empirical research, as outlined in [353], 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.

4.4.2 *Bug reproduction*

Bug reproduction, also known as bug replay, refers to the process of reproducing or recreating software defects or issues based on the information provided in a bug report. The process is crucial for software maintenance as it enables developers to understand, replicate, and fix the defects reported.

In 2023, Kang et al. [354] introduced 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. Focusing on mobile application crashes, Huang et al. [355] presented CrashTranslator, which leverages LLMs to predict the necessary exploration steps required to reproduce a crash and uses reinforcement learning to improve the search process and mitigate inaccurate predictions. Furthermore, Feng et al. [356] proposed a lightweight approach AdbGPT to automatically reproduce Android bugs from bug reports without any training.

4.4.3 *Code clone detection*

Code clone detection attempts to identify duplicate or similar code segments within a software codebase based on code similarity analysis. Zhang et al. [357] conducted the first empirical study to explore the potential of ChatGPT and GPT-4 in the task of code clone detection. Moumoula et al. [358] investigated the capabilities of four LLMs for detecting code clones across different programming languages with eight different prompt configurations. Dou et al. [359] explored different aspects of LLMs' capabilities by examining five different perspectives: simple

prompts, one-step chain-of-thought prompts, multi-step chain-of-thought prompts, code embeddings, and multiple programming languages.

4.4.4 *Code refactoring*

Code refactoring refers to the process of modifying a program's internal structure without altering its external behavior, with the goal of improving readability, maintainability, and reducing complexity [360]. While traditionally a manual and error-prone activity, recent advancements integrate LLMs and learning-based frameworks to enhance automation and developer acceptance. Liu et al. [361] proposed RefBERT, a two-stage LLM-based framework for automatic variable rename refactoring. RefBERT leverages constrained masked language modeling and contrastive learning to first predict sub-token counts and then generate meaningful variable names. Shirafuji et al. [362] explored the potential of GPT-3.5 to generate semantically equivalent, complexity-reduced code variants with few-shot prompting. Zhang et al. [363] proposed a hybrid knowledge-driven framework combining rule-based AST analysis with LLM prompts to refactor Python code into idiomatic forms. Pomian et al. [364] introduced EM-Assist, a novel plugin for IDEs that supports extract method (EM) refactoring by combining LLM suggestions with static analysis and program slicing. EM-Assist filters hallucinated LLM outputs and achieved a 53.4% match rate with real developer refactorings across 1752 instances. Overall, these studies reflect a shift from syntax-driven and rule-based refactoring to intelligent, context-aware models powered by LLMs, making refactoring more accessible and aligned with human intent.

4.4.5 *Code review*

Code review is a critical process in the software development lifecycle, with the aim of enhancing code quality by identifying and resolving issues before the code is merged into the main codebase. In code review practices, developers are tasked with thoroughly examining, understanding, and executing the code under review. This process involves evaluating various aspects of the code, such as its logic, functionality, performance, and style.

To reduce human review efforts, early work in this community primarily focuses on training domain LLMs based on T5 to automate the code review process [365–367]. For example, Li et al. [366] introduced AUGER to generate code review comments by pre-training T5 with a masked language modeling object. Tufano et al. [365] pre-trained and fine-tuned T5 to support different tasks, such as comment generation and code refinement. Initialized with CodeT5, CodeReviewer is pre-trained with four pre-training tasks specifically designed for the code review scenario: diff tag prediction, denoising code diff, denoising review comment, and review comment generation. Researchers also conduct some empirical studies from different aspects [368, 369]. For example, Guo et al. [370] explored the potential of ChatGPT in code review tasks, with a specific focus on code refinement based on code reviews. This study involves various research aspects, including the impact of prompt and temperature settings, the comparison with CodeReviewer, the qualitative analysis, and the analysis of root causes.

4.4.6 *Commit message generation*

Commit message generation attempts to create natural language descriptions for code commits [371]. 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. [372] introduced CommitBERT to generate messages by using CodeBERT as the initial weight during training. Wang et al. [373] introduced ExGroFi to fine-tune LLMs by incorporating the correlation between commits and issues during the training process. Li et al. [374] utilized the reasoning capabilities of LLMs to generate commit messages by framing it as a knowledge-intensive reasoning task. Regarding empirical studies, Xue et al. [375] explored the effectiveness of different LLMs in generating commit messages, and Lopes et al. [376] focused on ChatGPT.

4.4.7 *Compiler optimization*

Compiler optimization focuses on enhancing the performance, size, or energy efficiency of compiled code by applying transformation passes or tuning optimization strategies [377]. Traditional approaches often rely on hand-crafted heuristics or costly search-based methods, but recent advances in LLMs offer promising new directions. For example, Cummins et al. [378] trained a 7B-parameter transformer from scratch to optimize LLVM intermediate representation (IR) for code size. Their model takes unoptimized IR as input and outputs both an optimization pass sequence and the expected instruction counts without requiring any compiler executions during inference. Further extending

this work, Cummins et al. [379] presented LLM Compiler, a family of open-source LLMs explicitly designed for compiler optimization. LLM Compiler is first pre-trained on 546B tokens of compiler-centric data, including LLVM-IR and assembly, and further fine-tuned for tasks like flag tuning and disassembly. Complementing these efforts, Tu et al. [380] introduced LLM4CBI, which explores LLMs for isolating compiler bugs via test program mutation, a foundational step towards robust optimization. LLM4CBI leverages reinforcement learning and program analysis to guide prompt engineering for LLMs, outperforming prior methods in bug isolation effectiveness on both GCC and LLVM.

4.4.8 *Log analysis*

Logs record events that occur within a system, including user actions, system activities, error messages, and security alerts. Log analysis involves automatically examining and interpreting this log data, which is essential for developers to understand system status and diagnose potential problems. Existing approaches can be categorized into fine-tuning [381, 382], few-shot [383–387], and zero-shot learning [388–390]. For example, Le et al. [383] explored using prompt-based few-shot learning for log parsing, leveraging the flexibility of prompts to adapt to various log formats with minimal training examples. Researchers also empirically explore the potential of LLMs in log analysis, including the impact of data resampling [391], structuring logs [392], detecting unusual behaviors [392,393], log-based question-answering [394], and interpreting logs [395].

4.4.9 *Log anomaly detection*

Log anomaly detection aims to identify abnormal system behaviors by analyzing logs generated during system execution. These semi-structured logs capture event sequences, system states, and user actions, making them essential for debugging, fault diagnosis, and intrusion detection. However, challenges such as noisy log formats, imprecise template parsing, and the semantic gap between templates and parameters complicate the effective detection of anomalies. To overcome these limitations, recent research has moved toward integrating LLMs with advanced strategies, such as hierarchical representations and prompt engineering, for more effective anomaly detection. Building on the need to preserve full log semantics, Chai et al. [396] proposed LogSer, which reduces information loss during parsing and employs BERT for global and local anomaly detection by introducing token-level granularity control and parameter-aware embedding strategies. At the same time, Wang et al. [397] introduced DeepUserLog, which targets user logs with abundant and irregular key-value pairs by extracting semantic features directly from logs without relying on fixed templates. This design enables DeepUserLog to operate effectively in domains where template-based parsing is ineffective, such as transaction logs or personalized user activity traces. To mitigate the resource demands of LLMs, He et al. [398] introduced LogBP-LORA, a parameter-efficient log anomaly detection framework based on BERT and low-rank adaptation (LoRA). Instead of fine-tuning the entire BERT model, LogBP-LORA inserts lightweight bypass matrices in self-attention layers and only updates them, reducing training cost to just 0.06% of full fine-tuning. Moreover, it replaces conventional word-level inputs with log event sequence representations, which preserve inter-log semantics and reduce sequence length. Besides, Qi et al. [399] explored an orthogonal prompt-based anomaly detection approach LogGPT based on ChatGPT. Rather than training a dedicated model, LogGPT constructs task-specific prompts and queries ChatGPT for anomaly decisions.

4.4.10 *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 4.4.11), so as to improve the quality of returned patches. 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 studies from three aspects, i.e., LLMs as feature extractors, fine-tuning-based APCA and zero-shot-based APCA. First, as early as 2020, Tian et al. [400, 401] investigated the effectiveness of representation learning for patch correctness assessment. They select four embedding models, including re-trained (i.e., Doc2vec, code2vec and CC2vec) and pre-trained models (i.e., BERT), marking the first application of BERT in this field. In 2022, Tian et al. [402] proposed Quatrain, which utilizes CodeTrans to generate patch descriptions, and BERT to embed bug reports and patch descriptions.

Recently, Invalidator [403] identifies the correctness of patches via semantic and syntactic reasoning. Second, APPT [404] is the first approach that fine-tunes LLMs to predict patch correctness. APPT consists of three components: (1) BERT to extract features from source code tokens; (2) LSTM to capture dependency information between source and repaired code snippets; and (3) a DL classifier to predict whether a patch is overfitting or not. Third, PatchZero [405] attempts to predict patch correctness in a zero-shot manner. PatchZero reformats the format of the patch correctness assessment task to match the original pre-training objective of the LLMs. Molina et al. [406] introduced FixCheck to filter out incorrect patches by combining random testing and LLMs to generate fault-revealing test cases.

4.4.11 *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 [407–410]. 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.

Given that program repair is one of the most intensively studied tasks in the LLM4SE field, with over 50 related studies, we adopt a multi-level structure to provide a clear overview of its diverse research landscape. We summarize the existing LLM-based program repair work involving LLMs into five stages, and for each stage, we introduce the background context and highlight representative techniques. 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 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. In the following, we list and summarize the existing LLM-based repair techniques based on these developmental stages.

Repair as a downstream task of LLMs. Since the inception of LLMs, some researchers have usually employed 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. The preliminary evaluations demonstrate the remarkable performance of LLMs in understanding code semantics and learning bug-fixing code changes, thus motivating further research efforts to integrate LLMs into the domain of automated program repair, and paving the way for more advanced investigations in this field.

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. [411] proposed a new NMT-based APR technique CURE empowered with GPT. First, CURE extracts millions of methods from open-source Java projects and uses 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 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. 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. [412] proposed 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 need 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 [413]. For example, DeepDebug [414] is an early-stage work that views program repair as a Seq2Seq learning task by fine-tuning BART on Java datasets. At the same time, Berabi et al. [415] presented TFix, which fine-tunes T5 to fix JavaScript code errors. To support multilingual repair, in 2022, Yuan et al. [23] proposed CIRCLE, a T5-based APR framework equipped with continual learning ability across multiple programming languages. Recently, some studies utilize parameter-efficient fine-tuning to reduce training costs [416, 417].

Zero-shot LLM-based repair. Despite promising, the repair performance of fine-tuned LLMs is usually con-

strained 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. [19] proposed AlphaRepair as the first cloze-style APR approach that leverages LLMs without any fine-tuning. AlphaRepair replaces the entire buggy line with a line containing only mask tokens and queries CodeBERT to generate candidate patches. Repilot [418] utilizes a Java completion engine to offer real-time feedback during the auto-regressive token generation process, thus assisting LLMs in producing better patches. FitRepair seeks to enhance the performance of cloze-style APR by combining LLMs and domain repair knowledge.

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, Zhang et al. [419] proposed Gamma to combine the advances of LLMs and well-known template-based repair. Gamma summarizes a variety of fix templates, transforms them into mask patterns, and adopts LLMs to predict the correct code for masked code as a fill-in-the-blank task. Ruiz et al. [420] explored the potential of LLMs to repair bugs with a round-trip translation strategy. At the same time, Peng et al. [421] proposed a domain-aware prompt-based approach TypeFix to repair Python type errors with fix templates.

Empirical evaluation of LLMs. In conjunction with the aforementioned repair strategies tackling certain technical obstacles, there has been a concurrent surge in empirical studies examining the development and nuances of these methodologies. These empirical studies systematically explore the actual performance of LLMs during the repair workflow, with the aim to furnish insights for forthcoming program repair endeavors [422, 423]. For example, in 2022, Xia et al. [14] conducted an extensive study on the application of LLMs in real-world bug-fixing, with nine LLMs of varying sizes. In 2023, Jiang et al. [424] evaluated ten code LLMs on four APR benchmarks, to discuss the impact of fine-tuning, model size, and repair costs. Unlike previous studies focusing on software bugs [14, 424], Fan et al. [425] conducted a systematic study to explore whether APR techniques can fix the incorrect solutions produced by language models in LeetCode contests. Besides, with the rise of ChatGPT, a mass of research efforts have been made to explore the potential of ChatGPT in repair scenarios, such as QuixBugs [426] and DL program [427].

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 repair techniques discussed above. Unlike traditional APR, LLMs are pre-trained on diverse datasets to acquire general language knowledge, allowing them to be applied across a wider range of repair scenarios. For example, Jin et al. [428] proposed InferFix, a Transformer-based approach based on Codex, designed to fix both critical security and performance bugs detected by the static analysis tool Infer. So far, researchers have utilized various strategies (e.g., zero-shot learning, fine-tuning, and few-shot learning) to incorporate LLMs into multiple repair domains, including program issues [429], static warning [428, 430], syntax error [431], benchmarks [432], neural network implementation [433], integrated development environment [434], human study [435], programming problems [436], repair agent [437, 438], interactive repair [439, 440], multi-location repair [441, 442], docker build faults [443], repair costs [444], and formal proof [445].

4.4.12 *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. CEPROT [446] represents the first attempt to fine-tune CodeT5 to automatically identify and update obsolete method-level test cases in response to changes in production code. SYNBCIATR [447] designs three target-oriented contexts (i.e., class contexts, usage contexts, and environment contexts), which are used to construct prompts to query GPT-4 to generate repaired test cases. TaRGet [448] treats the test update process as a language translation task by fine-tuning LLMs with crucial context information.

4.4.13 *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. 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. We summarize existing vulnerability repair studies empowered with LLMs as follows.

Fine-tuning LLM-based vulnerability repair. VulRepair [449] is an early-stage LLM-based vulnerability repair approach by directly fine-tuning CodeT5. VQM [450] is a successor of VulRepair by leveraging a cross-attention mechanism to locate vulnerable code elements during patch generation. VulMaster [451] fine-tunes CodeT5 with information from diverse sources, including the structure of vulnerable code and expert knowledge.

Zero-shot LLM-based vulnerability repair. In 2023, Pearce et al. [452] examined 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.

Empirical study of LLM-based vulnerability repair. In 2023, Zhang et al. [20] conducted the first extensive empirical study to investigate the actual performance of various LLMs on 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. Second, they delved into studying the impact of LLMs on the repair workflow, including data pre-processing, model training, and repair inference phases. 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. Besides, Wu et al. [22] compared the performance of LLMs with existing learning-based vulnerability repair techniques, including Codex, CodeGen, CodeT5, PLBART and InCoder. A similar study is conducted by Huang et al. [423]. In 2023, Tol et al. [453] empirically explored the potential of leveraging LLMs to automatically generate patches for side-channel vulnerabilities.

In addition to the aforementioned studies, researchers also leverage LLMs to automate a variety of software maintenance tasks, including bug triaging [454, 455], code smell [456], emotion-cause extraction [457], exception handling recommendation [458], incident management [459, 460], issue labeling [461], privacy policy [462], code porting [463], and Android permission-related problems resolution [464].

4.5 Software management

Software management refers to the practice of overseeing the entire software lifecycle, ensuring that software products are delivered on time, within budget, and meet quality standards. To date, researchers mainly utilize LLMs to perform effort estimation, software tool configuration, developers' behavior analysis, and software repository mining. First, effort estimation attempts to predict the amount of time, resources, and effort required to complete a software project. Alhamed et al. [465] explored the potential of BERT with expert features to estimate the effort required for software maintenance tasks. Fine-SE [466] combines both semantic and expert features to develop an automatic integrated BERT-based approach for effort estimation. Second, software tool configuration attempts to ensure that the appropriate tools are selected and configured. Kannan [467] discussed various prompting strategies for interacting with GPT-4 in configuring tools like machine learning frameworks and complex software systems. Third, analyzing developers' behavior is crucial for project managers to understand team dynamics, productivity, and collaboration patterns. Cai et al. [458] conducted the first attempt to utilize LLMs to detect the causes of emotions within developer communications. They focus on zero-shot LLMs, such as ChatGPT and GPT-4, to automatically recognize and extract the causes behind emotional expressions such as frustration, happiness, or anger in platforms such as GitHub and Stack Overflow. Imran [468] conducted an empirical study for emotion classification by fine-tuning LLMs, such as BERT, RoBERTa, CodeBERT and GraphCodeBERT. Fourth, software repository mining involves analyzing and extracting useful information from software repositories, thus helping project managers gain insights into development trends, team productivity, code quality, and project health. Abedu et al. [469] explored the performance of LLMs answering questions related to software repositories to lower the barrier for stakeholders by automating the extraction and analysis of repository data.

Answer to RQ2. Overall, LLMs have been employed across various stages of SE research, tackling a diverse array of 112 tasks. On the 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 and GUI testing; and (3) include some unique areas of SE, such as fuzzing. Researchers need to devote more effort to addressing these more domain-specific issues, such as designing specific LLMs or embedding them into existing research workflows. 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 & design and software management is still relatively unexplored, suggesting a potential area of focus for future research in this field.

5 RQ3: integration perspective

In the rapidly evolving field of SE, LLMs have emerged in pivotal roles, offering unprecedented opportunities for various code-related tasks. However, the integration 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. Thus, this section delves into four crucial aspects of integrating LLMs for SE, focusing on evaluation and benchmarking in Section 5.1, security and reliability in Section 5.2, domain tuning in Section 5.3, and compressing and distillation in Section 5.4. It is worth noting that this section focuses on the challenges encountered during the integration process and the efforts made by researchers to address these challenges, rather than specific integration strategies, which have been discussed in Section 4, such as few-shot learning and fine-tuning for program repair in Section 4.4.11.

5.1 Evaluation and benchmarking

Despite being 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 empirical studies and benchmarks.

5.1.1 *Empirical study*

In Section 4, when introducing specific SE tasks, we have discussed corresponding empirical studies, such as program repair [424] and vulnerability repair [20]. We also notice that there exist some empirical studies that systematically explore the capabilities of LLMs from a more comprehensive perspective, such as human studies and educational scenarios.

Exploration for multiple tasks. These studies explore LLMs across multiple LLMs or tasks, providing macroscopic insights into future LLM-based SE work. In 2022, Mastropaolo et al. [470] empirically evaluated the performance of transfer learning on four code-related tasks by pre-training and fine-tuning T5, including (1) automatic bug-fixing, (2) injection of code mutants, (3) generation of assert statements, and (4) code summarization. Lu et al. [73] conducted an empirical study to investigate the performance of three common LLMs in the CodeXGLUE benchmark (discussed in Section 5.1.2), including the BERT-style, GPT-style, and Encoder-Decoder models. As a pioneering work, this study only reports some preliminary results on a limited set of models. In 2022, Zeng et al. [471] conducted a comprehensive study of eight LLMs across seven code-related tasks in 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). In 2023, Niu et al. [472] performed a large systematic study of 19 LLMs on 13 SE tasks. Besides, researchers explore the performance of LLMs in SE tasks regarding different aspects, including in-context learning [473], prompt design [474], and code distributions [475].

Human study and empirical replication. Liang et al. [476] explored how LLMs can be utilized to support human participation tasks in empirical software engineering. This study involves four LLMs (including ChatGPT, ERNIE Bot, Gemini, and ChatGLM) and three types of prompts (including zero-shot, one-shot, and optimized one-shot prompts). Endres et al. [250] investigated the potential of GPT-4 to replicate empirical software engineering research. They query GPT-4 to generate assumptions, analysis plans, and code modules based on the methodologies described in seven papers and conduct a user study involving 14 participants to evaluate the quality of the outputs generated by GPT-4.

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 the 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.

Thus, researchers explore the potential and concerns of LLMs in the SE education scenario. For example, to explore ChatGPT's effectiveness in answering software testing questions from a textbook, in 2023, Jalil et al. [477] evaluated ChatGPT with 31 questions from five 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. At the same time, to explore how well ChatGPT can perform in an introductory-level functional language programming course, Geng et al. [478] selected a second-year undergraduate computer science course to test ChatGPT. When LLMs (e.g., ChatGPT) are used by students during their learning processes, despite their potential, some concerns may arise. 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. [479] presented 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 LLMs. Tian et al. [480] conducted an empirical analysis of ChatGPT's potential as a fully automated programming assistant. Xue et al. [481] investigated the performance of ChatGPT in assisting students in an introductory computer science course. This study is conducted in a classroom setting and includes both quantitative and qualitative analyses to understand the impact of using ChatGPT on students' learning outcomes and behaviors during programming assignments.

5.1.2 *Benchmarking*

Benchmarking plays a crucial role in LLM-based SE research, helping to drive the advancement of this rapidly evolving field. Existing benchmarks in the community can be broadly classified into two categories. The first category consists of existing datasets created by traditional SE research, which are generally well-established and have been validated by the community over time. Details can be found in prior SE survey papers [3, 482]. The second category includes newly developed datasets tailored specifically for LLM-based SE research. These new datasets primarily focus on two aspects: addressing challenges unique to LLMs and exploring various SE domains, summarized as follows.

First, different from traditional SE studies, the pipeline 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. As a result, such LLMs are usually trained from all possible open-source projects in the wild, and it is difficult to ensure that there are no samples in the evaluation benchmark that appear in the pre-training dataset, i.e., the data leakage problem. To address this issue, researchers conduct new benchmarks from artifacts that are released after the training cutoff date of ChatGPT. For example, Zhang et al. [483] extensively explored the data leakage issue of ChatGPT in the program domain and introduced EvalGPTFix, a new benchmark based on competitive programming problems collected after 2021. Several other benchmarks includes HumanEval-Java [424], ConDefects [484] and TestBench [485].

Second, some benchmarks are tailored for different SE scenarios. HumanEval [74] is initially released by OpenAI to evaluate the code generation capabilities of Codex and has been the most popular benchmark in LLM-based code generation. HumanEval-X [83] is an extended version of HumanEval, specifically designed to evaluate the multi-lingual code generation capabilities of LLMs. Unlike HumanEval only for Python code generation, HumanEval-X supports both code generation and code translation tasks, spanning Python, C++, Java, JavaScript, and Go. EvalPlus [486] enhances HumanEval by generating additional high-quality test inputs. CodeXGlUE [73] is constructed by Microsoft to support a collection of ten code understanding and generation tasks and six programming languages. Compared with CodeXGLUE, CrossCodeBench [487] provides a larger scale and more diverse code-related tasks, including 216 tasks and more than 54M data instances. Recently, LLMs have been applied in some SE scenarios that have not been previously investigated, resulting in a gap in relevant benchmarks. As a result, with the advent of LLM-based SE techniques, researchers have also developed corresponding new datasets, such as class-level code generation [488], project-level code generation [185], code decompilation [255], and code

retrieval [489].

5.2 Security and reliability

As LLMs have been widely utilized and achieved state-of-the-art performances in various code-related tasks, the security and robustness of these models deserve an increasing amount of attention [490]. Similar to traditional DL models, LLMs have been shown to be vulnerable to adversarial attacks [491], 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. These studies focus on improving the applicability and reliability of LLMs specifically for software engineering tasks [492], rather than aiming to improve LLMs as general-purpose models, therefore falling within the LLM4SE. For details, see the survey by Chen et al. [493, 494].

In the following, we summarize existing studies on attacking LLMs for SE, which mainly fall into four 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. [491] proposed 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. Unlike ALERT focuses on code understanding tasks, like vulnerability detection and clone detection, Jha et al. [495] proposed 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 normally performs 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 leading 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 as 2021, Schuster et al. [496] performed a backdoor attack against the code completion model, including GPT-2. In 2022, Wan et al. [497] performed 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.

However, the previously designed triggers (i.e., dead code snippets) are very suspicious and can be easily identified by developers [497]. Thus, focusing on a more stealthy attack, in 2023, Sun et al. [498] proposed 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. Recently, AFRAIDOOR [499] leverages adversarial perturbations to inject adaptive triggers into code LLMs. Unlike previous studies designed for specific tasks, Li et al. [500] proposed 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. [501] proposed 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 on two LLMs (i.e., PLBART and CodeT5) and two code understanding

tasks (i.e., defect detection, clone detection) and three code generation tasks (i.e., code translation, code refinement, and code 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. [502] proposed the first imitation attack work to explore the feasibility of extracting specialized code abilities from LLMs using 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.

Others. In addition to the studies detailed above, researchers explore the security and reliability of LLMs from a mass of perspectives, including human study [503–506], glitch token analysis [507], testing [508–510], memorization detection [511], auto-generated code quality assurance [512–514], auto-generated code detection [515], knowledge understanding [516], model analysis [517], and jailbreak vulnerability fuzzing [518].

5.3 Domain 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 for targeted tasks. While fine-tuning LLMs has proven 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 [519], 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 [23], 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. It is worth noting that domain tuning refers to adapting LLMs to SE-specific codebases and tasks, aiming to improve their performance in concrete SE applications rather than general natural language tasks. In the following, we summarize existing studies on fine-tuning LLMs for SE, which mainly fall into three 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 [210, 520–523]. For example, Liu et al. [522] conducted an empirical study to explore the performance of parameter-efficient fine-tuning methods on two LLMs and four code-related tasks, including adapter tuning, prefix tuning and low-rank adaptation. In 2023, considering that fine-tuning LLMs incurs a high computational cost, Shi et al. [520] attempted to reduce the number of parameters requiring updates in Code LLMs by selectively freezing layers that capture fundamental code properties, while fine-tuning only the higher layers that are more sensitive to dynamic changes. Wang et al. [210] utilized adapter tuning to improve performance in code search and summarization tasks across multiple programming languages.

Effective continual fine-tuning. Such studies involve effective training strategies to address the catastrophic forgetting issue during fine-tuning [23, 524, 525]. For example, as early as 2022, Yuan et al. [23] proposed 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. [524] introduced 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. Besides, Weyssow et al. [525] investigated how to adapt PLMs to the dynamic nature of software development with continual learning, including replay-based and regularization-based strategies.

Others. In addition to the studies mentioned above, researchers utilize various tuning strategies to adapt LLMs for the SE domain, including noise-tolerant training [526], instruction tuning [336, 527], reinforcement learning [155, 528], and prompt learning [529].

5.4 Compression and distillation

Once LLMs are well-trained and achieve impressive results in various SE tasks, they need to be further deployed in real-world scenarios, such as integrated development environments. 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 these challenges, some optimization strategies have been utilized by SE researchers to enhance the usability and practicality of LLMs during deployment. For example, Shi et al. [530] proposed Compressor to compress LLMs into extremely small models without compromising performance. Compressor utilizes a genetic algorithm-based strategy to guide the process of model simplification and adopts knowledge distillation to obtain a well-performing small model. Compressor compresses two well-known LLMs (i.e., CodeBERT and GraphCodeBERT) to a size of 3 MB on two important tasks (i.e., vulnerability prediction and clone detection). Besides, Su et al. [531] introduced a smaller, distilled model with outputs generated by GPT-3.5 to generate concise natural language descriptions of source code.

Answer to RQ3. 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 adaptation. First, benchmarks play a pivotal role in shaping the trajectory of research advancements. One typical trend is the construction of new benchmarks to address the data leakage issue, e.g., HumanEval and EvalGPTFix. The other typical trend is the application of various SE 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, including multiple tasks, human study and SE education. Third, 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. Fourth, considering the huge parameter scale of LLMs, it is crucial to design domain-tuning strategies to adapt such LLMs in SE tasks, such as parameter and continual fine-tuning. Fifth, after LLMs are well-trained, deploying such LLMs within the development workflow necessitates further consideration of factors such as inference time and resource expenditure.

6 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.

6.1 LLMs deployment

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 [30] and program repair [14], 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, 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 63 GB of disk space[5)]. 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 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 [530]. 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

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

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.

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 a minor focus is 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 approaches with the predictive power of LLMs could help in bridging the gap between traditional SE approaches and advanced LLMs, providing a balance between transparency and performance.

6.2 Domain generalization

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 [156], fault detection [532], program repair [419], 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.

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 [533]. Recently, with advancements in computer vision technologies, vision-based GUI testing approaches have been developed and have shown promising progress.

We suggest that future work explore the use of multi-modal LLMs in version-based SE tasks. For instance, combining the text and image understanding capabilities of multi-modal LLMs could enhance the capture of syntactic and semantic information from source code, test scripts, and test reports in GUI testing. This integration may improve tasks such as GUI test generation, recording, and replaying.

Integrating LLMs into open-source ecosystems. The rapid development of LLMs has profoundly reshaped the open-source software ecosystem. On one hand, as mentioned in Section 3.4, open-source platforms such as GitHub, GitLab, and HuggingFace have provided large-scale, high-quality datasets that support the training and

evaluation of LLMs, driving their rapid evolution. On the other hand, the emergence of LLMs is transforming the paradigm of open-source platforms itself. As a result, developers and intelligent agents are beginning to collaborate within the open-source development lifecycle, including code generation, issue triage, issue resolution, documentation authoring, pull request submission, code review, and commit management. This emerging form of human-AI co-creation marks a fundamental shift from traditional human-driven collaboration to AI-augmented development, enabling open-source projects to evolve more efficiently and intelligently.

Future research should focus on exploring how LLMs can serve as the backbone of next-generation open-source ecosystems. In particular, two directions are especially promising. First, researchers should investigate LLM-driven human-AI collaboration paradigms, including multi-agent cooperation, adaptive task allocation, and continuous learning mechanisms, to enable more efficient and trustworthy human-machine synergy in open-source environments. Second, future studies should explore end-to-end intelligent code development strategies for open-source scenarios, leveraging LLMs to integrate requirement analysis, code generation, testing, debugging, and maintenance into a cohesive, closed-loop workflow.

6.3 LLMs evaluation

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, as such LLMS are typically trained on all publicly available repositories in the wild. As mentioned in Section 5.1.2, researchers [419] 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 challenging to determine whether the evaluation dataset has been exposed to such LLMs during training, as these LLMs are typically closed-source and their specific training details, such as pre-training datasets, are unknown. 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 solutions by simply being given the number of a programming problem in LeetCode. Considering the fact that there exists a considerable number of black-box LLMs for which no architecture or training data information has been released. The data leakage on such LLMs is a significant concern when evaluating their performance in code-related tasks within the SE community.

We recommend that future work 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) manually written programs, where researchers can create evaluation programs by hand to provide a unique and uncontaminated benchmark, such as HumanEval [74]; (2) recently released programs, such as EvalGPTFix [483], where researchers can seek out the latest programs from recent competitions or coding challenge websites, which often contain fresh and diverse problems that are less likely to have been included in the training sets of current LLMs; and (3) closed-source projects, where researchers can evaluate LLMs with some internal projects within companies, which are not previously exposed to public repositories, thereby providing a more authentic evaluation of LLMs' 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.

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 focus on 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 lack of evaluation dimensions, new metrics and specialized benchmarks should be introduced to assess some crucial aspects, such as robustness and efficiency.

7 Conclusion

LLMs are bringing significant changes to the 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 62 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 926 relevant studies for 112 code-related tasks across five crucial SE phases. We highlight several crucial aspects of the optimization and application for the LLM-based SE research, including empirical evaluation, security and reliability, domain tuning, and compression and distillation. 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.

Acknowledgements This work was partially supported by National Key Research and Development Program of China (Grant No. 2024YFF0-908005), National Natural Science Foundation of China (Grant Nos. U24A20337, 62372228), Fundamental Research Funds for the Central Universities (Grant No. 14380029), and Shenzhen-Hong Kong-Macau Technology Research Program (Type C) (Grant No. SGDX20230821091559018). We would like to thank anonymous reviewers for their time and comments.

Supporting information Appendixes A and B. The supporting information is available online at info.scichina.com and link.springer.com. The supporting materials are published as submitted, without typesetting or editing. The responsibility for scientific accuracy and content remains entirely with the authors.

References

1 Biolchini J, Mian P G, Natali A C C, et al. Systematic review in software engineering. System engineering and computer science department COPPE/UFRJ, Technical Report ES, 2005, 679: 45
2 Zelkowitz M V. Perspectives in software engineering. ACM Comput Surv, 1978, 10: 197–216
3 Yang Y, Xia X, Lo D, et al. A survey on deep learning for software engineering. ACM Comput Surv, 2022, 54: 1–73
4 Wang J, Huang Y, Chen C, et al. Software testing with large language model: survey, landscape, and vision. ArXiv:230707221
5 Devlin J, Chang M W, Lee K, et al. Bert: pre-training of deep bidirectional transformers for language understanding. ArXiv:181004805
6 Raffel C, Shazeer N, Roberts A, et al. Exploring the limits of transfer learning with a unified text-to-text transformer. J Mach Learn Res, 2020. 21: 5485–5551
7 Radford A, Wu J, Child R, et al. Language models are unsupervised multitask learners. OpenAI blog, 2019, 1: 9
8 Zhang Y, Wu Y, Chen T, et al. How do developers talk about github actions? Evidence from online software development community. In: Proceedings of the 46th International Conference on Software Engineering (ICSE), 2024. 492–504
9 Dong J, Sun J, Zhang W, et al. ConTested: consistency-aided tested code generation with LLM. Proc ACM Softw Eng, 2025, 2: 596–617
10 Fu Y, Li B, Li L, et al. The first prompt counts the most! An evaluation of large language models on iterative example-based code generation. Proc ACM Softw Eng, 2025, 2: 1583–1606
11 Yu H, Shen B, Zhang J, et al. Wenwang: toward effectively generating code beyond standalone functions via generative pre-trained models. ACM Trans Softw Eng Methodol, 2025, 34: 187
12 OpenAI. ChatGPT: optimizing language models for dialogue. 2023. https://openai.com/blog/chatgpt
13 Zhang Q, Fang C, Xie Y, et al. A systematic literature review on large language models for automated program repair. ArXiv:240501466
14 Xia C S, Wei Y, Zhang L. Automated program repair in the era of large pre-trained language models. In: Proceedings of the 45th International Conference on Software Engineering (ICSE), 2023. 1482–1494
15 Hu H, He C, Zhang H, et al. Aprmcts: improving LLM-based automated program repair with iterative tree search. ArXiv:250701827
16 Shang Y, Zhang Q, Fang C, et al. A large-scale empirical study on fine-tuning large language models for unit testing. Proc ACM Softw Eng, 2025, 2: 1678–1700
17 Zhang Q, Sun W, Fang C, et al. Exploring automated assertion generation via large language models. ACM Trans Softw Eng Methodol, 2025. 34: 1–25
18 Zhang Q, Fang C, Zheng Y, et al. Improving retrieval-augmented deep assertion generation via joint training. IEEE Trans Softw Eng, 2025, 51: 1232–1247
19 Xia C S, Zhang L. 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. 959–971
20 Zhang Q, Fang C, Yu B, et al. Pre-trained model-based automated software vulnerability repair: How far are we? IEEE Trans Depend Secure Comput, 2023, 21: 2507–2525
21 Fried D, Aghajanyan A, Lin J, et al. Incoder: a generative model for code infilling and synthesis. ArXiv:220405999
22 Wu Y, Jiang N, Pham H V, et al. How effective are neural networks for fixing security vulnerabilities. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2023. 1282–1294
23 Yuan W, Zhang Q, He T, et al. Circle: continual repair across programming languages. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, 2022. 678–690
24 Hu H, Xie X, Zhang Q. Repair-r1: better test before repair. ArXiv:250722853
25 Nashid N, Sintaha M, Mesbah A. Retrieval-based prompt selection for code-related few-shot learning. In: Proceedings of the 45th International Conference on Software Engineering (ICSE), 2023. 2450–2462
26 Watson C, Cooper N, Palacio D N, et al. A systematic literature review on the use of deep learning in software engineering research. ACM Trans Softw Eng Methodol, 2022, 31: 1–58

27 Wang S, Huang L, Gao A, et al. Machine/deep learning for software engineering: a systematic literature review. IEEE Trans Software Eng, 2022, 49: 1188–1231
28 Zhang H Z, Zhang K C, Li Z, et al. Deep learning for code generation: a survey. Sci China Inf Sci, 2024, 67: 191101
29 Niu C, Li C, Luo B, et al. Deep learning meets software engineering: a survey on pre-trained models of source code. In: Proceedings of the 31st International Joint Conference on Artificial Intelligence, 2022. 5546–5555
30 Zan D, Chen B, Zhang F, et al. Large language models meet nl2code: a survey. In: Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics, 2023. 7443–7464
31 Chen X P, Hu X, Huang Y, et al. Deep learning-based software engineering: progress, challenges, and opportunities. Sci China Inf Sci, 2025, 68: 111102
32 Fan A, Gokkaya B, Harman M, et al. Large language models for software engineering: survey and open problems. ArXiv:231003533
33 Hou X, Zhao Y, Liu Y, et al. Large language models for software engineering: a systematic literature review. ArXiv:230810620
34 Zheng Z, Ning K, Zhong Q, et al. Towards an understanding of large language models in software engineering tasks. Empir Softw Eng, 2025, 30: 50
35 Zheng Z, Ning K, Wang Y, et al. A survey of large language models for code: evolution, benchmarking, and future trends. ArXiv:231110372
36 Ran D, Wu M, Yang W, et al. Foundation model engineering: engineering foundation models just as engineering software. ACM Trans Softw Eng Methodol, 2025, 34: 1–18
37 Liu J, Wang K, Chen Y, et al. Large language model-based agents for software engineering: a survey. ArXiv:240902977
38 Zhang Q, Fang C, Gu S, et al. Large language models for unit testing: a systematic literature review. ArXiv:250615227
39 Zhou X, Cao S, Sun X, et al. Large language model for vulnerability detection and repair: literature review and the road ahead. ACM Trans Softw Eng Methodol, 2025, 34: 145
40 Sheng Z, Chen Z, Gu S, et al. LLMs in software security: a survey of vulnerability detection techniques and insights. ArXiv:2502.07049
41 Jiang J, Wang F, Shen J, et al. A survey on large language models for code generation. ArXiv:240600515
42 Gao C, Hu X, Gao S, et al. The current challenges of software engineering in the era of large language models. ACM Trans Softw Eng Methodol, 2025, 34: 127
43 He J, Treude C, Lo D. LLM-based multi-agent systems for software engineering: literature review, vision and the road ahead. ACM Trans Softw Eng Methodol, 2024, 34: 124
44 Zhang H, Babar M A, Tell P. Identifying relevant studies in software engineering. Inf Softw Tech, 2011, 53: 625–637
45 Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need. In: Proceedings of Advances in Neural Information Processing Systems, 2017
46 Wang Z, Yan M, Chen J, et al. 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. 788–799
47 Zhang Q, Fang C, Ma Y, et al. A survey of learning-based automated program repair. ACM Trans Softw Eng Methodol, 2023. 33: 1–69
48 Kanade A, Maniatis P, Balakrishnan G, et al. Learning and evaluating contextual embedding of source code. In: Proceedings of International Conference on Machine Learning, 2020. 5110–5121
49 Feng Z, Guo D, Tang D, et al. Codebert: a pre-trained model for programming and natural languages. In: Proceedings of Findings of the Association for Computational Linguistics, 2020. 1536–1547
50 Guo D, Ren S, Lu S, et al. Graphcodebert: pre-training code representations with data flow. ArXiv:200908366
51 Mukherjee M, Hellendoorn V J. Stack over-flowing with results: the case for domain-specific pre-training over one-size-fits-all models. ArXiv:230603268
52 Zhang D, Ahmad W, Tan M, et al. Code representation learning at scale. ArXiv:240201935
53 Lin J, Dong H, Xie Y, et al. Scaling laws behind code understanding model. ArXiv:240212813
54 Clement C B, Drain D, Timcheck J, et al. Pymt5: multi-mode translation of natural language and Python code with transformers. ArXiv:201003150
55 Mastropaolo A, Scalabrino S, Cooper N, et al. Studying the usage of text-to-text transfer transformer to support code-related tasks. In: Proceedings of the 43rd International Conference on Software Engineering (ICSE), 2021. 336–347
56 Ahmad W U, Chakraborty S, Ray B, et al. Unified pre-training for program understanding and generation. ArXiv:210306333
57 Wang Y, Wang W, Joty S, et al. Codet5: identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP'21), 2021. 8696–8708
58 Guo D, Lu S, Duan N, et al. Unixcoder: unified cross-modal pre-training for code representation. In: Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics, 2022. 7212–7225
59 Niu C, Li C, Ng V, et al. Spt-code: sequence-to-sequence pre-training for learning source code representations. In: Proceedings of the 44th International Conference on Software Engineering, 2022. 2006–2018
60 Le H, Wang Y, Gotmare A D, et al. Coderl: mastering code generation through pretrained models and deep reinforcement learning. In: Proceedings of Advances in Neural Information Processing Systems, 2022. 21314–21328
61 Zhang J, Panthaplackel S, Nie P, et al. Coditt5: pretraining for source code and natural language editing. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022. 1–12
62 Li Y, Choi D, Chung J, et al. Competition-level code generation with AlphaCode. Science, 2022, 378: 1092–1097
63 Wang Y, Le H, Gotmare A D, et al. Codet5+: open code large language models for code understanding and generation. ArXiv:230507922
64 Chandel S, Clement C B, Serrato G, et al. Training and evaluating a jupyter notebook data science assistant. ArXiv:220112901
65 Chai Y, Wang S, Pang C, et al. Ernie-code: beyond English-centric cross-lingual pretraining for programming languages. ArXiv:221206742
66 Shojaee P, Jain A, Tipirneni S, et al. Execution-based code generation using deep reinforcement learning. ArXiv:230113816
67 Liu J, Zhu Y, Xiao K, et al. Rltf: reinforcement learning from unit test feedback. ArXiv:230704349
68 Lin B, Wang S, Liu Z, et al. Cct5: a code-change-oriented pre-trained model. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2023. 1509–1521
69 Yu Z, Tao Y, Chen L, et al. B-coder: value-based deep reinforcement learning for program synthesis. ArXiv:2310.03173
70 Gong L, Elhoushi M, and Cheung A. Ast-t5: structure-aware pretraining for code generation and understanding. ArXiv:240103003
71 Zhu Q, Liang Q, Sun Z, et al. Grammart5: grammar-integrated pretrained encoder-decoder neural model for code. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024. 1–13
72 Svyatkovskiy A, Deng S K, Fu S, et al. 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. 1433–1443
73 Lu S, Guo D, Ren S, et al. Codexglue: a machine learning benchmark dataset for code understanding and generation. In: Proceedings of the 35th Conference on Neural Information Processing Systems, 2021. 1–14
74 Chen M, Tworek J, Jun H, et al. Evaluating large language models trained on code. ArXiv:210703374
75 Xu F F, Alon U, Neubig G, et al. A systematic evaluation of large language models of code. In: Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming, 2022. 1–10
76 Nijkamp E, Pang B, Hayashi H, et al. Codegen: an open large language model for code with multi-turn program synthesis. ArXiv:220313474
77 Zan D, Chen B, Yang D, et al. Cert: continual pre-training on sketches for library-oriented code generation. ArXiv:220606888
78 Allal L B, Li R, Kocetkov D, et al. Santacoder: don't reach for the stars! ArXiv:230103988
79 Li R, Allal L B, Zi Y, et al. Starcoder: may the source be with you! ArXiv:230506161
80 Christopoulou F, Lampouras G, Gritta M, et al. Pangu-coder: program synthesis with function-level language modeling. ArXiv:220711280

81 Shen B, Zhang J, Chen T, et al. Pangu-coder2: boosting large language models for code with ranking feedback. ArXiv:230714936
82 Chowdhery A, Narang S, Devlin J, et al. Palm: scaling language modeling with pathways. ArXiv:220402311
83 Zheng Q, Xia X, Zou X, et al. Codegeex: a pre-trained model for code generation with multilingual evaluations on Humaneval-X. ArXiv:230317568
84 Nijkamp E, Hayashi H, Xiong C, et al. Codegen2: lessons for training LLMs on programming and natural languages. ArXiv:230502309
85 Touvron H, Lavril T, Izacard G, et al. Llama: open and efficient foundation language models. ArXiv:230213971
86 Workshop B, Scao T L, Fan A, et al. Bloom: a 176b-parameter open-access multilingual language model. ArXiv:221105100
87 Di P, Li J, Yu H, et al. Codefuse-13b: a pretrained multi-lingual code large language model. In: Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice, 2024. 418–429
88 Xie R, Zeng Z, Yu Z, et al. Codeshell technical report. ArXiv:240315747
89 Xu Y, Su H, Xing C, et al. Lemur: harmonizing natural language and code for language agents. ArXiv:231006830
90 Wei Y, Wang Z, Liu J, et al. Magicoder: empowering code generation with OSS-instruct. In: Proceedings of the 41st International Conference on Machine Learning, 2024. 52632–52657
91 Team O. Octocoder. 2023 https://huggingface.co/bigcode/octocoder
92 Muennighoff N, Liu Q, Zebaze A, et al. Octopack: instruction tuning code large language models. ArXiv:230807124
93 Luo Z, Xu C, Zhao P, et al. Wizardcoder: empowering code large language models with evol-instruct. ArXiv:230608568
94 Song Z, Wang Y, Zhang W, et al. Alchemistcoder: harmonizing and eliciting code capability by hindsight tuning on multi-source data. ArXiv:240519265
95 Lei B, Li Y, Chen Q. Autocoder: enhancing code large language model with AIEV-Instruct. ArXiv:240514906
96 Team C. Codegemma: open code models based on Gemma. ArXiv:240611409
97 Bai J, Bai S, Chu Y, et al. Qwen technical report. ArXiv:230916609
98 Guo D, Zhu Q, Yang D, et al. Deepseek-coder: when the large language model meets programming—the rise of code intelligence. ArXiv:240114196
99 Zhu Q, Guo D, Shao Z, et al. Deepseek-coder-v2: breaking the barrier of closed-source models in code intelligence. ArXiv:240611931
100 Wang Y, He K, Dong G, et al. Dolphcoder: echo-locating code large language models with diverse and multi-objective instruction tuning. ArXiv:240209136
101 Mishra M, Stallone M, Zhang G, et al. Granite code models: a family of open foundation models for code intelligence. ArXiv:240504324
102 Ding Y, Liu J, Wei Y, et al. Xft: unlocking the power of code instruction tuning by simply merging upcycled mixture-of-experts. ArXiv:240415247
103 Wu Y, Huang D, Shi W, et al. Inversecoder: unleashing the power of instruction-tuned code LLMs with inverse-instruct. ArXiv:240705700
104 Rathinasamy K, Kumar A, Gayari G, et al. Narrow transformer: starcoder-based Java-LM for desktop. ArXiv:240703941
105 Lozhkov A, Li R, Allal L B, et al. Starcoder 2 and the stack v2: the next generation. ArXiv:240219173
106 Dou S, Liu Y, Jia H, et al. Stepcoder: improving code generation with reinforcement learning from compiler feedback. In: Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics, 2024. 4571–4585
107 Sun T, Chai L, Yang J, et al. Unicoder: scaling code large language model via universal code. ArXiv:240616441
108 Yu Z, Zhang X, Shang N, et al. Wavecoder: widespread and versatile enhancement for code large language models by instruction tuning. In: Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics, 2024. 5140–5153
109 Zhao W X, Zhou K, Li J, et al. A survey of large language models. ArXiv:230318223
110 Yin S, Fu C, Zhao S, et al. A survey on multimodal large language models. ArXiv:230613549
111 Roziere B, Gehring J, Gloeckle F, et al. Code llama: open foundation models for code. ArXiv:230812950
112 Clark K, Luong M T, Le Q V, et al. Electra: pre-training text encoders as discriminators rather than generators. ArXiv:200310555
113 Nong Y, Sharma R, Hamou-Lhadj A, et al. Open science in software engineering: a study on deep learning-based vulnerability detection. IEEE Trans Softw Eng, 2022, 49: 1983–2005
114 Moharil A, Sharma A. TABASCO: a transformer based contextualization toolkit. Sci Comput Programm, 2023, 230: 102994
115 Moharil A, Sharma A. Identification of intra-domain ambiguity using transformer-based machine learning. In: Proceedings of the 1st International Workshop on Natural Language-based Software Engineering, 2022. 51–58
116 Ezzini S, Abualhaija S, Arora C, et al. Automated handling of anaphoric ambiguity in requirements: a multi-solution study. In: Proceedings of the 44th International Conference on Software Engineering, 2022. 187–199
117 Sridhara G, Ranjani H G, Mazumdar S. ChatGPT: a study on its utility for ubiquitous software engineering tasks. ArXiv:230516837
118 Kolthoff K, Bartelt C, Ponzetto S P. Data-driven prototyping via natural-language-based GUI retrieval. Autom Softw Eng, 2023, 30: 13
119 Brie P, Burny N, Sluÿters A, et al. Evaluating a large language model on searching for GUI layouts. Proc ACM Hum-Comput Interact, 2023, 7: 1–37
120 Hey T, Keim J, Koziolek A, et al. Norbert: transfer learning for requirements classification. In: Proceedings of the 28th International Requirements Engineering Conference (RE), 2020. 169–179
121 Khan M A, Khan M S, Khan I, et al. Non functional requirements identification and classification using transfer learning model. IEEE Access, 2023, 11: 74997–75005
122 Rahman K, Ghani A, Alzahrani A, et al. Pre-trained model-based NFR classification: overcoming limited data challenges. IEEE Access, 2023, 11: 81787–81802
123 Han L, Zhou Q, Li T. Improving requirements classification models based on explainable requirements concerns. In: Proceedings of the 31st International Requirements Engineering Conference Workshops (REW), 2023. 95–101
124 Lubos S, Felfernig A, Tran T N T, et al. Leveraging LLMs for the quality assurance of software requirements. In: Proceedings of the 32nd International Requirements Engineering Conference (RE), 2024. 389–397
125 Preda A R, Mayr-Dorn C, Mashkoor A, et al. Supporting high-level to low-level requirements coverage reviewing with large language models. In: Proceedings of the 21st International Conference on Mining Software Repositories, 2024. 242–253
126 Poudel A, Lin J, Cleland-Huang J. Leveraging transformer-based language models to automate requirements satisfaction assessment. ArXiv:231204463
127 Ronanki K, Cabrero-Daniel B, Berger C. ChatGPT as a tool for user story quality evaluation: trustworthy out of the box? In: Proceedings of International Conference on Agile Software Development, 2022. 173–181
128 Ferrari A, Abualhaija S, Arora C. Model generation from requirements with LLMs: an exploratory study. ArXiv:240406371
129 Wang B, Wang C, Liang P, et al. How LLMs aid in UML modeling: an exploratory study with novice analysts. ArXiv:240417739
130 Tinnes C, Welter A, Apel S. Leveraging large language models for software model completion: results from industrial and public datasets. ArXiv:240617651
131 Xie D, Yoo B, Jiang N, et al. Impact of large language models on generating software specifications. ArXiv:230603324
132 Ma L, Liu S, Li Y, et al. Specgen: automated generation of formal program specifications via large language models. ArXiv:240108807
133 Mandal S, Chethan A, Janfaza V, et al. Large language models based automatic synthesis of software specifications. ArXiv:230409181
134 Hasan M R, Li J, Ahmed I, et al. Automated repair of declarative software specifications in the era of large language models. ArXiv:231012425
135 Li Z, Dutta S, Naik M. LLM-assisted static analysis for detecting security vulnerabilities. ArXiv:240517238
136 Luitel D, Hassani S, Sabetzadeh M. Improving requirements completeness: automated assistance through large language models. Requir Eng, 2024, 29: 73–95
137 Ren S, Nakagawa H, Tsuchiya T. Combining prompts with examples to enhance LLM-based requirement elicitation. In: Proceedings of

the 48th Annual Computers, Software, and Applications Conference (COMPSAC), 2024. 1376–1381
138 Sami M A, Rasheed Z, Waseem M, et al. Prioritizing software requirements using large language models. ArXiv:240501564
139 Jain C, Anish P R, Singh A, et al. A transformer-based approach for abstractive summarization of requirements from obligations in software engineering contracts. In: Proceedings of the 31st International Requirements Engineering Conference (RE), 2023. 169–179
140 Guo J L, Steghöfer J P, Vogelsang A, et al. Natural language processing for requirements traceability. ArXiv:240510845
141 Lin J, Liu Y, Zeng Q, et al. Traceability transformed: generating more accurate links with pre-trained BERT models. In: Proceedings of the 43rd International Conference on Software Engineering (ICSE), 2021. 324–335
142 Zhang S, Wang J, Dong G, et al. Experimenting a new programming practice with LLMs. ArXiv:240101062
143 Wei M, Harzevili N S, Huang Y, et al. Clear: contrastive learning for API recommendation. In: Proceedings of the 44th International Conference on Software Engineering, 2022. 376–387
144 Li Z, Li C, Tang Z, et al. PTM-apirec: leveraging pre-trained models of source code in api recommendation. ACM Trans Softw Eng Methodol, 2024, 33: 1–30
145 Huang Q, Wan Z, Xing Z, et al. Let's chat to find the APIs: connecting human, LLM and knowledge graph through AI chain. In: Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2023. 471–483
146 Chen Y, Gao C, Zhu M, et al. Apigen: generative API method recommendation. ArXiv:240115843
147 Mastropaolo A, Aghajani E, Pascarella L, et al. An empirical study on code comment completion. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME), 2021. 159–170
148 Nie P, Banerjee R, Li J J, et al. Learning deep semantics for test completion. In: Proceedings of the 45th International Conference on Software Engineering (ICSE), 2023. 2111–2123
149 Li Z, Wang C, Liu Z, et al. Cctest: testing and repairing code completion systems. In: Proceedings of the 45th International Conference on Software Engineering (ICSE), 2023. 1238–1250
150 Liu W, Yu A, Zan D, et al. Graphcoder: enhancing repository-level code completion via code context graph-based retrieval and language model. ArXiv:240607003
151 Cheng W, Wu Y, Hu W. Dataflow-guided retrieval augmentation for repository-level code completion. ArXiv:240519782
152 Wu D, Ahmad W U, Zhang D, et al. Repoformer: selective retrieval for repository-level code completion. ArXiv:240310059
153 Phan H N, Phan H N, Nguyen T N, et al. Repohyper: better context retrieval is all you need for repository-level code completion. ArXiv:240306095
154 Liu J, Chen Y, Liu M, et al. Stall+: boosting LLM-based repository-level code completion with static analysis. ArXiv:240610018
155 Wang Y, Wang Y, Guo D, et al. Rlcoder: reinforcement learning for repository-level code completion. ArXiv:240719487
156 Ciniselli M, Cooper N, Pascarella L, et al. An empirical study on the usage of BERT models for code completion. In: Proceedings of the 18th International Conference on Mining Software Repositories (MSR), 2021. 108–119
157 Ciniselli M, Cooper N, Pascarella L, et al. An empirical study on the usage of transformer models for code completion. IEEE Trans Softw Eng, 2021, 48: 4818–4837
158 van Dam T, Izadi M, van Deursen A. Enriching source code with contextual data for code completion models: an empirical study. ArXiv:230412269
159 Liu C, Cai Y, Lin Y, et al. Coedpilot: recommending code edits with learned prior edit relevance, project-wise awareness, and interactive nature. In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2024. 466–478
160 Li J, Li G, Li Z, et al. Codeeditor: learning to edit source code with pre-trained models. ACM Trans Softw Eng Methodol, 2023, 32: 1–22
161 Gupta P, Khare A, Bajpai Y, et al. Grace: generation using associated code edits. ArXiv:230514129
162 Liu C, Cetin P, Patodia Y, et al. Automated code editing with search-generate-modify. In: Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, 2024. 398–399
163 Yang G, Zhou Y, Chen X, et al. A syntax-guided multi-task learning approach for Turducken-style code generation. Empir Softw Eng, 2023, 28: 141
164 Jiang X, Dong Y, Wang L, et al. Self-planning code generation with large language models. ACM Trans Softw Eng Methodol, 2024, 33: 1–30
165 Zhang S, Chen Z, Shen Y, et al. Planning with large language models for code generation. ArXiv:230305510
166 Zhang Q, Fang C, Shang Y, et al. No man is an island: towards fully automatic programming by code search, code generation and program repair. ArXiv:240903267
167 Han H, Kim J, Yoo J, et al. Archcode: incorporating software requirements in code generation with large language models. ArXiv:240800994
168 Mu F, Shi L, Wang S, et al. ClarifyGPT: empowering LLM-based code generation with intention clarification. ArXiv:231010996
169 Li J, Zhao Y, Li Y, et al. Acecoder: an effective prompting technique specialized in code generation. ACM Trans Softw Eng Methodol, 2024, 33: 204
170 Ni A, Iyer S, Radev D, et al. Lever: learning to verify language-to-code generation with execution. In: Proceedings of International Conference on Machine Learning, 2023. 26106–26128
171 Chen X, Lin M, Schärli N, et al. Teaching large language models to self-debug. ArXiv:230405128
172 Chen A, Scheurer J, Korbak T, et al. Improving code generation by training with natural language feedback. ArXiv:230316749
173 Zhang K, Li Z, Li J, et al. Self-edit: fault-aware code editor for code generation. ArXiv:230504087
174 Dong Y, Jiang X, Jin Z, et al. Self-collaboration code generation via ChatGPT. ACM Trans Softw Eng Methodol, 2024. doi: 10.1145/3672459
175 Mastropaolo A, Pascarella L, Guglielmi E, et al. On the robustness of code generation techniques: an empirical study on github copilot. In: Proceedings of the 45th International Conference on Software Engineering (ICSE), 2023. 2149–2160
176 Zhong L, Wang Z. Can LLM replace stack overflow? A study on robustness and reliability of large language model code generation. In: Proceedings of the AAAI Conference on Artificial Intelligence, 2024. 21841–21849
177 Coignion T, Quinton C, Rouvoy R. A performance study of LLM-generated code on LeetCode. In: Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering, 2024. 79–89
178 Jin K, Wang C Y, Pham H V, et al. Can ChatGPT support developers? An empirical evaluation of large language models for code generation. In: Proceedings of the 21st International Conference on Mining Software Repositories (MSR), 2024. 167–171
179 Feng Y, Vanam S, Cherukupally M, et al. Investigating code generation performance of ChatGPT with crowdsourcing social data. In: Proceedings of the 47th Annual Computers, Software, and Applications Conference (COMPSAC), 2023. 876–885
180 Kou B, Chen S, Wang Z, et al. Do large language models pay similar attention like human programmers when generating code? Proc ACM Softw Eng, 2024, 1: 2261–2284
181 Liu C, Bao X, Zhang H, et al. Guiding ChatGPT for better code generation: an empirical study. In: Proceedings of IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2024. 102–113
182 Yan D, Gao Z, Liu Z. A closer look at different difficulty levels code generation abilities of ChatGPT. In: Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2023. 1887–1898
183 Liu C, Bao X, Zhang H, et al. Improving ChatGPT prompt for code generation. ArXiv:230508360
184 Cassano F, Gouwar J, Nguyen D, et al. MultiPL-E: a scalable and polyglot approach to benchmarking neural code generation. IEEE Trans Softw Eng, 2023, 49: 3675–3691
185 Yu H, Shen B, Ran D, et al. Codereval: a benchmark of pragmatic code generation with generative pre-trained models. In: Proceedings

of the 46th IEEE/ACM International Conference on Software Engineering, 2024. 1–12
186 Chen M, Zhang H, Wan C, et al. On the effectiveness of large language models in domain-specific code generation. ArXiv:231201639
187 Bairi R, Sonwane A, Kanade A, et al. CodePlan: repository-level coding using LLMs and planning. Proc ACM Softw Eng, 2024, 1: 675–698
188 Ma Z, An S, Xie B, et al. Compositional API recommendation for library-oriented code generation. In: Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension, 2024. 87–98
189 Liu M, Yang T, Lou Y, et al. Codegen4libs: a two-stage approach for library-oriented code generation. In: Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2023. 434–445
190 Li J, Li Y, Li G, et al. Skcoder: a sketch-based approach for automatic code generation. In: Proceedings of the 45th International Conference on Software Engineering (ICSE), 2023. 2124–2135
191 Lin F, Kim D J, Chen T H. When LLM-based code generation meets the software development process. ArXiv:240315852
192 Huang D, Bu Q, Zhang J M, et al. Agentcoder: multi-agent-based code generation with iterative testing and optimisation. ArXiv:231213010
193 Zhang K, Li J, Li G, et al. Codeagent: enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. ArXiv:240107339
194 Liu F, Li G, Zhao Q H, et al. Learning to represent code semantics. Sci China Inf Sci, 2025, 68: 172101
195 Qin Y H, Wang S W, Lin B, et al. GTE: learning code AST representation efficiently and effectively. Sci China Inf Sci, 2025, 68: 139101
196 Agarwal M, Shen Y, Wang B, et al. Structured code representations enable data-efficient adaptation of code language models. ArXiv:240110716
197 He J, Zhou X, Xu B, et al. Representation learning for stack overflow posts: How far are we? ACM Trans Softw Eng Methodol, 2024, 33: 1–24
198 Cui L, Yin J, Cui J, et al. Api2vec++: boosting API sequence representation for malware detection and classification. IEEE Trans Softw Eng, 2024, 50: 2142–2162
199 Lin Y, Wan C, Bai S, et al. Vargan: adversarial learning of variable semantic representations. IEEE Trans Softw Eng, 2024, 50: 1505–1517
200 Li J, Liu F, Li J, et al. Mcodesearcher: multi-view contrastive learning for code search. In: Proceedings of the 14th Asia-Pacific Symposium on Internetware, 2023. 270–280
201 Liu S, Wu B, Xie X, et al. Contrabert: enhancing code pre-trained models via contrastive learning. In: Proceedings of the 45th International Conference on Software Engineering (ICSE), 2023. 2476–2487
202 Shi Z, Xiong Y, Zhang X, et al. Cross-modal contrastive learning for code search. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME), 2022. 94–105
203 Shi Z, Xiong Y, Zhang Y, et al. Improving code search with multi-modal momentum contrastive learning. In: Proceedings of the 31st International Conference on Program Comprehension (ICPC), 2023. 280–291
204 Li X, Gong Y, Shen Y, et al. Coderetriever: a large scale contrastive pre-training method for code search. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing, 2022. 2898–2910
205 Shi E, Wang Y, Gu W, et al. Cocosoda: effective contrastive learning for code search. In: Proceedings of the 45th International Conference on Software Engineering (ICSE), 2023. 2198–2210
206 Salza P, Schwizer C, Gu J, et al. On the effectiveness of transfer learning for code search. IEEE Trans Softw Eng, 2023, 49: 1804–1822
207 Chi K, Li C, Ge J, et al. An empirical study on code search pre-trained models: academic progresses vs. industry requirements. In: Proceedings of the 15th Asia-Pacific Symposium on Internetware, 2024. 41–50
208 Li L, Liang B, Chen L, et al. Cross-modal retrieval-enhanced code Summarization based on joint learning for retrieval and generation. Inf Softw Tech, 2024, 175: 107527
209 Fang C, Sun W, Chen Y, et al. Esale: enhancing code-summary alignment learning for source code summarization. IEEE Trans Softw Eng, 2024, 50: 2077–2095
210 Wang D, Chen B, Li S, et al. One adapter for all programming languages? Adapter tuning for code search and summarization. In: Proceedings of the 45th International Conference on Software Engineering (ICSE), 2023. 5–16
211 Geng M, Wang S, Dong D, et al. Large language models are few-shot summarizers: multi-intent comment generation via in-context learning. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, 2024. 1–13
212 Wang C, Lou Y, Liu J, et al. Generating variable explanations via zero-shot prompt learning. In: Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2023. 748–760
213 Ahmed T, Pai K S, Devanbu P, et al. Automatic semantic augmentation of language model prompts (for code summarization). In: Proceedings of the 46th International Conference on Software Engineering (ICSE), 2024. 2720–2732
214 Rukmono S A, Ochoa L, Chaudron M R. Achieving high-level software component summarization via hierarchical chain-of-thought prompting and static code analysis. In: Proceedings of IEEE International Conference on Data and Software Engineering (ICoDSE), 2023. 7–12
215 Li J, Zhang Y, Karas Z, et al. Do machines and humans focus on similar code? Exploring explainability of large language models in code summarization. In: Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension, 2024. 47–51
216 Jin X, Lin Z. SimLLM: calculating semantic similarity in code summaries using a large language model-based approach. Proc ACM Softw Eng, 2024, 1: 1376–1399
217 Jin X, Larson J, Yang W, et al. Binary code summarization: benchmarking ChatGPT/GPT-4 and other large language models. ArXiv:231209601
218 Sun W, Fang C, You Y, et al. Automatic code summarization via ChatGPT: How far are we? ArXiv:230512865
219 Yang Z, Liu F, Yu Z, et al. Exploring and unleashing the power of large language models in automated code translation. ArXiv:240414646
220 Wang B, Li R, Li M, et al. Transmap: pinpointing mistakes in neural code translation. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2023. 999–1011
221 Pan R, Ibrahimzada A R, Krishna R, et al. Lost in translation: a study of bugs introduced by large language models while translating code. In: Proceedings of the 46th International Conference on Software Engineering (ICSE), 2024. 995–1007
222 Jiao M, Yu T, Li X, et al. On the evaluation of neural code translation: taxonomy and benchmark. In: Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2023. 1529–1541
223 Zhang J, Nie P, Li J J, et al. Multilingual code co-evolution using large language models. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2023. 695–707
224 Nam D, Macvean A, Hellendoorn V, et al. In-ide generation-based information support with a large language model. ArXiv:230708177
225 Wan Y, Zhao W, Zhang H, et al. What do they capture? A structural analysis of pre-trained language models for source code. In: Proceedings of the 44th International Conference on Software Engineering, 2022. 2377–2388
226 Artuso F, Mormando M, Di Luna G A, et al. Binbert: binary code understanding with a fine-tunable and execution-aware transformer. IEEE Trans Dependable Secure Comput, 2025, 22: 308–326
227 Shi J, Jiang S, Xu B, et al. ShellGPT: generative pre-trained transformer model for shell language understanding. In: Proceedings of the 34th International Symposium on Software Reliability Engineering (ISSRE), 2023. 671–682
228 Jain N, Vaidyanath S, Iyer A, et al. Jigsaw: large language models meet program synthesis. In: Proceedings of the 44th International Conference on Software Engineering, 2022. 1219–1231
229 Liventsev V, Grishina A, Härmä A, et al. Fully autonomous programming with large language models. ArXiv:230410423
230 Vella Zarb D, Parks G, Kipouros T. Synergistic utilization of LLMs for program synthesis. In: Proceedings of the Genetic and Evolutionary

Computation Conference Companion, 2024. 539–542
231 Khan J Y, Khondaker M T I, Uddin G, et al. Automatic detection of five API documentation smells: practitioners' perspectives. In: Proceedings of IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2021. 318–329
232 Huang Q, Wu Y, Xing Z, et al. Adaptive intellect unleashed: the feasibility of knowledge transfer in large language models. ArXiv:230804788
233 Wang S, Jean S, Sengupta S, et al. Measuring and mitigating constraint violations of in-context learning for utterance-to-API semantic parsing. ArXiv:230515338
234 Zhuo T Y, Du X, Xing Z, et al. Pop quiz! Do pre-trained code models possess knowledge of correct API names? ArXiv:230907804
235 Gilbert H, Sandborn M, Schmidt D C, et al. Semantic compression with large language models. ArXiv:230412512
236 von der Mosel J, Trautsch A, Herbold S. On the validity of pre-trained transformers for natural language processing in the software engineering domain. IIEEE Trans Softw Eng, 2022, 49: 1487–1507
237 Baral T, Rahman S, Chanumolu B N, et al. Optimizing continuous development by detecting and preventing unnecessary content generation. In: Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2023. 901–913
238 Zhang J, Liu S, Gong L, et al. BEQAIN: an effective and efficient identifier normalization approach with BERT and the question answering system. IIEEE Trans Softw Eng, 2022, 49: 2597–2620
239 Alsayed A S, Dam H K, Nguyen C. Microrec: leveraging large language models for microservice recommendation. In: Proceedings of the 21st International Conference on Mining Software Repositories (MSR), 2024. 419–430
240 Nasir M U, Earle S, Togelius J, et al. Llmatic: neural architecture search via large language models and quality diversity optimization. In: Proceedings of the Genetic and Evolutionary Computation Conference, 2024. 1110–1118
241 Banday B H, Islam T Z, Marathe A. Perfgen: a synthesis and evaluation framework for performance data using generative AI. In: Proceedings of the 48th Annual Computers, Software, and Applications Conference (COMPSAC), 2024. 188–197
242 Le D A, Bui A M, Nguyen P T, et al. Good things come in three: generating so post titles with pre-trained models, self improvement and post ranking. ArXiv:240615633
243 Jesse K, Devanbu P T, Sawant A. Learning to predict user-defined types. IEEE Trans Softw Eng, 2022, 49: 1508–1522
244 Qian C, Liu W, Liu H, et al. Chatdev: communicative agents for software development. In: Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics, 2024. 15174–15186
245 Tufano M, Drain D, Svyatkovskiy A, et al. 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. 54–64
246 Dinella E, Ryan G, Mytkowicz T, et al. Toga: a neural method for test oracle generation. In: Proceedings of the 44th International Conference on Software Engineering, 2022. 2130–2141
247 Zhang Q, Fang C, Zheng Y, et al. Improving deep assertion generation via fine-tuning retrieval-augmented pre-trained language models. ACM Trans Softw Eng Methodol, 2025, 34: 209
248 He Y, Huang J, Yu H, et al. An empirical study on focal methods in deep-learning-based approaches for assertion generation. Proc ACM Softw Eng, 2024, 1: 1750–1771
249 Pulavarthi V, Nandal D, Dan S, et al. Assertionbench: a benchmark to evaluate large-language models for assertion generation. ArXiv:240618627
250 Endres M, Fakhoury S, Chakraborty S, et al. Can large language models transform natural language intent into formal method postconditions? Proc ACM Softw Eng, 2024, 1: 1889–1912
251 Hossain S B, Dwyer M. Togll: correct and strong test oracle generation with LLMs. ArXiv:240503786
252 Xu X, Zhang Z, Feng S, et al. Lmpa: improving decompilation by synergy of large language model and program analysis. ArXiv:230602546
253 Wong W K, Wang H, Li Z, et al. Refining decompiled C code with large language models. ArXiv:231006530
254 Hu P, Liang R, Chen K. Degpt: optimizing decompiler output with LLM. In: Proceedings of Network and Distributed System Security Symposium, 2024. 1–16
255 Tan H, Luo Q, Li J, et al. Llm4decompile: decompiling binary code with large language models. ArXiv:240305286
256 Jiang N, Wang C, Liu K, et al. Nova$^+$: generative language models for binaries. ArXiv:231113721
257 Armengol-Estapé J, Woodruff J, Cummins C, et al. Slade: a portable small language model decompiler for optimized assembly. In: Proceedings of IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2024. 67–80
258 She X, Zhao Y, Wang H. Wadec: decompile webassembly using large language model. ArXiv:240611346
259 Shang X, Cheng S, Chen G, et al. How far have we gone in stripped binary code understanding using large language models. ArXiv:240409836
260 Li T O, Zong W, Wang Y, et al. Finding failure-inducing test cases with ChatGPT. ArXiv:230411686
261 Song Y, Xie X Y, Xu B W. When debugging encounters artificial intelligence: state of the art and open challenges. Sci China Inf Sci, 2024, 67: 141101
262 Wu Y, Liu Y, Yin Y, et al. Smartfl: semantics based probabilistic fault localization. IEEE Trans Softw Eng, 2025, 51: 2161–2180
263 Zhu Z, Wang Y, Li Y. Trobo: a novel deep transfer model for enhancing cross-project bug localization. In: Proceedings of the 14th International Conference on Knowledge Science, Engineering and Management, 2021. 529–541
264 Ciborowska A, Damevski K. Fast changeset-based bug localization with BERT. In: Proceedings of the 44th International Conference on Software Engineering (ICSE'22), 2022. 946–957
265 Chandramohan M, Nguyen D Q, Krishnan P, et al. Supporting cross-language cross-project bug localization using pre-trained language models. ArXiv:240702732
266 Wu Y, Li Z, Zhang J M, et al. Large language models in fault localisation. ArXiv:230815276
267 Kang S, An G, Yoo S. A quantitative and qualitative evaluation of LLM-based explainable fault localization. Proc ACM Softw Eng, 2024, 1: 1424–1446
268 Yang A Z, Le Goues C, Martins R, et al. Large language models for test-free fault localization. In: Proceedings of the 46th International Conference on Software Engineering (ICSE), 2024. 165–176
269 Widyasari R, Ang J W, Nguyen T G, et al. Demystifying faulty code: step-by-step reasoning for explainable fault localization. In: Proceedings of IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2024. 568–579
270 Deng Y, Xia C S, Peng H, et al. 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, 2023. 423–435
271 Deng Y, Xia C S, Yang C, et al. Large language models are edge-case generators: crafting unusual programs for fuzzing deep learning libraries. In: Proceedings of the 46th International Conference on Software Engineering (ICSE), 2024. 841–853
272 Yang C, Deng Y, Lu R, et al. White-box compiler fuzzing empowered by large language models. ArXiv:231015991
273 Eom J, Jeong S, Kwon T. Covrl: fuzzing JavaScript engines with coverage-guided reinforcement learning for LLM-based mutation. ArXiv:240212222
274 Meng R, Mirchev M, Böhme M, et al. Large language model guided protocol fuzzing. In: Proceedings of the 31st Annual Network and Distributed System Security Symposium, 2024. 1–17
275 Wang J, Yu L, Luo X. Llmif: augmented large language model for fuzzing IoT devices. In: Proceedings of IEEE Symposium on Security and Privacy (SP), 2024. 196–196
276 Xia C S, Paltenghi M, Le Tian J, et al. Fuzz4all: universal fuzzing with large language models. In: Proceedings of the 46th International Conference on Software Engineering (ICSE), 2024. 1547–1559
277 Hu J, Zhang Q, Yin H. Augmenting greybox fuzzing with generative AI. ArXiv:230606782

278 Dakhama A, Even-Mendoza K, Langdon W B, et al. Searchgem5: towards reliable gem5 with search based software testing and large language models. In: Proceedings of International Symposium on Search Based Software Engineering, 2023. 160–166
279 Zhang H, Rong Y, He Y, et al. Llamafuzz: large language model enhanced greybox fuzzing. ArXiv:240607714
280 Zhang C, Zheng Y, Bai M, et al. How effective are they? Exploring large language model based fuzz driver generation. In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2024. 1223–1235
281 Shou C, Liu J, Lu D, et al. Llm4fuzz: guided fuzzing of smart contracts with large language models. ArXiv:240111108
282 Yang C, Zhao Z, Zhang L. Kernelgpt: enhanced kernel fuzzing via large language models. ArXiv:240100563
283 Asmita, Oliinyk Y, Scott M, et al. Fuzzing BusyBox: leveraging LLM and crash reuse for embedded bug unearthing. In: Proceedings of the 33rd USENIX Security Symposium (USENIX Security 24), 2024. 883–900
284 Zhang Y, Zhang W, Ran D, et al. Learning-based widget matching for migrating GUI test cases. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, 2024. 1–13
285 Liu Z, Chen C, Wang J, et al. Fill in the blank: context-aware automated text input generation for mobile GUI testing. ArXiv:221204732
286 Liu Z, Chen C, Wang J, et al. Make LLM a testing expert: bringing human-like interaction to mobile GUI testing via functionality-aware decisions. ArXiv:231015780
287 Liu Z, Li C, Chen C, et al. Vision-driven automated mobile GUI testing via multimodal large language model. ArXiv:240703037
288 Yoon J, Feldt R, Yoo S. Intent-driven mobile GUI testing with autonomous large language model agents. In: Proceedings of IEEE Conference on Software Testing, Verification and Validation (ICST), 2024. 129–139
289 Khanfir A, Degiovanni R, Papadakis M, et al. Efficient mutation testing via pre-trained language models. ArXiv:230103543
290 Ibrahimzada A R, Chen Y, Rong R, et al. Automated bug generation in the era of large language models. ArXiv:231002407
291 Nong Y, Ou Y, Pradel M, et al. Vulgen: realistic vulnerability generation via pattern mining and deep learning. In: Proceedings of the 45th International Conference on Software Engineering, 2023. 2527–2539
292 Garg A, Degiovanni R, Papadakis M, et al. On the coupling between vulnerabilities and LLM-generated mutants: a study on vul4j dataset. In: Proceedings of IEEE Conference on Software Testing, Verification and Validation (ICST), 2024. 305–316
293 Wang B, Chen M, Lin Y, et al. An exploratory study on using large language models for mutation testing. ArXiv:240609843
294 Tian Z, Shu H, Wang D, et al. Large language models for equivalent mutant detection: how far are we? ArXiv:240801760
295 He P, Meister C, Su Z. Structure-invariant testing for machine translation. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020. 961–973
296 Gupta S, He P, Meister C, et al. 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. 863–875
297 Sun Z, Zhang J M, Xiong Y, et al. Improving machine translation systems via isotopic replacement. In: Proceedings of the 44th International Conference on Software Engineering, 2022. 1181–1192
298 Yu B, Hu Y, Mang Q, et al. 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, 2023. 883–894
299 Wang W, Huang J T, Wu W, et al. Mttm: metamorphic testing for textual content moderation software. In: Proceedings of the 45th International Conference on Software Engineering (ICSE), 2023. 2387–2399
300 Liu Z, Feng Y, Chen Z. 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. 115–126
301 Liu Z, Feng Y, Yin Y, et al. Qatest: a uniform fuzzing framework for question answering systems. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022. 1–12
302 Deng G, Liu Y, Mayoral-Vilches V, et al. Pentestgpt: an LLM-empowered automatic penetration testing tool. ArXiv:230806782
303 Wu L, Zhong X, Liu J, et al. Ptgroup: an automated penetration testing framework using LLMs and multiple prompt chains. In: Proceedings of International Conference on Intelligent Computing, 2024. 220–232
304 Happe A, Cito J. Getting pwn'd by AI: penetration testing with large language models. ArXiv:230800121
305 Vikram V, Lemieux C, Padhye R. Can large language models write good property-based tests? ArXiv:230704346
306 Mohajer M M, Aleithan R, Harzevili N S, et al. Skipanalyzer: an embodied agent for code analysis with large language models. ArXiv:231018532
307 Hao Y, Chen W, Zhou Z, et al. E&V: prompting large language models to perform static analysis by pseudo-code execution and verification. ArXiv:231208477
308 Li H, Hao Y, Zhai Y, et al. Enhancing static analysis for practical bug detection: an LLM-integrated approach. Proc ACM Program Lang, 2024, 8: 474–499
309 Li H, Hao Y, Zhai Y, et al. Assisting static analysis with large language models: a ChatGPT experiment. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2023. 2107–2111
310 Fang C, Miao N, Srivastav S, et al. Large language models for code analysis: do LLMs really do their job? In: Proceedings of 33rd USENIX Security Symposium (USENIX Security 24), 2024. 829–846
311 Tufano M, Drain D, Svyatkovskiy A, et al. Unit test case generation with transformers and focal context. ArXiv:200905617
312 Alagarsamy S, Tantithamthavorn C, Aleti A. A3test: assertion-augmented automated test case generation. ArXiv:230210352
313 Rao N, Jain K, Alon U, et al. Cat-LM training language models on aligned code and tests. In: Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2023. 409–420
314 Schäfer M, Nadi S, Eghbali A, et al. An empirical evaluation of using large language models for automated unit test generation. IIEEE Trans Softw Eng, 2024, 50: 85–105
315 Dakhel A M, Nikanjam A, Majdinasab V, et al. Effective test generation using pre-trained large language models and mutation testing. ArXiv:230816557
316 Ryan G, Jain S, Shang M, et al. Code-aware prompting: a study of coverage guided test generation in regression setting using LLM. ArXiv:240200097
317 Pizzorno J A, Berger E D. Coverup: coverage-guided LLM-based test generation. ArXiv:240316218
318 Wang Z, Liu K, Li G, et al. Hits: high-coverage LLM-based unit test generation via method slicing. ArXiv:240811324
319 Gu S, Fang C, Zhang Q, et al. Testart: improving LLM-based unit test via co-evolution of automated generation and repair iteration. ArXiv:240803095
320 Karmarkar H, Agrawal S, Chauhan A, et al. Navigating confidentiality in test automation: a case study in LLM driven test data generation. In: Proceedings of IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2024. 337–348
321 Chen Y, Hu Z, Zhi C, et al. Chatunitest: a framework for LLM-based test generation. In: Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, 2024. 572–576
322 Ni C, Wang X, Chen L, et al. Casmodatest: a cascaded and model-agnostic self-directed framework for unit test generation. ArXiv:240615743
323 Lemieux C, Inala J P, Lahiri S K, et al. Codamosa: escaping coverage plateaus in test generation with pre-trained large language models. In: Proceedings of the 45th International Conference on Software Engineering (ICSE), 2023. 919–931
324 Xiao D, Guo Y, Li Y, et al. Optimizing search-based unit test generation with large language models: an empirical study. In: Proceedings of the 15th Asia-Pacific Symposium on Internetware, 2024. 71–80
325 Tang Y, Liu Z, Zhou Z, et al. ChatGPT vs SBST: a comparative assessment of unit test suite generation. IEEE Trans Softw Eng, 2024, 50: 1340–1359
326 Yang L, Yang C, Gao S, et al. An empirical study of unit test generation with large language models. ArXiv:240618181

327 Guilherme V, Vincenzi A. An initial investigation of ChatGPT unit test generation capability. In: Proceedings of the 8th Brazilian Symposium on Systematic and Automated Software Testing, 2023. 15–24
328 Yuan Z, Liu M, Ding S, et al. Evaluating and improving ChatGPT for unit test generation. Proc ACM Softw Eng, 2024, 1: 1703–1726
329 Ouédraogo W C, Kaboré K, Tian H, et al. Large-scale, independent and comprehensive study of the power of LLMs for test case generation. ArXiv:240700225
330 Deljouyi A, Koohestani R, Izadi M, et al. Leveraging large language models for enhancing the understandability of generated unit tests. ArXiv:240811710
331 Cui C, Li T, Wang J, et al. Large language models for mobile gui text input generation: an empirical study. ArXiv:240408948
332 Zhang Y, Song W, Ji Z, et al. How well does LLM generate security tests? ArXiv:231000710
333 Pan R, Ghaleb T A, Briand L. LTM: scalable and black-box similarity-based test suite minimization based on language models. ArXiv:230401397
334 Chakraborty S, Krishna R, Ding Y, et al. Deep learning based vulnerability detection: are we there yet. IEEE Trans Softw Eng, 2022, 48: 3280–3296
335 Li Y, Wang S, Nguyen T N. 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. 292–303
336 Du X, Wen M, Zhu J, et al. Generalization-enhanced code vulnerability detection via multi-task instruction fine-tuning. ArXiv:240603718
337 Sun Y, Wu D, Xue Y, et al. Gptscan: detecting logic vulnerabilities in smart contracts by combining GPT with program analysis. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024. 1–13
338 Steenhoek B, Rahman M M, Jiles R, et al. An empirical study of deep learning models for vulnerability detection. In: Proceedings of the 45th International Conference on Software Engineering (ICSE), 2023. 2237–2248
339 Zhang C, Liu H, Zeng J, et al. Prompt-enhanced software vulnerability detection using ChatGPT. ArXiv:230812697
340 Noever D. Can large language models find and fix vulnerable software? ArXiv:230810345
341 Le T, Tran T, Cao D, et al. Kat: dependency-aware automated api testing with large language models. In: Proceedings of IEEE Conference on Software Testing, Verification and Validation (ICST), 2024. 82–92
342 Liu Z, Liu K, Xia X, et al. Towards more realistic evaluation for neural test oracle generation. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2023. 589–600
343 Hossain S B, Filieri A, Dwyer M B, et al. Neural-based test oracle generation: a large-scale evaluation and lessons learned. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2023. 120–132
344 Xue Z, Gao Z, Wang S, et al. Selfpico: self-guided partial code execution with LLMs. ArXiv:240716974
345 Sun J, Xing Z, Lu Q, et al. Silent vulnerable dependency alert prediction with vulnerability key aspect explanation. In: Proceedings of the 45th International Conference on Software Engineering (ICSE), 2023. 970–982
346 Liu K, Liu Y, Chen Z, et al. LLM-powered test case generation for detecting tricky bugs. ArXiv:240410304
347 Zhang H, Lu S, Li Z, et al. CodeBERT-Attack: adversarial attack against source code deep learning models via pre-trained model. J Softw Evolu Process, 2024, 36: e2571
348 Chen Y, Xie H, Ma M, et al. Automatic root cause analysis via large language models for cloud incidents. In: Proceedings of the 19th European Conference on Computer Systems, 2024. 674–688
349 Roy D, Zhang X, Bhave R, et al. Exploring LLM-based agents for root cause analysis. In: Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, 2024. 208–219
350 Ge X, Fang C, Zhang Q, et al. Pre-trained model-based actionable warning identification: a feasibility study. ArXiv:240302716
351 Zhang M, Tian Y, Xu Z, et al. LPR: large language models-aided program reduction. In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2024. 261–273
352 Zhang T, Irsan I C, Thung F, et al. Cupid: leveraging ChatGPT for more accurate duplicate bug report detection. ArXiv:230810022
353 Plein L, Bissyandé T F. Can LLMs demystify bug reports? ArXiv:231006310
354 Kang S, Yoon J, Yoo S. Large language models are few-shot testers: exploring LLM-based general bug reproduction. In: Proceedings of the 45th International Conference on Software Engineering (ICSE), 2023. 2312–2323
355 Huang Y, Wang J, Liu Z, et al. Crashtranslator: automatically reproducing mobile application crashes directly from stack trace. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, 2024. 1–13
356 Feng S, Chen C. Prompting is all your need: automated Android bug replay with large language models. ArXiv:230601987
357 Zhang Z, Saber T. Assessing the code clone detection capability of large language models. In: Proceedings of the 4th International Conference on Code Quality (ICCQ), 2024. 75–83
358 Moumoula M B, Kabore A K, Klein J, et al. Large language models for cross-language code clone detection. ArXiv:240804430
359 Dou S, Shan J, Jia H, et al. Towards understanding the capability of large language models on code clone detection: a survey. ArXiv:230801191
360 Dong C, Jiang Y, Zhang Y, et al. ChatGPT-based test generation for refactoring engines enhanced by feature analysis on examples. In: Proceedings of the 47th International Conference on Software Engineering (ICSE), 2025. 2714–2725
361 Liu H, Wang Y, Wei Z, et al. Refbert: a two-stage pre-trained framework for automatic rename refactoring. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2023. 740–752
362 Shirafuji A, Oda Y, Suzuki J, et al. Refactoring programs using large language models with few-shot examples. In: Proceedings of the 30th Asia-Pacific Software Engineering Conference (APSEC), 2023. 151–160
363 Zhang Z, Xing Z, Ren X, et al. Refactoring to pythonic idioms: a hybrid knowledge-driven approach leveraging large language models. Proc ACM Softw Eng, 2024, 1: 1107–1128
364 Pomian D, Bellur A, Dilhara M, et al. Next-generation refactoring: combining LLM insights and ide capabilities for extract method. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME), 2024. 275–287
365 Tufano R, Masiero S, Mastropaolo A, et al. Using pre-trained models to boost code review automation. In: Proceedings of the 44th International Conference on Software Engineering, 2022. 2291–2302
366 Li L, Yang L, Jiang H, et al. 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. 1009–1021
367 Li Z, Lu S, Guo D, 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. 1035–1047
368 Widyasari R, Zhang T, Bouraffa A, et al. Explaining explanation: an empirical study on explanation in code reviews. ArXiv:231109020
369 Tufano R, Dabić O, Mastropaolo A, et al. Code review automation: strengths and weaknesses of the state of the art. IEEE Trans Softw Eng, 2024, 50: 338–353
370 Guo Q, Cao J, Xie X, et al. Exploring the potential of ChatGPT in automated code refinement: an empirical study. ArXiv:230908221
371 Wang G, Sun Z, Dong J, et al. Is it hard to generate holistic commit message? ACM Trans Softw Eng Methodol, 2025, 34: 1–28
372 Jung T H. Commitbert: commit message generation using pre-trained programming language model. In: Proceedings of the 1st Workshop on Natural Language Processing for Programming, 2021. 26–33
373 Wang L, Tang X, He Y, et al. Delving into commit-issue correlation to enhance commit message generation models. ArXiv:230800147
374 Li J, Faragó D, Petrov C, et al. Only diff is not enough: generating commit messages leveraging reasoning and action of large language model. Proc ACM Softw Eng, 2024, 1: 745–766
375 Xue P, Wu L, Yu Z, et al. Automated commit message generation with large language models: an empirical study and beyond.

ArXiv:240414824
376 Lopes C V, Klotzman V I, Ma I, et al. Commit messages in the age of large language models. ArXiv:240117622
377 Ren L Y, Wang Z H, Zhang L, et al. Effective random test generation for deep learning compilers. Sci China Inf Sci, 2025, 68: 1–20
378 Cummins C, Seeker V, Grubisic D, et al. Large language models for compiler optimization. ArXiv:230907062
379 Cummins C, Seeker V, Grubisic D, et al. Meta large language model compiler: Foundation models of compiler optimization. ArXiv:240702524
380 Tu H, Zhou Z, Jiang H, et al. Isolating compiler bugs by generating effective witness programs with large language models. ArXiv:230700593
381 Tao S, Meng W, Cheng Y, et al. Logstamp: automatic online log parsing based on sequence labelling. SIGMETRICS Perform Eval Rev, 2022, 49: 93–98
382 Mehrabi M, Hamou-Lhadj A, Moosavi H. The effectiveness of compact fine-tuned LLMs in log parsing. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME), 2024. 438–448
383 Le V H, Zhang H. Log parsing with prompt-based few-shot learning. In: Proceedings of the 45th International Conference on Software Engineering (ICSE), 2023. 2438–2449
384 Ma Z, Chen A R, Kim D J, et al. Llmparser: an exploratory study on using large language models for log parsing. In: Proceedings of the 46th International Conference on Software Engineering (ICSE), 2024. 1209–1221
385 Jiang Z, Liu J, Chen Z, et al. Lilac: log parsing using LLMs with adaptive parsing cache. ArXiv:231001796
386 Liu Y, Tao S, Meng W, et al. Interpretable online log analysis using large language models with prompt strategies. In: Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension, 2024. 35–46
387 Xu J, Cui Z, Zhao Y, et al. Unilog: automatic logging via LLM and in-context learning. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, 2024. 1–12
388 Liu Y, Tao S, Meng W, et al. Logprompt: prompt engineering towards zero-shot and interpretable log analysis. In: Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, 2024. 364–365
389 Xiao Y, Le V H, Zhang H. Stronger, faster, and cheaper log parsing with LLMs. ArXiv:240606156
390 Huang J, Jiang Z, Chen Z, et al. Ulog: unsupervised log parsing with large language models through log contrastive units. ArXiv:240607174
391 Ma X, Zou H, Keung J, et al. On the influence of data resampling for deep learning-based log anomaly detection: insights and recommendations. ArXiv:240503489
392 Le V H, Zhang H. Log parsing: how far can ChatGPT go? In: Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2023. 1699–1704
393 Hadadi F, Xu Q, Bianculli D, et al. Anomaly detection on unstable logs with GPT models. ArXiv:240607467
394 Huang S, Liu Y, Qi J, et al. Gloss: guiding large language models to answer questions from system logs. In: Proceedings of IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2024. 91–101
395 Mudgal P, Wouhaybi R. An assessment of ChatGPT on log data. In: Proceedings of International Conference on AI-generated Content, 2023. 148–169
396 Chai X, Zhang H, Zhang J, et al. Log sequence anomaly detection based on template and parameter parsing via BERT. IEEE Trans Dependable Secure Comput, 2025, 22: 1150–1167
397 Wang W, Lu S, Luo J, et al. Deepuserlog: deep anomaly detection on user log using semantic analysis and key-value data. In: Proceedings of the 34th International Symposium on Software Reliability Engineering (ISSRE), 2023. 172–182
398 He S, Lei Y, Zhang Y, et al. Parameter-efficient log anomaly detection based on pre-training model and LORA. In: Proceedings of the 34th International Symposium on Software Reliability Engineering (ISSRE), 2023. 207–217
399 Qi J, Huang S, Luan Z, et al. Loggpt: exploring ChatGPT for log-based anomaly detection. In: Proceedings of IEEE International Conference on High Performance Computing & Communications, Data Science & Systems, Smart City & Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys), 2023. 273–280
400 Tian H, Liu K, Kaboré A K, et al. 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. 981–992
401 Tian H, Liu K, Li Y, et al. The best of both worlds: combining learned embeddings with engineered features for accurate prediction of correct patches. ACM Trans Softw Eng Methodol, 2023, 32: 1–34
402 Tian H, Tang X, Habib A, et al. 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. 1–13
403 Le-Cong T, Luong D M, Le X B D, et al. Invalidator: automated patch correctness assessment via semantic and syntactic reasoning. IEEE Trans Softw Eng, 2023, 49: 3411–3429
404 Zhang Q, Fang C, Sun W, et al. APPT: boosting automated patch correctness prediction via fine-tuning pre-trained models. IEEE Trans Softw Eng, 2024, 50: 474–494
405 Zhou X, Xu B, Kim K, et al. Patchzero: zero-shot automatic patch correctness assessment. ArXiv:230300202
406 Molina F, Copia J M, Gorla A. Improving patch correctness analysis via random testing and large language models. In: Proceedings of IEEE Conference on Software Testing, Verification and Validation (ICST), 2024. 317–328
407 Li F, Jiang J, Sun J, et al. Hybrid automated program repair by combining large language models and program analysis. ArXiv:240600992
408 Xu J, Fu Y, Tan S H, et al. Aligning LLMs for FL-free program repair. ArXiv:240408877
409 Zhang Q, Zhao Y, Sun W, et al. Program repair: automated vs. manual. ArXiv:220305166
410 Lou Y, Yang J, Benton S, et al. When automated program repair meets regression testing—an extensive study on two million patches. ACM Trans Softw Eng Methodol, 2024, 33: 1–23
411 Jiang N, Lutellier T, Tan L. Cure: code-aware neural machine translation for automatic program repair. In: Proceedings of the 43rd International Conference on Software Engineering (ICSE), 2021. 1161–1173
412 Li Y, Wang S, Nguyen T N. Dear: a novel deep learning-based approach for automated program repair. In: Proceedings of the 44th International Conference on Software Engineering, 2022. 511–523
413 Yang B, Tian H, Ren J, et al. Multi-objective fine-tuning for enhanced program repair with LLMs. ArXiv:240412636
414 Drain D, Wu C, Svyatkovskiy A, et al. Generating bug-fixes using pretrained transformers. In: Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming, 2021. 1–8
415 Berabi B, He J, Raychev V, et al. Tfix: learning to fix coding errors with a text-to-text transformer. In: Proceedings of International Conference on Machine Learning, 2021. 780–791
416 Dehghan M, Wu J J, Fard F H, et al. Mergerepair: an exploratory study on merging task-specific adapters in code LLMs for automated program repair. ArXiv:240809568
417 Silva A, Fang S, Monperrus M. Repairllama: efficient representations and fine-tuned adapters for program repair. ArXiv:231215698
418 Wei Y, Xia C S, Zhang L. Copiloting the copilots: fusing large language models with completion engines for automated program repair. ArXiv:230900608
419 Zhang Q, Fang C, Zhang T, et al. Gamma: revisiting template-based automated program repair via mask prediction. In: Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2023. 535–547
420 Ruiz F V, Grishina A, Hort M, et al. A novel approach for automatic program repair using round-trip translation with large language models. ArXiv:240107994

421 Peng Y, Gao S, Gao C, et al. Domain knowledge matters: improving prompts with fix templates for repairing Python type errors. In: Proceedings of the 46th International Conference on Software Engineering (ICSE), 2024. 12–24
422 Xiang J, Xu X, Kong F, et al. How far can we go with practical function-level program repair? ArXiv:240412833
423 Huang K, Meng X, Zhang J, et al. An empirical study on fine-tuning large language models of code for automated program repair. In: Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2023. 1162–1174
424 Jiang N, Liu K, Lutellier T, et al. Impact of code language models on automated program repair. ArXiv:230205020
425 Fan Z, Gao X, Roychoudhury A, et al. Automated repair of programs from large language models. ArXiv:220510583
426 Sobania D, Briesch M, Hanna C, et al. An analysis of the automatic bug fixing performance of ChatGPT. ArXiv:230108653
427 Cao J, Li M, Wen M, et al. A study on prompt design, advantages and limitations of ChatGPT for deep learning program repair. ArXiv:230408191
428 Jin M, Shahriar S, Tufano M, et al. Inferfix: end-to-end program repair with LLMs. ArXiv:230307263
429 Zhao J, Yang D, Zhang L, et al. Enhancing LLM-based automated program repair with design rationales. ArXiv:240812056
430 Wadhwa N, Pradhan J, Sonwane A, et al. CORE: resolving code quality issues using LLMs. Proc ACM Softw Eng, 2024, 1: 789–811
431 Joshi H, Sanchez J C, Gulwani S, et al. Repair is nearly generation: multilingual program repair with LLMs. In: Proceedings of the AAAI Conference on Artificial Intelligence, 2023. 5131–5140
432 Yang B, Tian H, Pian W, et al. Cref: an LLM-based conversational software repair framework for programming tutors. ArXiv:240613972
433 Charalambous Y, Manino E, Cordeiro L C. Automated repair of AI code with large language models and formal verification. ArXiv:240508848
434 Xin Q, Wu H, Reiss S P, et al. Towards practical and useful automated program repair for debugging. ArXiv:240708958
435 Yang A Z, Kolak S, Hellendoorn V J, et al. Revisiting unnaturalness for automated program repair in the era of large language models. ArXiv:240415236
436 Zhang J, Cambronero J P, Gulwani S, et al. PyDex: repairing bugs in introductory Python assignments using LLMs. Proc ACM Program Lang, 2024, 8: 1100–1124
437 Bouzenia I, Devanbu P, Pradel M. Repairagent: an autonomous, LLM-based agent for program repair. ArXiv:240317134
438 Lee C, Xia C S, Huang J T, et al. A unified debugging approach via LLM-based multi-agent synergy. ArXiv:240417153
439 Yin X, Ni C, Wang S, et al. Thinkrepair: self-directed automated program repair. ArXiv:240720898
440 Xia C S, Zhang L. Keep the conversation going: fixing 162 out of 337 bugs for $0.42 each using ChatGPT. ArXiv:230400385
441 Xin Q, Wu H, Tang J, et al. Detecting, creating, repairing, and understanding indivisible multi-hunk bugs. Proc ACM Softw Eng, 2024, 1: 2747–2770
442 Wang Y, Guo T, Huang Z, et al. Revisiting evolutionary program repair via code language model. ArXiv:240810486
443 Wu Y, Zhang Y, Wang T, et al. Towards understanding Docker build faults in practice: symptoms, root causes, and fix patterns. Proc ACM Softw Eng, 2025, 2: 868–890
444 Hidvégi D, Etemadi K, Bobadilla S, et al. Cigar: cost-efficient program repair with LLMs. ArXiv:240206598
445 First E, Rabe M N, Ringer T, et al. Baldur: whole-proof generation and repair with large language models. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2023. 1229–1241
446 Hu X, Liu Z, Xia X, et al. Identify and update test cases when production code changes: a transformer-based approach. In: Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2023. 1111–1122
447 Liu J, Yan J, Xie Y, et al. Augmenting LLMs to repair obsolete test cases with static collector and neural reranker. ArXiv:240703625
448 Yaraghi A S, Holden D, Kahani N, et al. Automated test case repair using language models. ArXiv:240106765
449 Fu M, Tantithamthavorn C, Le T, et al. 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. 935–947
450 Fu M, Nguyen V, Tantithamthavorn C, et al. Vision Transformer inspired automated vulnerability repair. ACM Trans Softw Eng Methodol, 2024, 33: 1–29
451 Zhou X, Kim K, Xu B, et al. Out of sight, out of mind: better automatic vulnerability repair by broadening input ranges and sources. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024. 1–13
452 Pearce H, Tan B, Ahmad B, et al. Examining zero-shot vulnerability repair with large language models. In: Proceedings of IEEE Symposium on Security and Privacy (SP), 2023. 2339–2356
453 Tol M C, Sunar B. Zeroleak: using LLMs for scalable and cost effective side-channel patching. ArXiv:230813062
454 Dipongkor A K, Moran K. A comparative study of transformer-based neural text representation techniques on bug triaging. In: Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2023. 1012–1023
455 Lee J, Han K, Yu H. A light bug triage framework for applying large pre-trained language model. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022. 1–11
456 Ma W, Yu Y, Ruan X, et al. Pre-trained model based feature envy detection. In: Proceedings of the 20th International Conference on Mining Software Repositories (MSR), 2023. 430–440
457 Imran M M, Chatterjee P, Damevski K. Uncovering the causes of emotions in software developer communication using zero-shot LLMs. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024. 1–13
458 Cai Y, Yadavally A, Mishra A, et al. Programming assistant for exception handling with codebert. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024. 1–13
459 Jiang Y, Zhang C, He S, et al. Xpert: empowering incident management with query recommendations via large language models. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024. 1–13
460 Ahmed T, Ghosh S, Bansal C, et al. Recommending root-cause and mitigation steps for cloud incidents using large language models. In: Proceedings of the 45th International Conference on Software Engineering (ICSE), 2023. 1737–1749
461 Colavito G, Lanubile F, Novielli N, et al. Leveraging GPT-like LLMs to automate issue labeling. In: Proceedings of the 21st International Conference on Mining Software Repositories (MSR), 2024. 469–480
462 Morales G, Pragyan K, Jahan S, et al. A large language model approach to code and privacy policy alignment. In: Proceedings of IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2024. 79–90
463 Mohammed N, Lal A, Rastogi A, et al. Enabling memory safety of C programs using LLMs. ArXiv:240401096
464 Oishwee S J, Stakhanova N, Codabux Z. Large language model vs. stack overflow in addressing Android permission related challenges. In: Proceedings of the 21st International Conference on Mining Software Repositories (MSR), 2024. 373–383
465 Alhamed M, Storer T. Evaluation of context-aware language models and experts for effort estimation of software maintenance issues. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME), 2022. 129–138
466 Li Y, Ren Z, Wang Z, et al. Fine-se: integrating semantic features and expert features for software effort estimation. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, 2024. 1–12
467 Kannan J. Can LLMs configure software tools. ArXiv:231206121
468 Imran M M. Emotion classification in software engineering texts: a comparative analysis of pre-trained transformers language models. In: Proceedings of the 3rd ACM/IEEE International Workshop on NL-based Software Engineering, 2024. 73–80
469 Abedu S, Abdellatif A, Shihab E. LLM-based chatbots for mining software repositories: challenges and opportunities. In: Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering, 2024. 201–210
470 Mastropaolo A, Cooper N, Palacio D N, et al. Using transfer learning for code-related tasks. IIEEE Trans Softw Eng, 2022, 49: 1580–1598
471 Zeng Z, Tan H, Zhang H, et al. 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. 39–51

472 Niu C, Li C, Ng V, et al. An empirical comparison of pre-trained models of source code. ArXiv:230204026
473 Gao S, Wen X C, Gao C, et al. What makes good in-context demonstrations for code intelligence tasks with LLMs? In: Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2023. 761–773
474 White J, Hays S, Fu Q, et al. ChatGPT prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. In: Proceedings of Generative AI for Effective Software Development, 2024. 71–108
475 Zhou X, Kim K, Xu B, et al. The devil is in the tails: how long-tailed code distributions impact large language models. ArXiv:230903567
476 Liang W, Xiao G. An exploratory evaluation of large language models using empirical software engineering tasks. In: Proceedings of the 15th Asia-Pacific Symposium on Internetware, 2024. 31–40
477 Jalil S, Rafi S, LaToza T D, et al. ChatGPT and software testing education: promises & perils. In: Proceedings of IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2023. 4130–4137
478 Geng C, Zhang Y H, Pientka B, et al. Can ChatGPT pass an introductory level functional language programming course? ArXiv:230502230
479 Nguyen P T, Di Rocco J, Di Sipio C, et al. Is this snippet written by ChatGPT? An empirical study with a codebert-based classifier. ArXiv:230709381
480 Tian H, Lu W, Li T O, et al. Is ChatGPT the ultimate programming assistant—how far is it? ArXiv:230411938
481 Xue Y, Chen H, Bai G R, et al. Does ChatGPT help with introductory programming? An experiment of students using ChatGPT in CS1. In: Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training, 2024. 331–341
482 Zhang J M, Harman M, Ma L, et al. Machine learning testing: survey, landscapes and horizons. IEEE Trans Softw Eng, 2020, 48: 1–36
483 Zhang Q, Zhang T, Zhai J, et al. A critical review of large language model on software engineering: an example from ChatGPT and automated program repair. ArXiv:231008879
484 Wu Y, Li Z, Zhang J M, et al. Condefects: a new dataset to address the data leakage concern for LLM-based fault localization and program repair. ArXiv:231016253
485 Zhang Q, Shang Y, Fang C, et al. Testbench: evaluating class-level test case generation capability of large language models. ArXiv:240917561
486 Liu J, Xia C S, Wang Y, et al. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. ArXiv:230501210
487 Niu C, Li C, Ng V, et al. Crosscodebench: benchmarking cross-task generalization of source code models. ArXiv:230204030
488 Du X, Liu M, Wang K, et al. Evaluating large language models in class-level code generation. In: Proceedings of the 46th International Conference on Software Engineering (ICSE), 2024. 982–994
489 Li X, Dong K, Lee Y Q, et al. Coir: a comprehensive benchmark for code information retrieval models. ArXiv:240702883
490 Hu H, He C, Xie X, et al. Lrp4rag: detecting hallucinations in retrieval-augmented generation via layer-wise relevance propagation. ArXiv:240815533
491 Yang Z, Shi J, He J, et al. Natural attack for pre-trained models of code. In: Proceedings of the 44th International Conference on Software Engineering, 2022. 1482–1493
492 Sun W, Chen Y, Fang C, et al. Eliminating backdoors in neural code models for secure code understanding. Proc ACM Softw Eng, 2025, 2: 1386–1408
493 Chen Y, Sun W, Fang C, et al. Security of language models for code: a systematic literature review. ACM Trans Softw Eng Methodol, 2025. doi: 10.1145/3735554
494 Han T, Huang S, Ding Z, et al. On the effectiveness of distillation in mitigating backdoors in pre-trained encoder. ArXiv:240303846
495 Jha A, Reddy C K. Codeattack: code-based adversarial attacks for pre-trained programming language models. In: Proceedings of the AAAI Conference on Artificial Intelligence, 2023. 14892–14900
496 Schuster R, Song C, Tromer E, et al. You autocomplete me: poisoning vulnerabilities in neural code completion. In: Proceedings of the 30th USENIX Security Symposium, 2021. 1559–1575
497 Wan Y, Zhang S, Zhang H, et al. 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. 1233–1245
498 Sun W, Chen Y, Tao G, et al. Backdooring neural code search. In: Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics, 2023. 9692–9708
499 Yang Z, Xu B, Zhang J M, et al. Stealthy backdoor attack for code models. IEEE Trans Softw Eng, 2024, 50: 721–741
500 Li J, Li Z, Zhang H, et al. Poison attack and poison detection on deep source code processing models. ACM Trans Softw Eng Methodol, 2024, 33: 62
501 Li Y, Liu S, Chen K, et al. Multi-target backdoor attacks for code pre-trained models. In: Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics, 2023. 7236–7254
502 Li Z, Wang C, Ma P, et al. On extracting specialized code abilities from large language models: a feasibility study. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024. 1–13
503 Wang W, Ning H, Zhang G, et al. Rocks coding, not development: a human-centric, experimental evaluation of LLM-supported SE tasks. Proc ACM Softw Eng, 2024, 1: 699–721
504 Nam D, Macvean A, Hellendoorn V, et al. Using an LLM to help with code understanding. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024. 1–13
505 Serafini R, Otto C, Horstmann S A, et al. ChatGPT-resistant screening instrument for identifying non-programmers. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024. 1–13
506 Tanzil M H, Khan J Y, Uddin G. ChatGPT incorrectness detection in software reviews. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024. 1–12
507 Li Y, Liu Y, Deng G, et al. Glitch tokens in large language models: categorization taxonomy and effective detection. Proc ACM Softw Eng, 2024, 1: 2075–2097
508 Jiang W, Zhai J, Ma S, et al. COSTELLO: contrastive testing for embedding-based large language model as a service embeddings. Proc ACM Softw Eng, 2024, 1: 906–928
509 Yang M, Chen Y, Liu Y, et al. Distillseq: a framework for safety alignment testing in large language models using knowledge distillation. ArXiv:240710106
510 Hyun S, Guo M, Babar M A. Metal: metamorphic testing framework for analyzing large-language model qualities. In: Proceedings of IEEE Conference on Software Testing, Verification and Validation (ICST), 2024. 117–128
511 Al-Kaswan A, Izadi M, van Deursen A. Traces of memorisation in large language models for code. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024. 1–12
512 Liu Y, Le-Cong T, Widyasari R, et al. Refining ChatGPT-generated code: characterizing and mitigating code quality issues. ACM Trans Softw Eng Methodol, 2024, 33: 1–26
513 Jesse K, Ahmed T, Devanbu P T, et al. Large language models and simple, stupid bugs. In: Proceedings of the 20th International Conference on Mining Software Repositories (MSR), 2023. 563–575
514 Liu Z, Tang Y, Luo X, et al. No need to lift a finger anymore? Assessing the quality of code generation by ChatGPT. IEEE Trans Softw Eng, 2024, 50: 1548–1584
515 Idialu O J, Mathews N S, Maipradit R, et al. Whodunit: classifying code as human authored or GPT-4 generated-a case study on codechef problems. In: Proceedings of the 21st International Conference on Mining Software Repositories, 2024. 394–406

516 Karmakar A, Robbes R. Inspect: intrinsic and systematic probing evaluation for code transformers. IEEE Trans Softw Eng, 2024, 50: 220–238
517 Song D, Xie X, Song J, et al. Luna: a model-based universal analysis framework for large language models. IEEE Trans Softw Eng, 2024, 50: 1921–1948
518 Yao D, Zhang J, Harris I G, et al. Fuzzllm: a novel and universal fuzzing framework for proactively discovering jailbreak vulnerabilities in large language models. In: Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2024. 4485–4489
519 Wang W, Wang Y, Joty S, et al. Rap-gen: retrieval-augmented patch generation with codet5 for automatic program repair. ArXiv:230906057
520 Shi E, Wang Y, Zhang H, et al. Towards efficient fine-tuning of pre-trained code models: an experimental study and beyond. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2023. 39–51
521 Weyssow M, Zhou X, Kim K, et al. Exploring parameter-efficient fine-tuning techniques for code generation with large language models. ArXiv:230810462
522 Liu J, Sha C, Peng X. An empirical study of parameter-efficient fine-tuning methods for pre-trained code models. In: Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2023. 397–408
523 Zhuo T Y, Zebaze A, Suppattarachai N, et al. Astraios: parameter-efficient instruction tuning code large language models. ArXiv:240100788
524 Gao S, Zhang H, Gao C, et al. Keeping pace with ever-increasing data: towards continual learning of code intelligence models. ArXiv:230203482
525 Weyssow M, Zhou X, Kim K, et al. On the usage of continual learning for out-of-distribution generalization in pre-trained language models of code. ArXiv:230504106
526 Gao S, Mao W, Gao C, et al. Learning in the wild: towards leveraging unlabeled data for effectively tuning pre-trained code models. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024. 1–13
527 Yang A Z, Tian H, Ye H, et al. Security vulnerability detection with multitask self-instructed fine-tuning of large language models. ArXiv:240605892
528 Jana P, Jha P, Ju H, et al. Cotran: an LLM-based code translator using reinforcement learning with feedback from compiler and symbolic execution. ArXiv:230606755
529 Sun W, Fang C, You Y, et al. A prompt learning framework for source code summarization. ArXiv:231216066
530 Shi J, Yang Z, Xu B, et al. Compressing pre-trained models of code into 3 MB. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022. 1–12
531 Su C Y, McMillan C. Distilled GPT for source code summarization. Autom Softw Eng, 2024, 31: 22
532 Fu M, Tantithamthavorn C. Linevul: a transformer-based line-level vulnerability prediction. In: Proceedings of the 19th International Conference on Mining Software Repositories, 2022. 608–620
533 Yu S, Fang C, Tuo Z, et al. Vision-based mobile app GUI testing: a survey. ArXiv:231013518