A Systematic Literature Review on Large Language Models for Automated Program Repair

QUANJUN ZHANG, CHUNRONG FANG, YANG XIE, and YUXIANG MA, State Key Laboratory for Novel Software Technology, Nanjing University, China

WEISONG SUN, School of Computer Science and Engineering, Nanyang Technological University, Singapore

YUN YANG, Department of Computing Technologies, Swinburne University of Technology, Australia

ZHENYU CHEN, State Key Laboratory for Novel Software Technology, Nanjing University, China

Automated Program Repair (APR) attempts to patch software bugs and reduce manual debugging efforts. Very recently, with the advances in Large Language Models (LLMs), a rapidly increasing number of APR techniques have been proposed, significantly facilitating software development and maintenance and demonstrating remarkable performance. However, due to ongoing explorations in the LLM-based APR field, it is challenging for researchers to understand the current achievements, challenges, and potential opportunities. This work provides the first systematic literature review to summarize the applications of LLMs in APR between 2020 and 2024. We analyze 127 relevant papers from LLMs, APR and their integration perspectives. First, we categorize existing popular LLMs that are applied to support APR and outline three types of utilization strategies for their deployment. Besides, we detail some specific repair scenarios that benefit from LLMs, e.g., semantic bugs and security vulnerabilities. Furthermore, we discuss several critical aspects of integrating LLMs into APR research, e.g., input forms and open science. Finally, we highlight a set of challenges remaining to be investigated and the potential guidelines for future research. Overall, our paper provides a systematic overview of the research landscape to the APR community, helping researchers gain a comprehensive understanding of achievements and promote future research. Our artifacts are publicly available at the GitHub repository: https://github.com/iSEngLab/AwesomeLLM4APR.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**.

Additional Key Words and Phrases: Large Language Model, Automated Program Repair, LLM4APR

1 INTRODUCTION

Software bugs are recognized as inevitable and destructive, posing safety issues for users worldwide and costing billions of dollars in financial losses annually [11, 156]. It is non-trivial and time-consuming for developers to fix detected software bugs manually [13]. **Automated Program Repair (APR)** plays a crucial role in software development and maintenance with the aim of fixing software bugs without human intervention. Following the foundational work GenProg [80, 155] in 2009, APR has been extensively investigated over the past decades [43, 105], and researchers have proposed a variety of APR techniques, including heuristic-based [64, 80, 98, 177], constraint-based [31, 99, 169, 171], and pattern-based [76, 91, 92] ones. Recently, inspired by the advances of **Deep Learning (DL)**, an increasing number of learning-based APR techniques have been proposed that utilize neural network models to automatically learn bug-fixing patterns [18, 66, 84, 85, 96, 142, 174, 175, 199, 200]. Thanks to the powerful ability of DL models to learn hidden repair patterns from massive code corpora, learning-based APR has achieved remarkable performance in the last couple of years [182], attracting considerable attention from both academia and industry [69, 70, 73].

Authors' addresses: Quanjun Zhang, quanjun.zhang@smail.nju.edu.cn; Chunrong Fang, fangchunrong@nju.edu.cn; Yang Xie, serialxy@outlook.com; Yuxiang Ma, 502022320009@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China, 210093; Weisong Sun, weisong.sun@ntu.edu.sg, School of Computer Science and Engineering, Nanyang Technological University, 50 Nanyang Avenue, Singapore, 639798; Yun Yang, yyang@swin.edu.au, Department of Computing Technologies, Swinburne University of Technology, Melbourne, Australia, 3122; Zhenyu Chen, zychen@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China, 210093.

arXiv:2405.01466v1 [cs.SE] 2 May 2024

Very recently, **Large Language Models (LLMs)** have been successfully applied to a broad range of source code-related tasks [147, 184], such as code generation [82, 148, 150, 201], code summarization [132, 133, 146], and test generation [4, 24, 57, 108, 128]. Benefiting from massive model parameters and vast training data, LLMs have demonstrated impressive performance and fundamentally revolutionized the research paradigm in the **Software Engineering (SE)** community. In the domain of APR, beginning with pioneering studies, e.g., TFix [7], CIRCLE [176] and AlphaRepair [163], the community has witnessed an explosion of repair studies utilizing LLMs, already achieving considerable advantages and further indicating significant potential for future research. However, the integration of LLMs within APR is a considerably complex undertaking, making it difficult for interested researchers to understand existing work. For example, existing LLM-based APR studies encompass different research perspectives (e.g., empirical [162], technical [163] and benchmark studies [187]), repair phases (e.g., patch generation [186] and correctness assessment [183]), repair scenarios (e.g., static warnings [69] and syntax errors [70]), mode architectures (e.g., encoder-only [185] and decoder-only [100]) and model utilization paradigms (e.g., fine-tuning [176], few-shot [108] and zero-shot [186]). Despite ongoing explorations in the field, the literature currently lacks a detailed and systematic review of the applications of LLMs in APR, making it challenging for researchers to understand the multitudinous design choices of existing work and conduct follow-up research.

This Paper. To bridge this gap, our work provides the first systematic literature review on the deployment of rapidly emerging LLM-based APR studies. Based on this, the community can gain a comprehensive understanding of the strengths, weaknesses, and gaps in existing LLM-based APR techniques. We discuss what LLMs are widely adopted in state-of-the-art APR research and how they are integrated into the repair workflow. We collect 127 relevant papers and perform a systematic analysis from LLMs, APR, and integration perspectives. From our analysis, we reveal the current challenges and point out possible future directions for LLM-based APR research. Overall, this work offers a thorough overview of the ongoing progress within the LLM-based APR community, aiding researchers in navigating this burgeoning field and advancing toward innovative practices.

Contributions. To sum up, this work makes the following contributions:

- *Survey Methodology.* We conduct the first systematic literature review with 127 high-quality APR papers that utilize recent LLMs to address repair challenges from 2020 to April 2024.
- *Trend Analysis.* We perform a detailed analysis of selected APR studies in terms of publication trends, distribution of publication venues, and types of contributions.
- *LLMs Perspective.* We summarize 46 LLMs utilized to support program repair and provide a summary of the typical usage and trends of different LLM categories in the APR domain.
- *APR Perspective.* We describe common repair scenarios that LLMs are applied to, encompassing 18 bug types, such as security vulnerabilities and programming problems.
- *Integration Perspective.* We discuss some key factors, including datasets, input representations and open science, that impact the performance of integrating LLMs into APR.
- *Challenges and Opportunities.* We summarize some crucial challenges of applying LLMs in the APR field, and pinpoint some potential guidelines for future LLM-based APR research.

Paper Organization. Section 2 introduces some basic concepts about APR and LLMs. Then, according to the contributions listed above, Section 3 lists our research questions (RQs) and the research methodology to collect papers related to our work. Section 4 investigates the trend and distribution of LLM-based APR studies. Section 5 summarizes LLMs that are used by existing APR studies. Section 6 illustrates the primary repair scenarios that LLMs are applied to and provides a brief description of each work. Section 7 discusses some crucial factors during the integration

of LLMs and APR, including datasets, input representation, patch correctness, and open science. Section 8 discusses some challenges and practical guidelines. Section 9 draws the conclusions.

2 BACKGROUND AND RELATED WORK

2.1 Automated Program Repair

2.1.1 Problem Description. APR, one of the most challenging activities in the SE field, aims to fix software bugs without human intervention and is regarded as a fundamental aspect of software automation. For example, in the software development and maintenance process, developers usually attempt to implement a designed functionality according to requirement specifications and write test suites to validate its correctness. If all test cases pass, the functionality is deemed correctly implemented; otherwise, developers must analyze the symptoms of failures and make necessary modifications to the buggy code snippets to ensure all test cases pass. Formally, given a buggy program P and its corresponding specification S that P fails to satisfy, APR is defined to find a minimal transformation P' of P that satisfies S. In practice, test cases are usually utilized as the specification, and in such a typical test-driven repair scenario, APR aims to find a minimal program variant that passes the available test suite, which contains at least one test case making the original buggy program P does not satisfy.
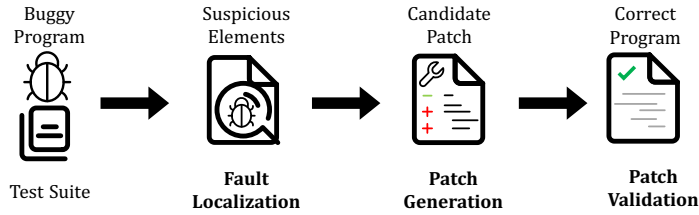

Fig. 1. The basic pipeline of automated program repair.

2.1.2 Repair Workflow. Fig. 1 illustrates the typical *generate-and-validate* workflow of APR, which is usually composed of three parts. Specifically, for a detected bug, (1) the **fault localization phase** identifies suspicious code elements that need to be fixed based on off-the-shelf localization techniques [158]; (2) the **patch generation phase** generates program variants (i.e., candidate patches) via transformation rules [182]; (3) the **patch validation phase** utilizes available test suites as the oracle to identify correct patches by dynamic execution [183]. It is important to note that a candidate patch that successfully passes the available test suite is termed a plausible patch. However, if such a plausible patch fails to generalize to additional test cases, it is considered an overfitting patch. Conversely, a plausible patch that remains effective across broader test cases is recognized as a correct patch, one that is semantically in alignment with patches written manually by developers.

2.1.3 Repair Techniques. The literature has introduced a multitude of APR techniques to generate correct patches from various perspectives. Such APR techniques can be categorized into four categories: (1) **heuristic-based APR** utilizes genetic programming to explore the search space of the correct patch, such as GenProg [80], Astor [98], ARJA [177], and SimFix [64]; (2) **constraint-based APR** usually focuses on the condition synthesis problem by treating program repair as a constraint-solving task, such as Nopol [171], Cardumen [99], ACS [169], and Dynamoth [31]; (3) **pattern-based APR** utilizes pre-defined repair templates that are usually hand-crafted by

experts to transform buggy code snippets into correct ones, such as TBar [92], FixMiner [76] and Avatar [91]; (4) **learning-based APR** treats patch generation as a neural machine translation task with the advance of DL models, such as Tufano et al. [142], SequenceR [18], CoCoNut [96], DLFix [84], CURE [66], Recoder [199], DEAR [85], SelfAPR [174], RewardAPR [175], and Tare [200].

Among the four types of APR techniques, learning-based APR has achieved remarkable performance by learning hidden bug-fixing patterns automatically from extensive source code databases [182]. Recently, inspired by the success of LLMs in NLP and SE tasks, researchers have increasingly been utilizing advanced LLMs to address software bugs [162, 163, 185, 186]. Compared with learning-based APR, such LLM-based APR techniques have demonstrated significantly better performance and have received growing attention in the community, which is the focus of our work.

2.2 Large Language Models

2.2.1 Preliminaries. LLMs refer to advanced Artificial Intelligence (AI) models that undergo extensive pre-training on large text corpora to enhance their capabilities of natural language understanding, generation, and interpretation in a human-like manner [193]. Typically, LLMs feature an enormous number of parameters, far exceeding the scale of traditional DL models, thus enabling them to assist in a wide range of tasks, such as question answering and machine translation.

In 1986, Rumelhart et al. [126] introduce Recurrent Neural Networks (RNNs), opening up the possibility of processing sequential data by maintaining a memory of previous inputs in their internal state. In 1997, Hochreiter et al. [53] introduce an extension of RNNs, namely Long Short-Term Memory Networks (LSTMs), to address the long-term dependency problem by a cell state and three types of gates. In 2017, Vaswani et al. [143] introduce Transformers to weigh the importance of different parts of the input data with the self-attention mechanism, laying the foundation of LLMs. The evolution from RNNs to LSTMs and then to Transformers represents a trajectory towards more efficiently handling longer sequences and more complex patterns in data, especially for tasks involving natural language processing. Particularly, Transformers, with their ability to manage long-range dependencies more effectively and their suitability for parallel computation, have become the dominant model architecture in many areas of AI research and application.

LLMs are typically constructed on top of the Transformer architecture mentioned above, employing a *pre-training-and-fine-tuning* paradigm. Specifically, these models undergo pre-training to acquire generic language representations through self-supervised learning on extensive unlabeled data. Subsequently, they are adapted to various downstream tasks through supervised fine-tuning using a limited amount of labeled data. Thanks to the advanced model architecture and training paradigm, LLMs have led to breakthroughs in various NLP fields, notably with models like BERT [23], GPT [119], T5 [121], PaLM [20], LLaMA [139], LLaMA2 [140], LLaMA3 [102]. Recently, research efforts have been rapidly growing in the domain of source code, leading to significant advancements with models like CodeBERT [35], CodeGPT [95], CodeT5 [152], InCoder [37], CodeGen [109], CodeLlama [124], StarCoder [83]) and StarCoder2 [94].

2.2.2 Model Categories. The literature has seen a variety of LLMs supporting NLP and SE research, which can be categorized into three main categories based on their model architectures. (1) **Encoder-only LLMs**, such as CodeBERT [35], GraphCodeBERT [46], train the encoder part of the Transformer to generate a fixed-dimensional bidirectional representation with Masked Language Modeling (MLM) and Next Sentence Prediction (NSP). MLM aims to predict the original tokens that have been randomly masked out, and NSP predicts whether two given sentences actually follow each other in a text. (2) **Decoder-only LLMs**, such as CodeGPT [95], train the decoder part of the Transformer to support auto-regressive tasks with Causal Language Modeling (CLM), which aims to predict new tokens in a sequence based on previous tokens. (3) **Encoder-decoder**

LLMs, such as CodeT5 [152], train both encoder and decoder parts of the Transformer to support sequence-to-sequence generation tasks with denoising objectives. We will summarize existing LLMs and how they are leveraged to support program repair in Section 5.1.

2.3 Related Work

In this survey, we systematically collect, summarize, and analyze APR studies empowered with advanced LLMs. Although our primary focus is the integration of LLMs with APR, given its growing prominence, there exists some related studies that explore similar aspects of either LLMs or APR. We clarify the fundamental differences between our work and prior work below.

The first group of related surveys attempts to analyze the applications of LLM more generally in SE. Zhang et al. [184] provide a detailed summarization of the LLM-based SE research from two intuitive perspectives: LLMs and their applications in SE. They introduce representative work across 30 LLMs, 15 pre-training tasks, 16 downstream tasks, and 43 specific code-related tasks. Besides, Wang et al. [147] provide a review of LLMs in software testing; Fan et al. [32] discuss the achievements and challenges of LLMs in SE; Hou et al. [55] conduct a systematic literature review on LLM4SE. However, these works target the whole software engineering/testing workflow rather than our specific program repair task. They only provide a bird's-eye view of a limited number of LLM-based APR papers, while our work provides an in-depth analysis from various perspectives. Their work indicates that program repair is the most popular scenario in which LLMs are applied within software engineering. Given the complexity and significance of program repair, we are motivated to conduct a more in-depth review of APR achievements involving LLMs.

The second group of related surveys attempts to review the achievements of APR. Gazzola et al. [43] present a survey to organize repair studies, and Monperrus et al. [105] present a bibliography of behavioral and state repair studies. Both works collect publications up to 2017, hence there is no overlap with our research as the first LLM-based APR work is published in 2020. The most highly related work to this paper is Zhang et al. [182], which provides a systematic survey to summarize the state-of-the-art in the learning-based APR community. However, they mainly focus on DL models and devote merely a single section to discussing LLM-based APR studies (only five papers), failing to include the surge of works that emerge during the last two years. In contrast, our work solely focuses on the technology of LLM applications in the APR community until April 2024, involving the trend of utilized LLMs, optimization techniques, repair scenarios, and input forms.

3 SURVEY METHODOLOGY

In this section, guided by the principles outlined by Petersen et al. [117] and Kitchenham et al. [74], we present details of our systematic literature review methodology.

3.1 Research Questions

We attempt to provide a comprehensive overview of recent LLMs' applications to APR by summarizing the relevant studies and further providing guidelines on the follow-up research. To achieve this, this systematic literature review answers the following Research Questions (RQs):

- **RQ1: What is the trend of APR studies that utilize LLMs?**
- **RQ2: Which popular LLMs have been applied to support APR?**
- **RQ3: What repair scenarios have been facilitated by LLMs?**
- **RQ4: What key factors contribute to the integration of LLMs for APR?**

3.2   Search Strategy

We search and identify primary papers following the "Quasi-Gold Standard" (QGS) [178] method, which is a common practice to construct a set of known studies for refining search strings by combining manual and automated search processes. We first conduct a manual search to identify a set of relevant studies and derive a search string from them, as detailed in Section 3.3. We then utilize the search string to perform an automated search and employ a series of relatively strict filtering steps to obtain the most relevant studies, as detailed in Section 3.4. We finally use a snowballing search to supplement the search results further, as detailed in Section 3.5. Given the abundance of relevant papers from different research communities, such as SE and AI, this search strategy allows us to capture the most pertinent papers while achieving greater efficiency than a purely manual search with a relatively rigorous process. Particularly, we undertake the following five phases to search for and identify relevant studies.

(1) **Search Sources**. This phase selects high-quality publication venues for the initial manual search and well-known digital databases for the subsequent automated search.
(2) **QGS Establishment**. This phase inspects all papers identified in the manual search and filters them by inclusion/exclusion criteria.
(3) **Search Items.** This phase definins search items with domain knowledge and established QGS.
(4) **Paper Collection and Selection**. This phase conducts an automated search using the above search items and filters the collected papers by inclusion/exclusion criteria.
(5) **Snowballing Search**. This phase conducts a snowballing search to complement the final search results.

3.3   Search Items

To perform the manual search and establish QGS, we select four top-tier SE conferences (ICSE, ESEC/FSE, ASE, ISSTA) and two journals (TOSEM and TSE) and search for relevant papers involving both program repair and LLMs. We manually inspect and identify 25 papers from 2020 to 2024 that meet our criteria, which then form the basis for constructing QGS. Following prior surveys [149, 172, 182], we divide the search items used for searching papers automatically into two groups: (1) an APR-related group containing some commonly-used keywords related to program repair; and (2) an LLM-related group containing some popular keywords related to large language models or pre-trained models. Besides, as the APR community benefits from well-maintained forums and real-time updated works, we further refine our search items by concluding some frequent keywords from three sources: a community-driven website[1], a living review of APR by Monperrus [106] and the most recent learning-based APR survey [182]. Finally, we identify a search string that includes several LLM-related and APR-related keywords frequently appearing in program repair studies that use LLMs. The complete set of search keywords is as follows:

> ("program repair" OR "software repair" OR "automatic repair" OR "code repair" OR "bug repair" OR "bug fix" OR "code fix" OR "automatic fix" OR "patch generation" OR "patch correctness" OR "patch validation" OR "fix generation" OR "code transformation" OR "code edit" OR "fix error") **AND** ("LLM(s)" OR "Large Language Model(s)" OR "Pre-trained" OR "Pretrained" OR "Pre-training" OR "Pretraining" OR "PLM(s)" OR "(Code)BERT" OR "(Code)T5" OR "(Code)GPT" OR "Codex" OR "ChatGPT" OR "(Code)Llama" OR "GPT-*" OR "neural" OR "machine" OR "deep" OR "learning" OR "transformer/transformers" OR "model/models" OR "transfer" OR "supervised")

---

[1]http://program-repair.org/bibliography.html

Table 1. Inclusion criteria and exclusion criteria.

Inclusion criteria
① The paper utilizes an LLM in its framework.
② The paper addresses the task of program repair.
③ The paper is accessible with full text.
④ The paper is only written in English.
Exclusion criteria
❶ The paper is fewer than seven pages.
❷ The paper is an old conference version extended to journals with the same authors.
❸ The paper uses repair methods to contribute to LLMs.
❹ The paper is published as an SLR, review, or survey only mentioning the development of LLMs and APR.
❺ The paper is published in a workshop or a doctoral symposium.
❻ The paper is a grey publication, e.g., a technical report or thesis.
❼ The paper falls into the category of short papers, tool demonstrations, and editorials.

Table 2. Checklist of Quality Assessment Criteria (QAC) for LLM-based APR studies.

ID	Quality Assessment Criteria
QAC1	Is the LLM adopted as the research subject rather than only as a baseline method?
QAC2	Is the impact of the paper on the APR community clearly stated?
QAC3	Is the contribution of the paper clearly stated?
QAC4	Is the paper published in a high-repute venue?
QAC5	Is the relevant artifact made open source, e.g., datasets?
QAC6	Is there a clear motivation for the paper?
QAC7	Is the implementation of the proposed approach described clearly?
QAC8	Is the experimental setup described in detail, e.g., hyper-parameters and environments?
QAC9	Are the findings clearly supported by experimental results?
QAC10	Are the key contributions and limitations of the study discussed?

It is noteworthy that we also include the list of keywords related to machine/deep learning from Zhang et al. [182], so as to avoid missing any papers related to our work as much as possible.

3.4 Study Collection and Selection

To perform the automated search, we collect potentially relevant papers by primarily searching the Google Scholar repository, ACM Digital Library, and IEEE Explorer Digital Library at the end of April 2024. Once we gather the studies based on our automated search strategy, we proceed with a filtering and deduplication phase to eliminate papers that do not align with the study objectives. First, we aim to filter out papers published before 2017, considering that the Transformer architecture [143], which forms the foundation for LLMs, is introduced in 2017. Second, we remove duplicate papers, as the same paper might be indexed by multiple databases. Thirdly, we exclude papers with fewer than seven pages [182] (Exclusion criterion ❶), resulting in a total of 283 papers. Finally, we manually inspect the titles, abstracts, keywords, and venues of the papers to include relevant papers, according to our inclusion/exclusion criteria in Table 1, reducing the number of papers to 241. It is worth noting that, about Exclusion criterion ❺, we took into account *International Automated Program Repair Workshop*, considering its relevance to the APR field and high-quality publications. Finally, we perform a full-text inspection to scrutinize the remaining papers and decide whether they are relevant and high-quality to the LLM-based APR field. Following Wang et al. [149], we answer ten questions to assess the relevance and quality of included papers, shown in Table 2. Regarding QAC4, given the nascent nature of this field and the fact that many works,

(a) Number of publications per year.   (b) Cumulative number of publications per year.
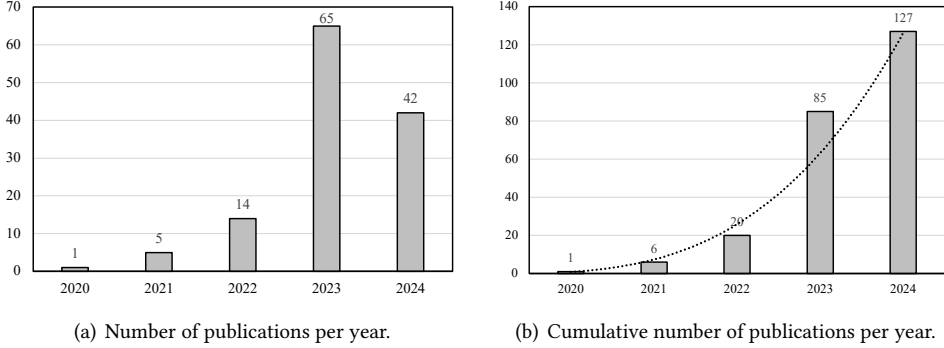
Fig. 2. Publication trends of LLM-based APR studies.

particularly those involving recently released LLMs, have not completed the peer review process, we consider papers from arXiv and select high-quality papers with the quality assessment process, to make our survey more comprehensive and up-to-date. We obtain 110 papers that are related to our work.

3.5 Snowballing Search

To ensure maximal comprehensiveness of the collected papers, we additionally employ the snowballing search [153] to manually incorporate any potential papers that are overlooked during our previous search process. The snowballing search is a common practice to examine the references (i.e., backward snowballing) or citations (i.e., forward snowballing) within a list of papers to discover additional relevant works. Specifically, we scrutinize every reference and citation within the collected papers to assess their relevance to our research objectives. This meticulous manual analysis of all potential papers through the entire study selection process ultimately leads to the inclusion of 127 papers in our survey.

4 RQ1: WHAT IS THE TREND OF APR STUDIES THAT UTILIZE LLMS?

4.1 What are the trends of publication over time?

We analyze the publication trends of APR studies empowered with LLMs between 2020 and April 2024. Although the Transformer architecture [143] has been proposed in 2017, and the groundbreaking LLM, BERT [23], has been introduced in 2018, we do not find any studies using LLMs to fix bugs before 2020. Fig. 2(a) illustrates the number of relevant studies, and we find that the number of publications from 2020 to 2023 shows a significant increase, with the number reaching 65 papers in 2023. It should be noted that we collect papers as of April 2024; thus the number of relevant studies (i.e., 42 papers) in 2024 cannot reveal the overall trend of LLM-based APR in 2024. We fit the number of publications as a power function to predict the publication trend in the last four years, and find that the slope of the curve fitting the distribution increases substantially between 2020 and 2023. Particularly, the coefficient of determination attains the peak value ($R^2 = 0.973$), estimating that there may be over 90 relevant publications using various LLMs to solve bug-fixing during 2024. We also calculate the cumulative number of publications as shown in Fig. 2(b). We perform the same power function fit ($R^2 = 0.985$) and estimate more than 140 papers by the end of 2024, which indicates that the number of relevant studies using LLMs in APR intends to experience a strong rise in the future. Overall, utilizing LLMs to address bug-fixing has become a

prevalent trend since 2020, and huge numbers of studies will adopt LLMs to address the challenges of APR in the future.

Table 3. Publication venues with LLM-based APR studies.

Acronym	Venues	Papers	Proportion
arXiv	N.A.	55	43.31%
ICSE	International Conference on Software Engineering	13	10.24%
ESEC/FSE	European Software Engineering Conference and International Symposium on Foundations of Software Engineering	9	7.09%
APR	International Automated Program Repair Workshop	6	4.72%
ICLR	International Conference on earning Representations	5	3.94%
ASE	International Conference on Automated Software Engineering	6	4.72%
TOSEM	Transactions on Software Engineering Methodology	4	3.15%
TSE	Transactions on Software Engineering	3	2.36%
ISSTA	International Symposium on Software Testing and Analysis	2	1.57%
AAAI	AAAI Conference on Artificial Intelligence	1	0.79%
ACL	The Association for Computational Linguistics	1	0.79%
COMPSAC	International Computer Software and Applications Conference	1	0.79%
GECCO	Genetic and Evolutionary Computation Conference	1	0.79%
ICSME	The International Conference on Software Maintenance and Evolution	1	0.79%
MSR	International Conference on Mining Software Repositories	1	0.79%
NeurIPS	Neural Information Processing Systems	1	0.79%
OOPSLA	Conference on Object-Oriented Programming Systems, Languages,and Applications	1	0.79%
PLDI	ACM SIGPLAN Conference on Programming Language Design\& Implementation	1	0.79%
PMLR	Proceedings of Machine Learning Research	1	0.79%
S&P	IEEE Symposium on Security and Privacy	1	0.79%
SIGCSE	The ACM Special Interest Group on Computer Science Education	1	0.79%
SLE	ACM SIGPLAN International Conference on Software Language Engineering	1	0.79%
TDSC	IEEE Transactions on Dependable and Secure Computing	1	0.79%
TIFS	IEEE Transactions on Information Forensics and Security	1	0.79%
WWW	Proceedings of the ACM Web Conference	1	0.79%
Others	N.A.	8	6.30%
Total	N.A.	127	100.00%

4.2 What is the distribution of the publication venues?

We analyze 127 studies published in various publication venues. Table 3 lists the number of relevant papers published in each publication venue. We find that 56.69% of papers are published in peer-reviewed venues, and ICSE and ESEC/FSE are the most popular venues that contain 13 papers, followed by APR (six papers), ICLR (five papers), and ASE (four papers). Besides, the top popular venues are top-tier conferences and symposiums, and only four journals have at least one relevant publication, i.e., TOSEM (four papers), TSE (three papers), TDSC (one paper), and TIFS (one paper). This finding indicates a preference for presenting the latest work at conferences due to the timeliness of conference proceedings. These collected papers are published in different research fields, including SE (e.g., TOSEM and ICSE), AI (e.g., NeurIPS, AAAI and ICLR), NLP (e.g., ACL), Programming Language (e.g., OOPSLA and PLDI) and Security (e.g., S&P, TDSC and TIFS). We also find that the remaining 43.31% of papers are non-peer-reviewed and published on arXiv. This phenomenon can likely be attributed to the rapid emergence of relevant studies in a short development time. Considering the varying quality levels of such non-peer-reviewed papers, we conducted a rigorous evaluation to ensure the inclusion of high-quality relevant papers in this work.

4.3 What is the distribution of the programming languages?

Fig. 3 presents the distribution of different programming languages targeted by existing LLM-based APR techniques. We find that Java is the most widely utilized programming language, accounting for 37% of all collected papers, followed by Python (24%) and C (11%). The possible reason lies in the availability of mature datasets for these languages, which serve as recognized benchmarks for evaluating repair techniques, such as Defects4J [71], QuixBugs [89], BFP [142], CVEfixes [8],
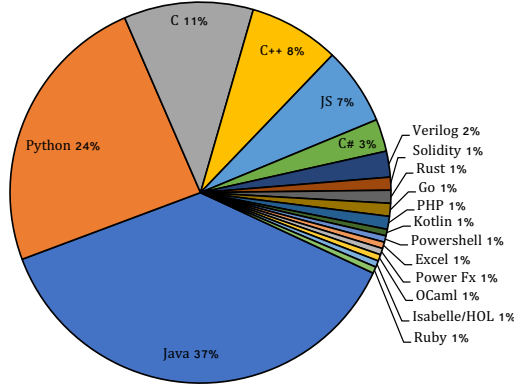
Fig. 3. Distribution of LLM-based studies across different programming languages.

Big-Vul [33], and TFix [7]. We also find that LLM-based APR encompasses a broader range of programming languages compared to traditional APR. For example, our collected papers involve 18 different programming languages in total, whereas learning-based APR techniques are typically limited to only five languages [182]. Importantly, we notice that some rare languages, previously overlooked in the APR community, are now being addressed. This broader language adaptability of LLM-based APR might stem from the inherent capabilities of LLMs to encapsulate general programming knowledge that can be transferred across multiple languages by fine-tuning [176]. Besides, LLMs' robust natural language understanding capabilities facilitate the few-shot or zero-shot repair settings with limited learning samples, which is a significant advantage over DL models [182] that typically require extensive repair corpora for training. Consequently, LLMs can efficiently handle lesser-known programming languages, like Verilog [1] and Rust [22], which are often underrepresented in previous APR research [106]. This phenomenon highlights the promising prospects and scalability of APR brought about by recent LLMs.

4.4 What are the types of the publication contributions?

Table 4. Categories of four main contributions in collected studies.

Category	Description	Papers
New Technique or Methodology	The study proposes a novel approach to address specific challenges in the APR community empowered by LLMs.	78
Empirical Study	The study explores the performance of LLMs in fixing software bugs with a quantitative and qualitative analysis.	38
Benchmark	The paper introduces a new benchmark specifically designed to evaluate the fix capabilities of LLMs for various bug types	9
Human Study	The study conducts a survey to explore practitioners' perspectives or experiences of LLMs when fixing software bugs.	2

We categorize collected papers according to their main contributions into four categories: new technique or methodology, empirical study, benchmark, and human study, as illustrated in Fig. 4. We find that 78 relevant papers are published, with the aim of proposing a novel repair approach or framework with LLMs to address various issues in the APR community. Besides, 38 papers concentrate on conducting empirical studies to explore the actual benefits of LLMs in fixing various
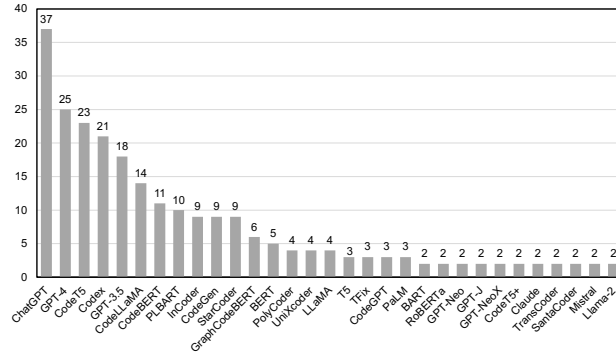
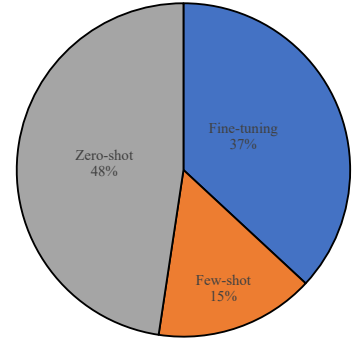Fig. 4. Distribution of existing LLMs utilized in APR.

Fig. 5. Distribution of three ways to utilize LLMs for APR.

bugs, such as the potential of fine-tuning LLMs in vulnerability repair [44, 185]. We further notice nine relevant studies constructing new benchmarks to evaluate the performance of LLMs and two papers [44, 101] administering a survey to offer insights into how practitioners or developers think about and employ LLMs to fix software bugs in practice.

Summary of Results for RQ1

(1) LLMs have shown a booming trend in fixing software bugs, with 127 papers between 2020 and 2024. (2) The number of conference papers employing LLMs for APR significantly exceeds that of journal papers, with ICSE and TOSEM being the most popular conference and journal venues, respectively. (3) LLM-based APR papers are published in different research fields, including SE, AI, and Security. (4) There are 18 programming languages that LLM-based APR has been applied to, with Java, Python, C, and C++ being the most frequently targeted. (5) LLMs have been applied to some underrepresented programming languages, such as Verilog and Rust. (6) The vast majority of collected studies primarily focus on introducing new techniques and conducting empirical research, while two papers perform user studies to understand practitioners' attitudes and experiences regarding leveraging various LLMs for solving bug-fixing tasks.

---

5 RQ2: WHICH POPULAR LLMS HAVE BEEN APPLIED TO SUPPORT APR?

5.1 What is the distribution of involved LLMs?

Plenty of LLMs, with millions to billions (even more) of parameters, have been proposed and adapted to automatically fix software bugs. We analyze all collected papers and summarize 46 relevant LLMs that are already utilized in existing APR studies. Fig. 4 illustrates the variety of different LLMs and lists the number of times these models have been utilized by prior APR studies. We only include those LLMs with more than one time in collected papers due to page limitations. As can be seen from Fig. 4, ChatGPT (37) and GPT-4 (25) from OpenAI are the two most popular LLMs, followed by CodeT5 (23) and Codex (21). We categorize these LLMs into three groups according to their architectures: encoder-only, encoder-decoder, and decoder-only LLMs.

5.1.1 Encoder-only LLMs. Encoder-only LLMs denote a category of LLMs that only utilizes the encoder stack of the Transformer architecture. Typically, these models are 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. In the APR community, prominent models like CodeBERT (11) [35] and GraphCodeBERT (6) [46] have been investigated in semantic bugs [59, 100, 163, 202] and security vulnerabilities [185]. As such LLMs only contain an encoder component that is capable of generating context-aware representations for inputs, they are particularly suited for code understanding tasks, such as code search, while not directly applicable to code generation tasks, such as program repair. Thus, in the APR community, as Zhang et al. [185] mention, researchers may need to integrate a new decoder that initializes from scratch to the pre-trained encoder to construct an encoder-decoder architecture for patch generation. We also notice such encoder-only LLMs are utilized to identify patch correctness [79, 135–137, 183, 198], as discussed in Section 7.3.

5.1.2 Enocoder-decoder LLMs. Encoder-decoder LLMs denote a category of LLMs that utilize both the encoder and decoder stacks of the Transformer architecture, thus inherently suitable for transforming one sequence into another. Particularly, the encoder takes one sequence as input and encodes it into a fixed-size hidden state, which effectively captures the semantics and meaning of the input sequence. Then the decoder processes the hidden state and produces the corresponding output sequence using attention mechanisms to refer back to parts of the input sequence as needed. Thanks to the encoder-decoder architecture, such LLMs are particularly suited for code generation tasks in a sequence-to-sequence learning setting, such as program repair. In the APR community, a mass of encoder-decoder LLMs have been widely adopted, such as CodeT5 (23) [152], PLBART (10) [2], UniXcoder (4) [45] and T5 (3) [121]. Similar to traditional learning-based APR [182], such studies usually treat APR as a natural machine translation (NMT) task by supervised sequence-to-sequence learning, such as CIRCLE [176], TFix [7], VulRepair [39] and RAP-Gen [151].

5.1.3 Decoder-only LLMs. Decoder-only LLMs denote a category of LLMs that utilize only the decoder stack of the Transformer architecture. Decoder-only LLMs are typically pre-trained using a causal language modeling (CLM) objective, learning to predict the next word in a sentence based on the preceding words. These models are specifically designed for generating sequences of text autoregressively, i.e., producing one token at a time and using what has been generated so far as context for subsequent tokens. In the APR community, decoder-only LLMs are the most popular and widely used group, compared with encoder-only and encoder-decoder LLMs. Notable repair applications of decoder-only LLMs include GPT-series models (e.g., GPT-1 [119], GPT-2 [120], GPT3 [14], GPT-3.5 [111], ChatGPT [112], and GPT-4 [113]), some open-sourced models, (e.g., CodeGPT [95], GPT-Neo [10], GPT-NeoX [9], GPT-J [145], InCoder [37], CodeGen [109], CodeLLaMA [124] and StarCoder [83]), as well as some closed-source models (e.g., Codex [16] in Fan et al. [34] and CEDAR [108]). The emergence of decoder-only LLM-based APR studies is primarily due to two reasons. The first reason is that these models can naturally perform program repair from a few examples or simple instructions without any fine-tuning. The second reason is the recent surge in decoder-only LLMs, marked by the introduction of commercial products by leading Internet companies, such as ChatGPT and GPT-4 by OpenAI.

5.2   What approaches are employed to optimize LLMs for program repair?

LLMs typically acquire general knowledge from extensive datasets. Thus, a fundamental research issue arises when integrating off-the-shelf LLMs with APR: *how to adapt general-propose LLMs to the specific program repair task*. Fig. 5 displays the prevalence of three common adaptation strategies in LLM-based APR research: fine-tuning, few-shot learning, and zero-shot learning. Our findings indicate that zero-shot learning, employed in 48% of the studies, is the most popular approach, suggesting a trend towards using LLMs as-is for program repair tasks. Meanwhile, fine-tuning is utilized in 37% of the cases, followed by few-shot learning at 15%.

5.2.1 Fine-tuning. Fine-tuning refers to a process where LLMs are further trained on a smaller, task-specific dataset. This is an intuitive way to allow LLMs to adjust their weights and biases through supervised learning, enabling them to perform as expected on new tasks that are similar to, but not exactly the same as, those they were initially trained on. In the APR community, fine-tuning is widely utilized during the early emergence of LLMs with millions of parameters, such as T5 and CodeT5, as it can significantly improve performance on the target program repair task without the need to train an LLM from scratch.

We summarize the existing APR studies via fine-tuning LLMs into three stages. First, researchers directly regard program repair as a downstream task by specific datasets, such as fine-tuning T5 [7], CodeT5 [39], CodeBERT [100], and GPT-2 [77]. Second, researchers utilize more advanced fine-tuning strategies for better performance. For example, CIRCLE utilizes continual learning to repair multiple languages with a single model, and RAP-Gen [151] utilizes retrieval-augmented generation to guide patch search space. Recently, Zirak et al. [202] empirically explore the domain shift problem in APR with two LLMs, i.e., TFix and CodeBERT, and three fine-tuning methods, i.e., Full-Fine-Tuning, Tuning-With-Light-Weight-Adapter-Layers, and Curriculum-Learning. Third, researchers conduct empirical studies to explore the actual fix capabilities of various LLMs in different repair scenarios. For example, Zhang et al. [185] fine-tune five LLMs to repair C/C++ security vulnerabilities, Wu et al. [159] involve four LLMs in Java vulnerabilities, and Jiang et al. [65] consider four LLMs on Java semantic bugs.

5.2.2 Few-shot Learning. Few-shot learning refers to the ability of LLMs to learn or adapt to new tasks with a very limited amount of data—often only a few examples. This is an effective way to use examples to help LLMs understand the targeted task and generate appropriate responses without any explicit retraining or fine-tuning. In the APR community, few-shot learning is usually utilized and impressive in LLMs, particularly with billions of parameters, as it requires LLMs' powerful ability to generalize from very limited data. Researchers typically provide LLMs with a small number of repair examples directly in the input prompt and require LLMs to generate correct patches. For example, Nashid et al. [108] construct effective prompts by retrieving similar repair demonstrations for CodeX, and Xia et al. [162] provide LLMs with examples from the same buggy project to learn the coding style.

5.2.3 Zero-shot Learning. Zero-shot learning takes the concept of few-shot learning even further by requiring LLMs to perform program repair without any explicit examples. This is a recently popular way to query LLMs to perform a variety of unseen tasks, where LLMs are given a task description and must use their pre-existing knowledge and understanding to generate a response or solution. In the APR community, zero-shot rapidly emerges following the advent of LLMs with record-breaking parameters, exemplified by ChatGPT, as it requires a powerful foundation model to perform human-like chats.

There are two typical development routes that utilize LLMs for program repair in a zero-shot learning setting. The first one is **cloze-style repair**, i.e., reframing program repair as a cloze-style task, and then invoking LLMs to predict partially correct code with the help of repair patterns, such as AlphaRepair [163], GAMMA [186], FitRepair [161] and Repilot [154]. The second one is **conversational-based repair**, i.e., constructing complex prompts with various valuable information (e.g., buggy code, failure diagnostics, even execution feedback), and then chatting with LLMs to generate correct patches, such as Pearce et al. [115], TypeFix [116], RustAssistant [22], Zhang et al. [187], Prenner et al. [118], Sobania et al. [131], and Napoli et al. [107]. Such repair routes usually require LLMs capable of processing long-text prompts and human-like conversations, thus predominantly employing powerful LLMs with billion-level parameters, like ChatGPT and GPT-4. Besides, zero-shot gets rid of training datasets, thus generalizing to various repair scenarios where
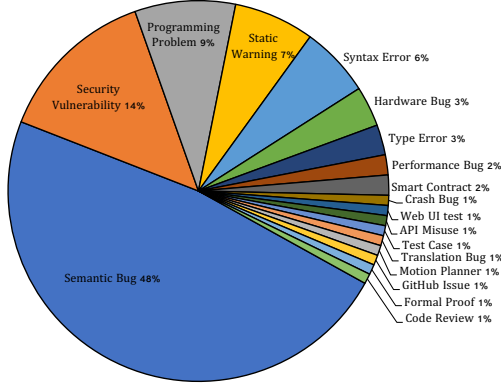
Fig. 6. Distribution of LLM-based APR papers across different bug types.

gathering training data is challenging or impossible, such as hardware bugs [1], DL programs [15] and crash bugs [30].

Overall, fine-tuning, which uses supervised learning to adapt LLMs more closely to the specifics of program repair, becomes popular with the emergence of early million-parameter-level models like CodeT5. Few-shot learning demonstrates LLMs' capability to generalize from a few examples, popular with the emergence of subsequent billion-parameter-level models like Codex. Zero-shot learning showcases LLMs' capability to tackle the program repair task without any prior direct exposure to training or examples, particularly with the emergence of recent models with tens or hundreds of billions of parameters like ChatGPT and GPT-4.

Summary of Results for RQ2

(1) We summarize 46 different LLMs already utilized to fix bugs, and these LLMs can be classified into three categories based on model architectures, i.e., encoder-only, encoder-decoder, and decoder-only. (2) Decoder-only LLMs are the most frequently utilized model architecture, and four of the top popular LLMs are decoder-only models. (3) ChatGPT, GPT-4, CodeT5, and Codex are the most popular LLMs in existing LLM-based APR studies, utilized 37, 25, 23, and 21 times, respectively. (4) We summarize three typical ways of leveraging the vast knowledge encapsulated in LLMs for the specific program repair task, i.e., fine-tuning, few-shot, and zero-shot.

---

6 RQ3: WHAT REPAIR SCENARIOS HAVE BEEN FACILITATED BY LLMS?

In this section, we conduct a comprehensive analysis of the utilization of LLMs in various repair scenarios. We categorize 18 existing repair scenarios addressed with LLMs, following the taxonomy from Monperrus et al. [106] and Zhang et al. [182], including semantic bugs, syntax bugs, and static warnings. Fig. 6 presents the distribution of repair studies across different repair scenarios that LLMs are applied to. We find that semantic bugs triggered by test suites have attracted the highest attention from the LLM-based APR research, constituting approximately 48% of the total research volume. This phenomenon is also observed in traditional learning-based APR [182], and is mainly driven by well-known benchmarks, such as Defects4J [71]. Security vulnerabilities account for about 14% of the research proportion, highlighting the significance of LLMs in addressing

cybersecurity challenges. The third highest number of studies is observed in the programming problems domain, constituting approximately 9% of the total research volume. We also find a growing interest in rare bug types that are usually ignored by prior work, such as static warnings (7%), syntax errors (6%), and hardware bugs (3%). This underscores that, thanks to LLMs' general knowledge gleaned from vast amounts of data, researchers have begun to explore repair scenarios not previously addressed in prior works.

6.1 Semantic Bugs

Semantic bugs refer to logical errors in a syntactically correct program when the code does not do what the programmer intends, even though it may compile and run without syntax errors. Considering that the majority of LLM-based APR studies are concentrated in the semantic bug field, we summarize these representative studies based on three ways of using LLMs.

6.1.1 Repair with Fine-tuning. As early as 2021, Drain et al. [29] introduce DeepDebug, which learns to fix bugs in Java methods mined from real-world GitHub repositories through fine-tuning BART on the BFP [142] dataset. In 2022, Yuan et al. [176] propose CIRCLE, an LLM-based multi-lingual APR framework by fine-tuning a pre-trained T5 with continual learning. Particularly, CIRCLE utilizes a difficulty-based rehearsal strategy to achieve lifelong learning without access to all historical data and elastic regularization to resolve catastrophic forgetting. CIRCLE is the first approach to repair multiple programming language bugs with a single repair model in the continual learning setting. Hao et al. [49, 50] propose APRFiT by fine-tuning CodeT5 and GraphCodeBERT with a curricular learning framework. In 2023, RAP-Gen [151] further improves the repair performance of fine-tuning LLMs via a retrieval-augmented generation framework. RAP-Gen first utilizes a hybrid patch retriever to search for a relevant fix pattern from an external codebase, and utilizes Codet5 as the foundation model to synthesize candidate patches with the augmented inputs, i.e., the retrieved fix pattern and original buggy code.

6.1.2 Repair with Few-shot Learning. In few-shot learning, LLMs take a prompt as input , which contains natural language instructions, a few examples of task demonstration, and a query, and generate an output. In 2023, Nashid et al. [108] propose a CEDAR, retrieval-based prompt selection approach for several code generation tasks, including program repair. CEDAR first automatically retrieves bug-fixing demonstrations relevant to the buggy code based on embedding or frequency similarity, and creates prompts to query Codex to generate correct patches in a few-shot learning manner. Meanwhile, Do et al. [27] conduct a preliminary study with ChatGPT and Galpaca-30b (a variant of LLaMA) by using a few-shot example generation pipeline.

6.1.3 Repair with Zero-shot Learning. Unlike the above fine-tuning-based APR work, which relies on high-quality bug-fixing code pairs, some approaches are proposed to directly leverage off-the-shelf LLMs without any training. Such approaches attempt to generate patches in a zero-shot setting and can be categorized into two types, namely cloze-style APR and conversation-style APR.

Cloze-style APR leverages the pre-training objective (e.g., model language modeling) of LLMs to predict correct code tokens with the help of repair patterns. In 2022, Xia et al. [163] introduce AlphaRepair, a cloze-style APR tool to directly query LLMs to generate patches in a zero-shot learning setting. AlphaRepair first replaces the buggy line with a mask line and queries CodeBERT to fill the mask line to generate candidate patches. Furthermore, FitRepair [161] is a more advanced cloze-style APR approach based on CodeT5 and the plastic surgery hypothesis. FitRepair utilizes knowledge-intensified fine-tuning and repair-oriented fine-tuning to help CodeT5 learn the buggy project-specific knowledge and the cloze-style task knowledge, respectively. FitRepair then retrieves relevant identifiers with static analysis, which are fed into fine-tuned CodeT5 to generate candidate

patches. Similarly, Repilot [154] improves the cloze-style APR AlphaRepair with a completion engine. Repilot builds an interaction between LLMs and a completion engine to generate more valid patches by first pruning away infeasible tokens suggested by LLMs and then completing the token based on the suggestions provided by the completion engine. GAMMA [186] further explores the potential of using LLMs to generate patches in a zero-shot learning scenario with a list of well-summarized fix templates. Particularly, GAMMA attempts to address the donor code retrieval issue of traditional template-based APR (e.g., TBar [92]) and regards patch generation as a fill-in-the-blank task by querying LLMs to predict the correct code for masked tokens in a pre-defined fix pattern. Unlike the cloze-style APR, Ribeiro et al. [122] frame the APR problem as a code completion task and apply CodeGPT to fix bugs from ManySStuBs4J [72].

Conversation-style APR leverages the powerful natural language and programming language understanding capabilities of LLMs to generate patches iteratively using test failure information. In 2023, Xia et al. [164, 165] propose ChatRepair, the first conversation-driven APR approach that interleaves patch generation with instant feedback to perform APR in a dialogue manner. ChatRepair constructs prompts with test failure information and queries ChatGPT to generate correct patches from previously incorrect and plausible patches. However, ChatRepair mainly relies on negative feedback (i.e., failure information derived from failing tests) to guide the conversations, which may not always offer specific and adequate prompts for an effective repair. Thus, Kong et al. [75] introduce ContrastRepair, which includes positive feedback from passing tests to supplement the negative feedback. Given a buggy program and a failing test case, ContrastRepair generates a similar passing test case by making minimal modifications to the failing test case. ContrastRepair then constructs a contrastive pair to LLMs, allowing them to better pinpoint the root cause of the bug and generate accurate patches. In the above conversation-style APR scenarios, LLMs take a prompt containing some tokens about buggy code as the input and infer the following tokens about patches as the output. During the conversations, all tokens in the input prompt and output answer consume the computational and financial costs, such as $0.06 per 1k input tokens and $0.03 per 1k generated tokens for GPT-4. To reduce the computational cost of ChatRepair, Hidvegi et al. [52] propose CigaR, a token-efficiency LLM-based APR approach that concentrates on token cost minimization of ChatGPT. CigaR designs three prompts to help ChatGPT minimize the overall token cost with previous responses, including (1) an initiation prompt to initialize the repair process, (2) an improvement prompt to refine partial patches, avoiding discarding potentially valuable patches, and (3) a multiplication prompt builds upon the already generated plausible patches to synthesize more plausible patches with diversity maximization. Unlike previous work relying on a fixed prompt template, RepairAgent [12] treats ChatGPT as an agent capable of autonomously planning and executing actions to generate patches by using dynamic prompts and a state machine to select suitable tools.

6.1.4 Empirical Study. In addition to the above novel techniques, researchers have conducted numerous empirical studies to explore the capabilities of LLMs in fixing semantic bugs. As early as 2021, Mashhadi et al. [100] preliminarily evaluate the performance of fine-tuning CodeBERT for fixing Java bugs from ManySStuBs4J. Lajko et al. [77] empirically fine-tune GPT-2 to automatically generate candidate patches for JavaScript bugs. Prenner et al. [118] investigate CodeX in fixing QuixBugs with a zero-shot setting and Sobania et al. [131] utilizes a more powerful LLM ChatGPT with prompt engineering to generate patches for QuixBugs. In 2023, Horvath et al. [54] explore the impact of model architectures and program representations, involving two popular programming languages, i.e., Java and JavaScript, three different code representations, i.e., raw text, command sequences, and ASTs, and four LLMs, i.e., T5, CodeT5, RoBERTa, and GPTNeo. Meanwhile, Zhao et al. [194] conduct a comprehensive investigation into the effective utilization of LLMs for repairing

code review bugs, involving two LLMs (i.e., ChatGPT and GPT-4) in a zero-shot learning manner, and six LLMs (InCoder, CodeT5+, CodeFuse, LLaMA, CodeGen-2, CodeLLaMA) in a fine-tuning manner. Recently, inspired by the phenomenon that grammatical errors in natural language can be fixed by round-trip translation, i.e., translating sentences to another intermediate language and then back, Ruiz et al. [125] investigate to what extent the RTT pipeline can fix bugs in code in a zero-shot fashion. The comprehensive study involves eight LLMs: PLBART, CodeT5, TransCoder, SantaCoder, InCoder, StarCoderBase, GPT-3.5, and GPT-4, and four APR benchmarks: Defects4J-v1.2, Defects4J-v2.0, QuixBugs, and HumanEval-Java.

Furthermore, more comprehensive empirical studies are published in top-tier SE conferences. Xia et al. [162] conduct an comprehensive study to explore the performance of LLMs in program repair, involving nine LLMs from two categories (i.e., infilling and generative models), and five datasets across three programming languages. They explore three ways to use LLMs for patch generation: complete function generation, correct code infilling, and single-line generation. Meanwhile, Jiang et al. [65] empirically explore the fixing capabilities of ten variants from four LLMs under zero-shot and fine-tuning settings, involving four Java benchmarks. They also construct a new benchmark, HumanEval-Java, which none of the LLMs has seen during training to address the data leakage issue. Huang et al. [61] conduct an empirical study on fixing capabilities of LLMs in the fine-tuning paradigm, involving five LLMs, three programming languages, and three repair scenarios.

6.2 Security Vulnerabilities

A software vulnerability refers to a flaw, bug, or weakness in the design, implementation, operation, or management of software. When exploited, vulnerabilities can lead to unauthorized access, manipulation, or disruption of the software's intended functionality. In 2022, Fu et al. [39] propose VulRepair, an LLM-based automated program repair approach by fine-tuning CodeT5 with vulnerability datasets. Furthermore, in 2023, considering that a vulnerable function only contains a few core elements that need to be fixed, Fu et al. [38] propose VQM, an LLM-based automated vulnerability repair approach that leverages VIT-based approaches for object detection to help CodeT5 focus more on vulnerable code areas during patch generation. Inspired by the relationship between semantic bugs and security vulnerabilities, Zhang et al. [185] propose an enhanced LLM-based vulnerability framework based on transfer learning, demonstrating superior performance against VulRepair. In 2024, Zhou et al. [197] propose VulMaster, a CodeT5-based approach to address several limitations of VulRepair, i.e., (1) handling lengthy vulnerable code by the Fusion-in-Decoder framework; (2) capturing the structure of the vulnerable code by AST; and (3) understanding the expert vulnerability knowledge by the CWE system. De et al. [21] leverage Quantized Low-Rank Adaptation (QLoRA) [86] to fine-tune two advanced 7-billion-parameter LLMs, i.e., CodeLlama [124] and Mistral [103], to fix C vulnerabilities.

Empirical Study. Pearce et al. [115] evaluate the performance of LLMs in repair vulnerabilities in a zero-shot manner, involving five commercially available black-box and two open-source LLMs. They particularly investigate the potential of designing prompts that query LLMs to patch vulnerable code. Furthermore, Zhang et al. [185] conduct an empirical study investigating the performance of fine-tuning LLMs for vulnerability repair, involving five LLMs from three categories, two C vulnerability datasets and more than 100 trained LLMs, which is the largest set to date. They also explore the potential of ChatGPT in repairing security vulnerabilities in a zero-shot manner. Similarly, Wu et al. [159] explore the fixing capabilities of five LLMs in a zero-shot manner and four LLMs in a fine-tuning manner on two real-world Java vulnerability datasets. Recently, Le et al. [78] conduct a preliminary study of ChatGPT and Bard in detecting and fixing security vulnerabilities in JavaScript programs.

6.3 Static Warnings

Static warnings refer to automated alerts generated by static tools that examine code to identify potential errors, inefficiencies, or security vulnerabilities without executing the program. In 2021, Berabi et al. [7] introduce TFix, which is considered a pioneer in incorporating LLMs into APR. Inspired by T5, TFix formulates the program repair problem as a text-to-text task on code sequence, i.e., given a coding error as text, predicting a new text as the patch. TFix is particularly fine-tuned from T5 with a high-quality dataset consisting of 100k bug-fixing pairs across 52 error types detected by a static analyzer, ESLint. Similar to RAP-Gen [151], InferFix [69] employs a retrieval-augmented generation manner to fine-tune LLMs for program repair, i.e., (1) utilizing a retriever to search for semantically equivalent bugs and corresponding fixes from an external codebase, and (2) fine-tuning LLMs on supervised bug-fixing datasets with prompts augmented via adding retrieved similar fixes. However, RAP-Gen targets semantic bugs while InferFix repairs static errors detected by the static analyzer Infer; and RAP-Gen utilizes CodeT5 while InferFix has a significantly larger model Codex with 12 billion parameters. Recently, Wadhwa et al. [144] present CORE to resolve code quality issues flagged by static analysis tools. CORE utilizes ChatGPT as the proposer LLM to generate candidate code revisions and GPT-4 as the ranker to rank the candidate revisions before presenting them to the developer. Alrashedy et al. [5] introduce FDSP to resolve repair security vulnerabilities detected by a static code analysis tool, Bandit [6], in a self-debugging manner.

In addition to the above technical papers, Kim et al. [73] empirically investigate the performance of TFix in fixing errors from industrial Samsung Kotlin projects detected by a static analysis tool SonarQube. Mohajer et al. [104] conduct a more comprehensive study of LLMs in the static code analysis domain, and propose SkipAnalyzer, an LLM-based powered tool to perform three related tasks: detecting bugs, filtering false positive warnings, and patching the detected bugs.

6.4 Syntax Errors

Syntax errors refer to parsing mistakes that occur when code does not follow the rules or grammar of the programming language, such as invalid statements and expressions. These errors are detected by the compiler or interpreter during the parsing phase before the program is executed. Ahmed et al. [3] propose SYNSHINE to fix syntax errors by pre-training and fine-tuning RoBERTa with compiler diagnostics. RustAssistant [22] utilizes GPT-3.5 and GPT-4 to fix Rust compilation errors with prompt engineering. RING [70] is a multilingual syntax error repair approach powered by Codex and few-shot learning and is proven promising in six different languages. PCR-Chain [62] attempts to resolve fully-qualified names and fix last-mile syntax errors based on ChatGPT and Chain-of-Thought Prompting.

6.5 Type Errors

Type errors occur when an operation or function is applied to an object of an inappropriate type. This indicates that the operation expects a certain data type but receives another, thus leading to an exception. These errors are prevalent in dynamically typed languages such as Python and JavaScript, which do not allow operations that are inherently impossible with the given data types. In 2024, Chow et al. [19] construct PyTyDefects, a dataset consisting of 2,766 type error-fix pairs from 176 GitHub repositories. They also introduce PyTy, a T5-based repair technique that fine-tunes the off-the-shelf LLM TFix with PyTyDefects for fixing type errors in Python. Unlike the fine-tuning-based approach PyTy, TypeFix [116] is a prompt-based repair approach for Python type errors in a zero-shot manner. Similar to GAMMA, TypeFix queries LLMs to generate patches by filling the masks in code prompts using fix templates. However, TypeFix distinguishes itself by automatically mining these templates through a hierarchical clustering algorithm, rather than

relying on predefined ones in GAMMA. Furthermore, Ribeiro et al. [123] present Mentat, a type error repair technique for OCaml programs powered by GPT-3. Mentat first analyzes the source code to generate contextually relevant prompts, then exploits GPT-3's advanced language understanding and generation capabilities to produce potential patches.

6.6 Programming Problems

This type of bug refers to incorrect solutions to programming problems from competition platforms, such as LeetCode. In 2022, Zhang et al. [179] propose MMAPR to repair both syntactic and semantic mistakes from introductory Python programming through Codex and few-shot learning. Fan et al. [34] systematically evaluate whether repair techniques can fix the incorrect solutions produced by LLMs in LeetCode contests. Similarly, Liu et al. [93] analyze the correctness of ChatGPT-generated code in Java and Python for 2,033 programming tasks from LeetCode and query ChatGPT to mitigate incorrect code with different prompts. Recently, Zhang et al. [187] empirically evaluate the performance of ChatGPT in fixing incorrect submissions from programming problems. Similar studies are conducted by Haque et al. [51] and Tian et al. [138]. Ishizue et al. [63] combine two LLMs (i.e., ChatGPT and GPT-4) with Refactory [58] to fix programming assignment problems. They leverage LLMs to repair programs with syntax errors and those that Refactory fails to repair. Recently, Zhao et al. [192] explore how LLMs, including ChatGPT, StarCoder, and CodeLlama, perform in repairing student assignments from higher-level programming courses. They construct a student assignment dataset named Defects4DS, which contains 682 submissions from 4 programming assignments, and introduce a repair framework named PaR, which generates patches by selecting peer solutions to create prompts.

6.7 Performance Bugs

Performance bugs refer to inefficient code snippets that do not interfere with functionality but lead to unnecessary time and resource consumption. Garg et al. [41] propose DeepDev-PERF, the first LLM-based approach to repair performance bugs for C# applications by fine-tuning the BART model. Building upon DeepDev-PERF, RAPGen [42] further enhances the repair process through prompt engineering, which demonstrates superior efficiency over traditional intensive fine-tuning methods. RAPGen retrieves relevant instructions from a knowledge base, constructs an input prompt tailored to the specific bug, and then uses Codex to generate the optimized code. This approach is similar to CEDAR [108] yet uniquely leverages prompt engineering to achieve more effective and resource-efficient solutions.

6.8 Hardware Bugs

APR is well-explored in the software domain, while the research is less mature for hardware description languages, such as Verilog. Ahmad et al. [1] empirically explore the potential of LLMs to repair security-relevant bugs in hardware designs. They leverage prompt engineering to query Codex, ChatGPT, GPT-4, and CodeGen for 15 security-related bugs from 10 CWEs. Inspired by the retrieval-augmented generation, Tsai et al. [141] introduce RTLFixer, a framework to fix Verilog syntax errors with LLMs and iterative prompting. RTLFixer formulates an input prompt to query GPT-4 to generate the correct code. RTLFixer then utilizes error logs from the compiler and human guidance from a retrieval database as feedback to perform an interactive debugging loop until all errors are resolved. Yao et al. [173] from Huawei focus on the domain of chip design and propose HDLdebugger, an LLM-assisted hardware description language debugging framework. This framework encompasses data generation through reverse engineering, a search engine to enhance the retrieval-augmented generation, and a fine-tuning approach to train retrieval-augmented LLMs. Similarly, Fu et al. [40] focus on the hardware design iteration process and introduce LLM4SECHW,

an LLM-based hardware debugging framework. LLM4SECHW constructs a hardware debugging-oriented dataset from open-source hardware projects and fine-tunes a suite of hardware domain-specific LLMs capable of automatically reading hardware designs and fixing bugs.

6.9 Smart Contracts

Smart contracts are self-executing contracts on a blockchain network with the terms of the agreement directly written into code. In 2023, Napoli et al. [107] conduct a preliminary study to explore the capabilities of ChatGPT in fixing vulnerable smart contracts with 143 vulnerable Solidity codes. In 2024, Zhang et al. [181] introduce ACFIX, a GPT-based approach to repair access control vulnerabilities in smart contracts based on static code slicing and Chain-of-Thought Prompting. ACFIX mines common AC practices for major categories of code functionality. These practices form a knowledge base that guides LLMs in repairing code with analogous functionalities.

6.10 Misc Repair Types

We summarize other repair scenarios that are covered in fewer than two papers as follows.

Crash Bugs. Crash bugs are considered critical issues as they might cause unexpected program behaviors and termination. Du et al. [30] conduct the first investigation into ChatGPT's capability to resolve real-world crash bugs, particularly code-related and environment-related ones. They design different prompts and propose IntDiagSolver, an interaction approach that involves continuous interaction with LLMs.

API Misuse. Zhang et al. [189] conduct an empirical study to explore the performance of learning-based APR techniques on API misuse repair, involving nine LLMs with millions of parameters under a fine-tuning setting, i.e., CodeBERT, GraphCodeBERT, CodeGPT, PolyCoder-160M, PolyCoder-0.4B, CodeTrans, PLBART, CodeT5 and UniXcoder.

Web UI. Xu et al. [170] conduct the first feasibility study on integrating traditional Web UI test repair techniques with ChatGPT for enhancing Web UI test repair. They first utilize traditional repair approaches to generate a preliminary list of candidate-matched elements and employ ChatGPT to execute a global matching to select the best-matched element further.

Translation Bugs. Code translation refers to the process of converting source code from a source language into another language without changing the original program's functionality or logic. Pan et al. [114] present a large-scale empirical study to investigate the ability of LLMs for code translation across five different languages, including C, C++, Go, Java, and Python. They provide a bug taxonomy of unsuccessful translations from 15 categories and five groups, and design an iterative translation bug repair approach to fix unsuccessful translations from LLMs with a set of heuristics for prompt crafting.

Test Repair. Yaraghi et al. [127] introduce TARGET, an LLM-based approach to repair broken test cases by treating test repair as a language translation task. They first construct TARBENCH, a comprehensive benchmark containing 45,373 broken test repairs across 59 open-source projects. They then fine-tune off-the-shelf LLMs (i.e., PLBART, CodeGen and CodeT5+) to generate correct test code with essential code context.

Motion Planning Algorithm. In the automated vehicles scenario, notion planners refer to algorithms that are responsible for computing a viable path for the vehicle to travel from an initial state to a designated goal region within a predefined time frame.

Lin et al. [90] introduce DrPlanner, the first automated framework that utilizes GPT-4 to diagnose and repair motion planners. DrPlanner first designs a structured prompt to describe the planners using both natural and programming languages. DrPlanner then queries GPT-4 to repair generate algorithms with continuous diagnostic feedback in a closed-loop manner.
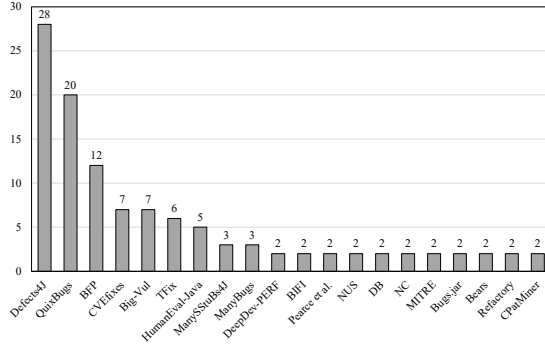
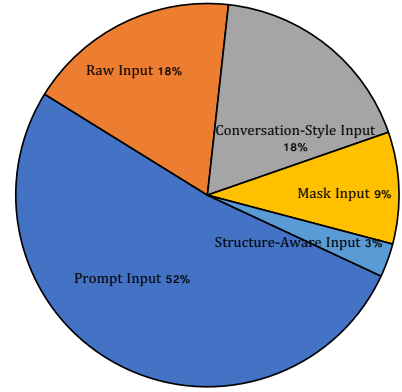Fig. 7. Distribution of publications across datasets.

Fig. 8. Distribution of publications across input forms.

Software Formal Proof. Formal software verification aims to validate the correctness of software properties. First et al. [36] introduce Baldur, an LLM-based approach to automate the generation and repair of whole formal proofs. Baldur leverages two versions of Minerva [81], which is pre-trained on a mathematics corpus based on the PaLM [20]: one with eight billion parameters and another with 62 billion parameters. Baldur first fine-tunes a generation model to synthesize whole proofs for theorems on a proof dataset, and then fine-tunes a repair model based on the proof assistant's error messages to repair incorrectly generated proofs.

Github Issue. Unlike most prior work [16], which evaluates LLMs in fixing self-contained problems, such as programming problems, Jimenez et al. [68] explore the potential of LLMs to resolve GitHub issues in a realistic software engineering setting. Given an issue (such as a bug report or a feature request) submitted to popular GitHub Python repositories, they fine-tune CodeLlama-7B and CodeLlama-13B to generate a patch that passes the unit and system tests.

Code Review Refinement. Guo et al. [48] conduct the first empirical study to explore the potential of ChatGPT in code review, specifically focusing on automated code refinement based on existing code reviews. They leverage prompt engineering to compare ChatGPT with CodeReviewer [87] using two datasets: an established one named CodeReview [87] and a newly introduced one named CodeReview-New. They also design several strategies to improve the performance of ChatGPT, such as using more advanced models.

Summary of Results for RQ3

Overall, we observe that LLMs have been applied in a wide array of repair scenarios in the literature, involving 18 bug types. In some common scenarios dominated by traditional APR, such as semantic bugs, researchers continue to invest substantial efforts in investigating the application of LLMs. Besides, thanks to LLMs' general knowledge learned from all possible Internet data, LLM-based APR has been extended to some rare scenarios that are previously unexplored, such as hardware bugs and Web UI.

7 RQ4: WHAT KEY FACTORS CONTRIBUTE TO THE INTEGRATION OF LLMS FOR APR?

7.1 What sources of datasets are used to evaluate LLM APR studies?

Benchmarking is crucial for evaluating the performance of APR techniques, fundamentally shaping the direction of development within the research community. For example, Defects4J [71] has been the common practice over the last decade, facilitating a standard comparison scenario to understand the strengths and weaknesses of proposed approaches and guiding researchers towards addressing the most pressing challenges in the field. We identify a total of 78 benchmarks from all collected studies and present the benchmarks utilized more than once in Fig. 7.

We find that Defects4J [71] (28 papers) is the most popular benchmark, followed by QuixBugs [89] (20 papers) and BFP [142] (12 papers), all of which are existing benchmarks from previous APR studies. This phenomenon is reasonable, as existing benchmarks are usually well-constructed and have been inspected and utilized by the community, making them the preferred choice for evaluating new APR techniques. For example, a mass of LLM-based APR techniques are evaluated with existing popular benchmarks, such as CIRCLE [176], AlphaRepair [163], and GAMMA [186].

Besides, we notice some newly constructed benchmarks in LLM-based APR, such as TFix [7] (six papers), HumanEval-Java [65] (five papers), and Pearce et al. [115] (two papers). We summarize these new benchmarks into three categories. The first category dataset is tailored for rare repair scenarios. As discussed in Section 6, LLMs have been used in a variety of scenarios, some of which have not been considered by previous work, leading to a gap in relevant benchmarks. As a result, with the advent of LLM-based APR techniques, researchers have also developed corresponding new datasets, such as TFix [7] for static warnings, DeepDev-PERF [41] for performance bugs, Du et al. [30] for crash bugs, and Zhang et. al [189] for API misuse. The second category dataset attempts to address the limitations of previous benchmarks. For example, considering BFP [142] lacking test suites, FixEval [51] offers a collection of unit tests for a large-scale of competitive programming problems and is evaluated with PLBART and CodeT5. The third category of datasets is designed to address the issues unique to LLMs, particularly the data leakage problem. For example, Jiang et al. [65] create a new evaluation benchmark, HumanEval-Java, that has not been seen by LLMs during pre-training. Zhang et al. [187] extensively explore the data leakage issue of ChatGPT in the APR domain and introduce EvalGPTFix, a new benchmark from competitive programming problems after the training cutoff point of ChatGPT. DebugBench [138] is a follow-up of EvalGPTFix with a larger scale and more diverse types of bugs. DebugBench contains 4,253 buggy programs from the LeetCode community, covering four major bug categories and 18 minor types in C++, Java, and Python. Similarly, ConDefects [160] contains 1,254 Java faulty programs and 1,625 Python faulty programs from the online competition platform AtCoder. These collected programs are produced between October 2021 and September 2023 to address the data leakage issue for LLM-based APR approaches. Different from the aforementioned benchmarks derived from programming problems[47], Silva et al. [130] introduce GitBug-Java, a reproducible benchmark comprising 199 recent Java bugs. These bugs are extracted from the 2023 commit history of 55 notable open-source repositories to mitigate the risk of data leakage.

7.2 What input forms are software bugs transformed into when utilizing LLMs?

Thanks to the powerful natural language understanding capabilities of LLMs, the inputs of LLM-based APR contain rich information, thus more complex than that of traditional APR techniques [182]. We summarize various input forms into five categories according to their data types. As illustrated in Fig. 8, we find 52% of collected papers leverage prompt engineering to feed LLMs with bug-fixing information, and 18% utilize a conversational-style representation to provide dynamic information.

We also find only 18% of LLM-based APR studies adopt raw bug-fixing inputs in a manner similar to traditional DL/ML models [182]. We will discuss the five input representations utilized by LLMs as follows.

❶**Raw Bug-fixing Input**. Similar to most traditional learning-based APR, this type of input regards APR as an NMT task, which translates a sentence from one source language (i.e., buggy code) to another target language (i.e., fixed code). Such representation directly feeds LLMs with the buggy code snippet and has been typically employed to train LLMs with supervised learning in semantic bugs [28, 100, 202] security veulnerabilities [39, 185], and static warnings [73]. For example, Zhang et al. [185] investigate the performance of three bug-fixing representations (i.e., context, abstraction, and tokenization) to fine-tune five LLMs for vulnerability repair.

❷**Prompt Input**. This type of input incorporates more information to the buggy code. The prompt concatenates different input components with some prefixed prompt, thus effectively bridging the gap between pre-trained tasks and the APR downstream task. For example, CIRCLE [176] utilizes manually-designed prompt template to convent buggy code and corresponding context into a unified fill-in-the-blank format. Particularly, they utilize "Buggy line:" and "Context:" to denote the buggy and contextual code, and they utilize "The fixed code is:" to query a T5-based model to generate candidate patches according to the previous input. Besides, TFix [7], Zirak et al. [202] and Kim et al. [73] represent all valuable information about the bug as a single piece of text, including bug type, bug message, bug line, and bug context. Furthermore, InferFix [69] and RAP-Gen [151] construct prompts by retrieving relevant repair examples from an external codebase.

❸**Mask Input**. This type of input masks the buggy code and queries LLMs to fill the masks with the correct code tokens. Unlike the above input forms, the mask input reformulates the APR problem as a cloze-style task and directly leverages LLMs' pre-training objectives in a zero-shot setting. AlphaRepair [163] is considered the first work to demonstrate the potential of mask inputs, and researchers have proposed various follow-ups to better perform mask prediction for patch generation, such as GAMMA [186] with well-summarized repair patterns, FitRepair [161] with the plastic surgery hypothesis, Repilot [154] with a completion engine, as well as empirical studies [65, 162].

❹**Conversation-Style Representation** This type of input further extends the prompt input with feedback-driven chats like humans. Conversation-style representation contains more complex information, such as dynamic execution results, while iteratively improving generated patches through multiple rounds of dialogue. For example, Sobania et al. [131] conduct an early exploration into the feasibility of leveraging ChatGPT's conversational capabilities for program repair, motivating some follow-ups [165, 187].

❺**Structure-Aware Input**. This type of input represents source code as syntactic structures, such as Abstract Syntax Trees (ASTs). For example, Horvath et al. [54] utilizes RoBERTa and GPTNeo to encode ASTs for program repair. Besides, VulMaster [197] utilizes the AST as part of its input to capture the structural aspects of the vulnerable code.

7.3 How are LLMs used to support patch correctness?

Test-overfitting has recently been the focus of APR research due to the mainstream test-driven generate-and-validate repair workflow in the community. Particularly, after candidate patches are generated, the developer-written test suites are utilized as the oracle to validate the correctness of patches, and the passing patches are returned to developers. However, as an incomplete specification, test suites only describe a part of the program's behavioral space, resulting in plausible yet overfitting patches without fixing bugs. Thus, developers need to spend enormous effort filtering out overfitting patches manually, even resulting in a negative debugging performance [134, 190]. Researchers have proposed various automated **patch correctness** assessment (APCA) techniques to identify

whether a plausible patch is overfitting or not, so as to improve the quality of returned patches. For example, PATCH-SIM [168] assesses the correctness of patches by calculating the similarity of dynamic execution traces. PATCH-SIM is acknowledged as a foundational work in the APCA field, providing crucial guidance for the development of follow-up works, particularly learning-based or LLM-based ones [182].

Table 5. A summary of existing APR studies using LLMs to predict patch correctness.

Year	Study	LLMs	Repository
2020	Tian et al. [135]	BERT	https://github.com/TruX-DTF/DL4PatchCorrectness
2022	Quatrain [137]	BERT	https://github.com/Trustworthy-Software/Quatrain
2023	Tian et al. [136]	BERT	https://github.com/HaoyeTianCoder/Panther
2023	Invalidator [79]	CodeBERT	https://github.com/thanhlecongg/Invalidator
2023	PatchZero [198]	StarCoder	N.A.
2024	APPT [183]	BERT, CodeBERT, GraphCodeBERT	https://github.com/iSEngLab/APPT

Table 5 presents existing APCA studies involving LLMs. We summarize them into three stages.

❶**LLMs as Feature Extractor.** In 2020, Tian et al. [135, 136] empirically explore the performance of code embeddings via representation learning models in reasoning about patch correctness. Following the similarity-based pipeline from PATCH-SIM, they first calculate the similarities of patched and buggy code snippets based on code embeddings, and then predict patch correctness with a binary classifier. They consider four embedding models, including re-trained (i.e., Doc2vec, code2vec and CC2vec) and pre-trained models (i.e., BERT), which is the first APCA study empowered with LLMs. Recently, Le et al. [79] propose Invalidator, to assess the correctness of patches via semantic and syntactic reasoning. Similar to Tian et al. [135], they utilize CodeBERT to extract code features and train a classifier for prediction. Unlike the above studies calculating similarities of patches, Tian et al. [137] formulate APCA as a question-answering (QA) problem and propose Quatrain. Quatrain first utilizes CodeBERT to encode bug reports and patch descriptions and trains a QA model for prediction.

❷**Fine-tuning LLM-based APCA.** In 2024, Zhang et al. [183] propose APPT, equipped with BERT as the encoder stack, followed by an LSTM stack and a deep learning classifier. Unlike previous studies [135, 136] limiting BERT to extract features without benefiting from training, APPT further fine-tunes LLMs in conjunction with other components as a whole pipeline to fully adapt it specifically for reasoning about patch correctness. APPT is implemented with BERT by default and is also proven generalizable to other advanced LLMs, such as CodeBERT and GraphCodeBERT.

❸**Zero-shot LLM-based APCA.** In 2023, Zhou et al. [198] propose PatchZero to explore the feasibility of LLMs in predicting patch correctness with a zero-shot setting. PatchZero directly queries LLMs to generate the next token about patch correctness (i.e., a token either "correct" or "overfitting") based on previous tokens, which is similar to LLMs' original pre-training objective.

7.4 How are LLMs utilized to facilitate both code generation and program repair?

Compared with traditional APR approaches [182] that usually employ heuristics or neural networks to generate a multitude of patches in one go, LLMs can iteratively refine the generated patches based on the outcomes of dynamic execution. As mentioned in Section 5.2, the iterative patch generation capabilities brought by LLMs have facilitated the emergence of conversational-based repair techniques [22, 107, 115, 116, 118, 131, 187]. In addition to conversational-based repair, we have identified some efforts that improve code generation performance by repairing LLM-generated code with feedback, referred to as self-repair. Different from conversational-based repair, which

utilizes feedback for patch generation, self-repair approaches leverage LLMs to identify code errors implemented by themselves via investigating execution results and explaining the generated code. Such self-repair approaches integrate code generation and program repair, which can be considered a further step toward automated programming.

Table 6. A summary of existing studies that use APR to boost code generation.

Year	Study	LLMs	Repository
2023	AgentCoder [60]	GPT-3.5,GPT-4,Claude,PaLM	N.A.
2023	Self-Edit [180]	InCoder,CodeGen,Codex	https://github.com/zkcpku/Self-Edit
2024	Self-Debugging [17]	ChatGPT,GPT-4,Codex,StarCoder	N.A.
2024	Zheng et al. [195]	CodeLLaMA,DeepseekCoder	https://github.com/OpenCodeInterpreter/OpenCodeInterpreter
2024	CYCLE [26]	CodeGen,StarCoder	https://github.com/ARiSE-Lab/CYCLE_OOPSLA_24
2024	LDB [196]	StarCoder,CodeLLaMA,GPT-3.5	https://github.com/FloridSleeves/LLMDebugger
2024	Olausson et al. [110]	GPT-3.5,GPT-4,CodeLLaMA	https://github.com/theoxo/self-repair
2024	Hu et al. [56]	GPT-4	N.A.

Table 6 presents some LLM-based studies that leverage program repair to boost code generation. Self-Edit [180] represents the first attempt to adopt a neural code editor that takes both the generated code and error messages as inputs to improve the code quality on the competitive programming task. Self-Edit is evaluated with both fine-tuned models (i.e., PyCodeGPT, GPT-Neo, CodeGen, GPT-Neo, InCoder, GPT-J) and prompt-based LLMs (i.e., InCoder, CodeGen, Codex). Chen et al. [17] from DeepMind propose Self-Debugging to teach LLMs to debug their own predicted code via few-shot prompting, including Codex, ChatGPT, GPT-4 and StarCoder. There also exist some similar self-repair studies, including OpenCodeInterpreter [195], Cycle[26], LDB [196], SelfEvolve [67], Self-Refine [97], Hu et al. [56], AgentCoder [60]. Recently, Olausson et al. [110] conduct an empirical study to investigate the ability of CodeLlama, GPT-3.5, and GPT-4 to perform self-repair in code generation. They find that self-repair is not a panacea for code generation challenges, as existing LLMs often fail to provide reliable, accurate, and valuable feedback on why the code is incorrect.

7.5 How often do the collected LLM-based APR papers provide publicly available artifacts?

Open science plays a crucial role in advancing scientific progress through principles of transparency, reproducibility, and applicability. Given the benefits, the SE community has been actively promoting open science principles and encouraging all researchers to share their artifacts, thereby bolstering the reliability of research findings. In this section, we investigate the extent to which the analyzed papers make their artifacts publicly accessible.

We find that 80 studies provide the replication packages in their papers, accounting for 62.99% (80/127) of all collected studies. Among 78 studies that propose novel LLM-based APR approaches, which is the largest contribution type in Table 4, we find that 53.85% (42/78) of them fail to make their artifacts publicly available. This makes it difficult for researchers to validate experimental findings, conduct quantitative comparisons with existing studies, and build follow-ups instead of reinventing the wheels. Considering that some papers have not been published, we then focus on top-tier SE venues, i.e., ICSE, ASE, FSE, ISSTA, TSE and TOSEM, and identify 86.84% of papers (33/38) make related artifacts publicly open, indicating a high-quality commitment to reproducibility in high-quality papers. Besides, some studies only provide datasets or train models without source code or essential instructions. Overall, open science is a critical challenge in advancing LLM-based APR research development because many factors, such as datasets, data pre-processing methods, source code, hyper-parameters, and documents, may lead to the reproducibility of studies. Therefore,

we hope that researchers in the LLM-based APR community can provide high-quality open-source artifacts for convenient reproduction.

Summary of Results for RQ4

(1) We summarize 78 different datasets that are utilized to benchmark LLMs in fixing bugs. (2) Defects4J, QuixBugs, BFP, CVEfixes, and Big-Vul are most frequently adopted in the LLM-based APR. (3) We categorize the input forms within all collected papers into five groups: raw bug-fixing input, prompt input, mask input, conversation-style input, and structure-aware input. (4) Prompt input is the most frequently used form in applying LLMs to program repair, indicating that designing effective prompts is particularly important for leveraging LLMs' natural language processing capabilities. (5) We summarize some studies that leverage LLMs to predict patch correctness. (6) 62.99% of all collected papers have made their artifacts open source, and the ratio increases to 86.84% for top-tier SE publications.

---

8 CHALLENGES & OPPORTUNITIES OF LLM-BASED APR

Our study reveals the following crucial challenges and practical guidelines for future LLM-based APR work.

APR as a Part of Fully Autonomous Programming. As mentioned in Section 2, existing APR approaches operate under a "near-correct assumption", which posits that experienced programmers are capable of writing programs that are almost correct, requiring only minimal modifications to rectify bugs and ensure compliance with all test cases. This assumption has long served as the foundation for APR research development. However, the evolution of LLMs and their application in programming suggests a future where APR can transcend its traditional boundaries, moving towards a more holistic approach in conjunction with fully autonomous programming. In this context, APR can be reimagined not only just as a tool for correcting minor coding errors, but as an integral component of a self-correcting, self-improving system that iteratively enhances the quality of automatically generated code. Fortunately, we have observed initial explorations into combining repair with programming, yet these efforts are far from sufficient. For example, Fan et al. [34] employ LLMs to fix buggy solutions generated by themselves, and recent studies [165, 187] iteratively refine auto-generated code with dynamic execution.

In the future, the opportunities presented by integrating APR with fully autonomous programming are vast. First, it is flexible to implement collaborative Human-AI Programming tools [88], where the initial code is written by developers and continuously optimized and repaired by LLMs. Particularly for complex problem-solving, LLMs suggest innovative solutions that had not been considered by human programmers. This partnership could accelerate development cycles, reduce the burden of debugging on human developers, and lead to more creative and effective solutions. Second, the general knowledge inherent in LLMs endows them with the capability to support multiple downstream tasks, which opens up the possibility of bridging the gap between code generation, testing, debugging, and fixing. For example, fault localization is a precondition of patch generation, while patch validation can reflect the accuracy of fault localization, making these takes interconnected. Thus, it is promising to explore the capabilities of LLMs in these connected tasks using real-time feedback within a unified framework.

More Attention about Repair Costs of LLMs. As mentioned in Section 5.1, the literature tends to employ the growing size of LLMs to achieve better performance, such as from T5-250M in CIRCLE to Codex-12B in InferFix. This trend is reasonable as Xia et al. [162] demonstrate that larger models typically generate more correct patches for software bugs. However, APR,

being a task highly relevant to humans, the millions, billions and even trillions of parameters pose significant challenges in the development workflow. First, training LLMs with billions (or even more) of parameters is a highly time-consuming and resource-intensive process. The GPU resources required for such training are often prohibitively expensive, making them inaccessible for many researchers in both academic and industrial settings. For example, in Section 5.2, we observe that most fine-tuning-LLM-based APR studies utilize CodeT5/T5 or similar-sized models, except for InferFix from Microsoft, which fine-tunes Codex. However, InferFix is proposed by the world-leading technology company Microsoft and trained with industrial-grade hardware, i.e., 64 32-GB-V100-GPUs. Second, despite the increased likelihood of generating correct patches with larger models, the patch generation time cost also increases. For example, Jiang et al. [65] demonstrate PLBART takes 0.70–0.89 seconds on average to generate a correct patch, CodeGen, although fixing more bugs than PLBART with more parameters, requires 3.64–13.88 seconds on average to generate a correct patch. Third, the increase in patch generation time further compresses the time available for patch validation, as developers need to spend more time waiting for the model to complete its inference. For example, Shi et al. [129] demonstrate that the vast storage and runtime memory consumption, coupled with high inference latency, make these LLMs prohibitive for integration in modern IDEs, especially on resource-constrained or real-time terminal devices.

In the future, to address the first challenge, it is promising to explore the potential of parameter-efficient fine-tuning approaches on APR, such as prefix-tuning and low-rank adaptation [25]. To address the second challenge, researchers can optimize the size of LLMs without significantly compromising their performance, such as model pruning, quantization, and knowledge distillation [129]. To address the third challenge, we recommend boosting patch validation with advanced strategies, such as mutation testing [166], or utilizing LLMs to rank candidate patches before validation.

Human Study with LLMs. As summarized in Section 6, with the introduction of LLMs, APR has achieved groundbreaking progress in terms of the number of correctly fixed bugs in popular benchmarks, exemplified by the advancements on Defects4J-1.2 from 64 bugs by CIRCLE, to 74 bugs by AlphaRepair, and 82 bugs by GAMMA. The progress prompts us to consider whether LLMs have indeed facilitated improvements in real-world debugging and how LLM-based APR has broader implications for developers' daily activities. Previous research [157, 188] highlights the potential pitfalls of developing APR tools without adequate feedback from developers, which could compromise their effectiveness in real-world deployment. However, there remains a significant gap in our understanding of how software engineers tackle software problems in practical settings, including their use of dedicated debugging tools and their expertise in debugging techniques.

Thus, in the future, researchers should conduct human studies to gain deeper insights into the maturity and reliability of LLM-based APR tools in terms of human factors. Possible directions are to investigate whether LLMs can assist developers in reducing the debugging cost, such as fixing more bugs, accelerating the bug-fixing process, and handling more complex bugs. Besides, it would be valuable to investigate developers' perceptions and interactions with LLMs based on their practical experiences and established debugging practices.

Exploring More and Rare Repair Scenarios. As summarized in Section 6, we observe that most existing LLM-based APR studies are concentrated on a limited number of bug types, particularly semantic bugs. However, there exist some rare repair scenarios that benefit less from LLMs, such as hardware bugs (one paper) and concurrency bugs (zero paper). The key challenge lies in insufficient training data from which LLMs can learn.

We suggest that future work concentrates on three possible directions to broaden the scope of LLM applications for more repair scenarios, such as software requirement [167] and fuzzing [191]. First, transfer learning is an effective training approach for rare scenarios. We can first fine-tune LLMs with abundant data in a source scenario and then utilize a small amount of data to transfer

the acquired knowledge to a target scenario. The source and target scenarios should have similar data distributions. For example, Zhang et al. [185] demonstrate that transferring learning from bug-fixing can improve the vulnerability repair performance of five LLMs by 9.40% on average. Second, it is promising to utilize the cloze-style APR with repair patterns to generate patches for rare scenarios. Unlike fine-tuning, which requires a substantial amount of labeled data, this approach directly leverages expert domain knowledge (such as pre-defined patterns) to guide the general pre-trained knowledge of LLMs. Besides, for scenarios not previously encountered by LLMs, researchers can employ unlabeled data in the target scenario to learn the data distribution of the project or language under test in an unsupervised learning setting. Third, in-context learning and prompt engineering are feasible solutions for directly querying billion-parameter LLMs to generate correct code in a target scenario, given that these models are immensely large and encompass virtually all data available on the Internet.

Integration with Off-the-Shelf APR. As mentioned in Section 6, researchers typically utilize LLMs as core backbones to design novel repair approaches in an end-to-end setting, such as sequence-to-sequence learning [7, 176]. Parallelly, the community has seen some explorations treating LLMs as components integrated into existing repair workflow. These studies attempt to boost the capabilities of off-the-shelf APR approaches instead of proposing new techniques. For example, CURE [66] combines GPT and CoCoNut [96] to capture code syntax for the APR task. Built on top of DLFix [84], DEAR [85] attempts to fix multi-hunk, multi-statement bugs by fine-tuning BERT to learn fixing-together relationships among statements, i.e., identifying if two statements are needed to be fixed together or not. Recently, GAMMA [186] integrates LLMs into the traditional template-based APR TBar by querying LLMs to generate masked code tokens instead of retrieving donor code from local files. These efforts demonstrate the potential of integrating LLMs with off-the-shelf APR techniques, yet there is currently a lack of more in-depth work in this area.

In the future, researchers could attempt to combine LLMs with more traditional APR techniques. For example, it is promising to utilize LLMs to help SMT solvers generate patches for constraint-based APR, or feed search algorithms with LLM-generated potential patches to build search space for heuristic-based APR. Besides, domain-specific repair techniques can benefit from the powerful code-understanding capabilities of LLMs, thus extending to a broader range of repair scenarios. For example, we can design fix templates for specific scenarios, such as static warnings, and then utilize the general knowledge contained in LLMs to generate correct patches.

Data Leakage Issue. As highlighted in Section 7.1, Zhang et al. [187] identify that existing repair benchmarks have been inadvertently included in the pre-training data of popular LLMs, such as ChatGPT, through web scraping and other methods. For example, ChatGPT is able to enumerate all projects within Defects4J [71], one of the most popular APR benchmarks. Researchers [186] can ascertain the exposure for open-source LLMs by inspecting pre-training data against benchmarks. It is significantly more challenging with more powerful black-box LLMs due to a lack of training details. However, as preliminary explorations, they are mainly limited in the quantity of involved bugs and the variety of bug types. For example, all of them are created from programming problems with only small-scale or medium-scale buggy solutions, without delving into large-scale, real-world projects that encompass complex API calls, such as Defecst4J [71]. More importantly, data leakage issues in other repair scenarios, such as security vulnerabilities and API misuse, continue to be overlooked. There may be overlaps among datasets across different repair scenarios, such as Defects4J, which also serves as a source for a subset of the API misuse dataset [189]. Overall, the risk of data leakage introduces bias into the benchmarking of existing work, necessitating urgent efforts from researchers to mitigate it.

We recommend that future work can be conducted in the following two directions. First, it is crucial to construct a large-scale benchmark free from data leakage that contains real-world

projects so as to evaluate the actual fix capabilities of LLMs in a more practical debugging scenario. Commercial closed-source software, real-time updated programming websites, or manually written programs may serve as potential data sources. Second, considering the variety of bug types that LLMs have been applied to, researchers need to consider and attempt to address the data leakage risk when conducting related studies.

9 CONCLUSION

Automated Program Repair (APR) tackles the long-standing challenge of fixing software bugs automatically, thus facilitating software testing, validation, and debugging practices. Very recently, Large Language Models (LLMs) have brought significant changes to the APR domain, already yielding impressive progress and further demonstrating a promising future in follow-up research. In this paper, we provide a systematic literature review of existing LLM-based APR techniques from LLMs, APR and their integration perspectives We summarize popular LLMs, typical utilization strategies, and repair scenarios. We also discuss some crucial factors, such as input forms and self-debug metrics, within the LLM-based APR community. Finally, we outline several challenges, such as data leakage issues, and suggest potential directions for future research.

REFERENCES

[1] Baleegh Ahmad, Shailja Thakur, Benjamin Tan, Ramesh Karri, and Hammond Pearce. 2024. On Hardware Security Bug Code Fixes By Prompting Large Language Models. *IEEE Transactions on Information Forensics and Security* (2024). Early Access, DOI: 10.1109/TIFS.2024.3374558.

[2] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-Training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, 2655–2668.

[3] Toufique Ahmed, Noah Rose Ledesma, and Premkumar Devanbu. 2022. SynShine: Improved Fixing of Syntax Errors. *IEEE Transactions on Software Engineering* 49, 4 (2022), 2169–2181.

[4] Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. 2023. A3test: Assertion-Augmented Automated Test Case Generation. *arXiv preprint arXiv:2302.10352* (2023).

[5] Kamel Alrashedy and Abdullah Aljasser. 2023. Can LLMs Patch Security Issues? *arXiv preprint arXiv:2312.00024* (2023).

[6] Bandit. 2024. A Static tool to Find Common Security Issues in Python Code. URL: https://github.com/PyCQA/bandit. Lasted accessed: 2024-04-01.

[7] Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. 2021. TFix: Learning to Fix Coding Errors with a Text-to-Text Transformer. In *International Conference on Machine Learning*. PMLR, 780–791.

[8] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-source Software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. 30–39.

[9] Sidney Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. 2022. GPT-Neox-20b: An Open-Source Autoregressive Language Model. In *Proceedings of BigScience Episode# 5–Workshop on Challenges & Perspectives in Creating Large Language Models*. 95–136.

[10] Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. 2021. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow. URL: https://doi.org/10.5281/zenodo.5297715. Lasted accessed: 2024-04-01.

[11] CO Boulder. 2013. Failure to Adopt Reverse Debugging Costs Global Economy $41 Billion Annually. https://totalview.io/press-releases/university-cambridge-study-failure-adopt-reverse-debugging-costs-global-economy-41 Lasted accessed: 2024-04-01.

[12] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. *arXiv preprint arXiv:2403.17134* (2024).

[13] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. 2013. Reversible Debugging Software: Quantify the Time and Cost Saved Using Reversible Debuggers. *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep* 229 (2013).

[14] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language Models Are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, Vol. 33. 1877–1901.

[15] Jialun Cao, Meiziniu Li, Ming Wen, and Shing-chi Cheung. 2023. A Study on Prompt Design, Advantages and Limitations of ChatGPT for Deep Learning Program Repair. *arXiv preprint arXiv:2304.08191* (2023).

[16] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021).

[17] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching Large Language Models to Self-Debug. In *The Twelfth International Conference on Learning Representations.* https://openreview.net/forum?id=KuPixIqPiq

[18] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-To-Sequence Learning for End-To-End Program Repair. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1943–1959.

[19] Yiu Wai Chow, Luca Di Grazia, and Michael Pradel. 2024. PyTy: Repairing Static Type Errors in Python. In *Proceedings of the 46th International Conference on Software Engineering.* 871–871.

[20] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2023. PaLM: Scaling Language Modeling with Pathways. *Journal of Machine Learning Research* 24, 240 (2023), 1–113.

[21] David de Fitero-Dominguez, Eva Garcia-Lopez, Antonio Garcia-Cabot, and Jose-Javier Martinez-Herraiz. 2024. Enhanced Automated Code Vulnerability Repair Using Large Language Models. *arXiv preprint arXiv:2401.03741* (2024).

[22] Pantazis Deligiannis, Akash Lal, Nikita Mehrotra, and Aseem Rastogi. 2023. Fixing Rust Compilation Errors Using Llms. *arXiv preprint arXiv:2308.05177* (2023).

[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT'19).* Association for Computational Linguistics, 4171–4186.

[24] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K Lahiri. 2022. TOGA: A Neural Method for Test Oracle Generation. In *Proceedings of the 44th International Conference on Software Engineering.* ACM, 2130–2141.

[25] Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, et al. 2023. Parameter-Efficient Fine-Tuning of Large-Scale Pre-Trained Language Models. *Nature Machine Intelligence* 5, 3 (2023), 220–235.

[26] Yangruibo Ding, Marcus J Min, Gail Kaiser, and Baishakhi Ray. 2024. CYCLE: Learning to Self-Refine the Code Generation. *arXiv preprint arXiv:2403.18746* (2024).

[27] Tung Do Viet and Konstantin Markov. 2023. Using Large Language Models for Bug Localization and Fixing. In *2023 12th International Conference on Awareness Science and Technology.* IEEE, 192–197.

[28] Dawn Drain, Colin B Clement, Guillermo Serrato, and Neel Sundaresan. 2021. DeepDebug: Fixing Python Bugs Using Stack Traces, Backtranslation, and Code Skeletons. *arXiv preprint arXiv:2105.09352* (2021).

[29] Dawn Drain, Chen Wu, Alexey Svyatkovskiy, and Neel Sundaresan. 2021. Generating Bug-Fixes Using Pretrained Transformers. In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming.* 1–8.

[30] Xueying Du, Mingwei Liu, Juntao Li, Hanlin Wang, Xin Peng, and Yiling Lou. 2023. Resolving Crash Bugs Via Large Language Models: An Empirical Study. *arXiv preprint arXiv:2312.10448* (2023).

[31] Thomas Durieux and Martin Monperrus. 2016. DynaMoth: Dynamic Code Synthesis for Automatic Program Repair. In *Proceedings of the 11th International Workshop on Automation of Software Test.* 85–91.

[32] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. *arXiv preprint arXiv:2310.03533* (2023).

[33] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 508–512.

[34] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated Repair of Programs from Large Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering*. IEEE, 1469–1481.

[35] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics*. 1536–1547.

[36] Emily First, Markus Rabe, Talia Ringer, and Yuriy Brun. 2023. 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*. 1229–1241.

[37] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=hQwb-lbM6EL

[38] Michael Fu, Van Nguyen, Chakkrit Tantithamthavorn, Dinh Phung, and Trung Le. 2023. Vision Transformer-Inspired Automated Vulnerability Repair. *ACM Transactions on Software Engineering and Methodology* (2023).

[39] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Phung Dinh. 2022. VulRepair: A T5-Based Automated Software Vulnerability Repair. In *the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 935–947.

[40] Weimin Fu, Kaichen Yang, Raj Gautam Dutta, Xiaolong Guo, and Gang Qu. 2023. LLM4SecHW: Leveraging domain-specific large language model for hardware debugging. In *2023 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. IEEE, 1–6.

[41] Spandan Garg, Roshanak Zilouchian Moghaddam, Colin B Clement, Neel Sundaresan, and Chen Wu. 2022. DeepDev-PERF: A Deep Learning-Based Approach for Improving Software Performance. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 948–958.

[42] Spandan Garg, Roshanak Zilouchian Moghaddam, and Neel Sundaresan. 2023. Rapgen: An Approach for Fixing Code Inefficiencies in Zero-Shot. *arXiv preprint arXiv:2306.17077* (2023).

[43] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 45, 1 (2019), 34–67.

[44] Haotong Ge and Yuemeng Wu. 2023. An Empirical Study of Adoption of ChatGPT for Bug Fixing among Professional Developers. *Innovation & Technology Advances* 1, 1 (Jun. 2023), 21–29. https://doi.org/10.61187/ita.v1i1.19

[45] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 7212–7225.

[46] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *Proceedings of the 9th International Conference on Learning Representations*. 1–18.

[47] Jiawei Guo, Ziming Li, Xueling Liu, Kaijing Ma, Tianyu Zheng, Zhouliang Yu, Ding Pan, Yizhi Li, Ruibo Liu, Yue Wang, et al. 2024. CodeEditorBench: Evaluating Code Editing Capability of Large Language Models. *arXiv preprint arXiv:2404.03543* (2024).

[48] Qi Guo, Junming Cao, Xiaofei Xie, Shangqing Liu, Xiaohong Li, Bihuan Chen, and Xin Peng. 2023. Exploring the Potential of ChatGPT in Automated Code Refinement: An Empirical Study. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 379–391.

[49] Sichong Hao, Xianjun Shi, and Hongwei Liu. 2024. Exploring the Potential of Pre-Trained Language Models of Code for Automated Program Repair. *Electronics* 13, 7 (2024), 1200.

[50] Sichong Hao, Xianjun Shi, Hongwei Liu, and Yanjun Shu. 2023. Enhancing Code Language Models for Program Repair by Curricular Fine-tuning Framework. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 136–146.

[51] Md Mahim Anjum Haque, Wasi Uddin Ahmad, Ismini Lourentzou, and Chris Brown. 2023. FixEval: Execution-Based Evaluation of Program Fixes for Programming Problems. In *2023 IEEE/ACM International Workshop on Automated Program Repair*. IEEE, 11–18.

[52] Dávid Hidvégi, Khashayar Etemadi, Sofia Bobadilla, and Martin Monperrus. 2024. CigaR: Cost-efficient Program Repair with LLMs. *arXiv preprint arXiv:2402.06598* (2024).

[53] Sepp Hochreiter, J urgen Schmidhuber, and Corso Elvezia. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (1997), 1735–1780.

[54] Dániel Horváth, Viktor Csuvik, Tibor Gyimóthy, and László Vidács. 2023. An Extensive Study on Model Architecture and Program Representation in the Domain of Learning-Based Automated Program Repair. In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*. 31–38.

[55] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large Language Models for Software Engineering: A Systematic Literature Review. *arXiv preprint arXiv:2308.10620* (2023).

[56] Xueyu Hu, Kun Kuang, Jiankai Sun, Hongxia Yang, and Fei Wu. 2024. Leveraging Print Debugging to Improve Code Generation in Large Language Models. *arXiv preprint arXiv:2401.05319* (2024).

[57] Xing Hu, Zhuang Liu, Xin Xia, Zhongxin Liu, Tongtong Xu, and Xiaohu Yang. 2023. Identify and Update Test Cases When Production Code Changes: A Transformer-Based Approach. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1111–1122.

[58] Yang Hu, Umair Z Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. 2019. Re-Factoring Based Program Repair Applied to Programming Assignments. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 388–398.

[59] Yaojie Hu, Xingjian Shi, Qiang Zhou, and Lee Pike. 2022. Fix Bugs with Transformer through a Neural-Symbolic Edit Grammar. In *Deep Learning for Code Workshop*. https://openreview.net/forum?id=SBgE6i_WkZq

[60] Dong Huang, Qingwen Bu, Jie M Zhang, Michael Luck, and Heming Cui. 2023. AgentCoder: Multi-Agent-based Code Generation with Iterative Testing and Optimisation. *arXiv preprint arXiv:2312.13010* (2023).

[61] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An Empirical Study on Fine-Tuning Large Language Models of Code for Automated Program Repair. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 1162–1174.

[62] Qing Huang, Jiahui Zhu, Zhenchang Xing, Huan Jin, Changjing Wang, and Xiwei Xu. 2023. A Chain of AI-Based Solutions for Resolving Fqns and Fixing Syntax Errors in Partial Code. *arXiv preprint arXiv:2306.11981* (2023).

[63] Ryosuke Ishizue, Kazunori Sakamoto, Hironori Washizaki, and Yoshiaki Fukazawa. 2024. Improved Program Repair Methods using Refactoring with GPT Models. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*. 569–575.

[64] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 298–309.

[65] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *Proceedings of the 45th International Conference on Software Engineering*. 1430–1442.

[66] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering*. 1161–1173.

[67] Shuyang Jiang, Yuhao Wang, and Yu Wang. 2023. SelfEvolve: A Code Evolution Framework via Large Language Models. *arXiv preprint arXiv:2306.02907* (2023).

[68] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=VTF8yNQM66

[69] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. InferFix: End-to-End Program Repair with LLMs. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 1646–1656.

[70] Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. 2023. Repair Is Nearly Generation: Multilingual Program Repair with LLMS. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 5131–5140.

[71] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 23rd International Symposium on Software Testing and Analysis*. 437–440.

[72] Rafael-Michael Karampatsis and Charles Sutton. 2020. How Often Do Single-Statement Bugs Occur? The Manysstubs4j Dataset. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR'20)*. 573–577.

[73] Misoo Kim, Youngkyoung Kim, Hohyeon Jeong, Jinseok Heo, Sungoh Kim, Hyunhee Chung, and Eunseok Lee. 2022. An Empirical Study of Deep Transfer Learning-Based Program Repair for Kotlin Projects. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1441–1452.

[74] Barbara Ann Kitchenham and Stuart Charters. 2007. *Guidelines for Performing Systematic Literature Reviews in Software Engineering*. Technical Report EBSE 2007-001. Keele University and Durham University Joint Report. 1–65

pages.

[75] Jiaolong Kong, Mingfei Cheng, Xiaofei Xie, Shangqing Liu, Xiaoning Du, and Qi Guo. 2024. ContrastRepair: Enhancing Conversation-Based Automated Program Repair Via Contrastive Test Case Pairs. *arXiv preprint arXiv:2403.01971* (2024).

[76] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. FixMiner: Mining Relevant Fix Patterns for Automated Program Repair. *Empirical Software Engineering* 25, 3 (2020), 1980–2024.

[77] Márk Lajkó, Viktor Csuvik, and László Vidács. 2022. Towards Javascript Program Repair with Generative Pre-trained Transformer. In *2022 IEEE/ACM International Workshop on Automated Program Repair*. IEEE, 61–68.

[78] Tan Khang Le, Saba Alimadadi, and Steven Y Ko. 2024. A Study of Vulnerability Repair in JavaScript Programs with Large Language Models. *arXiv preprint arXiv:2403.13193* (2024).

[79] Thanh Le-Cong, Duc-Minh Luong, Xuan Bach D Le, David Lo, Nhat-Hoa Tran, Bui Quang-Huy, and Quyet-Thang Huynh. 2023. Invalidator: Automated Patch Correctness Assessment Via Semantic and Syntactic Reasoning. *IEEE Transactions on Software Engineering* 49, 06 (2023), 3411–3429.

[80] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 01 (2012), 54–72.

[81] Aitor Lewkowycz, Anders Johan Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Venkatesh Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. 2022. Solving Quantitative Reasoning Problems with Language Models. In *Advances in Neural Information Processing Systems*. https://openreview.net/forum?id=IFXTZERXdM7

[82] Jia Li, Yongmin Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. 2023. SkCoder: A Sketch-based Approach for Automatic Code Generation. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2124–2135.

[83] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: May the Source be With You! *arXiv preprint arXiv:2305.06161* (2023).

[84] Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. DLFix: Context-based Code Transformation Learning for Automated Program Repair. In *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering*. 602–614.

[85] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022. DEAR: A Novel Deep Learning-based Approach for Automated Program Repair. In *Proceedings of the 44th International Conference on Software Engineering*. 511–523.

[86] Yixiao Li, Yifan Yu, Chen Liang, Nikos Karampatziakis, Pengcheng He, Weizhu Chen, and Tuo Zhao. 2024. LoftQ: LoRA-Fine-Tuning-aware Quantization for Large Language Models. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=LzPWWPAdY4

[87] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. 2022. 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*. 1035–1047.

[88] Jingjing Liang, Ruyi Ji, Jiajun Jiang, Shurui Zhou, Yiling Lou, Yingfei Xiong, and Gang Huang. 2021. Interactive Patch Filtering as Debugging Aid. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 239–250.

[89] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A Multi-Lingual Program Repair Benchmark Set Based on the Quixey Challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion'17)*. 55–56.

[90] Yuanfei Lin, Chenran Li, Mingyu Ding, Masayoshi Tomizuka, Wei Zhan, and Matthias Althoff. 2024. DrPlanner: Diagnosis and Repair of Motion Planners Using Large Language Models. *arXiv preprint arXiv:2403.07470* (2024).

[91] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. Avatar: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 1–12.

[92]   Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. TBar: Revisiting Template-based Automated
       Program Repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*.
       31–42.
[93]   Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D. Le, and David Lo.
       2024. Refining ChatGPT-Generated Code: Characterizing and Mitigating Code Quality Issues. *ACM Transactions on
       Software Engineering and Methodology* (2024). https://doi.org/10.1145/3643674 Just Accepted.
[94]   Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang,
       Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. StarCoder 2 and The Stack v2: The Next Generation. *arXiv
       preprint arXiv:2402.19173* (2024).
[95]   Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain,
       Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou,
       Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning
       Benchmark Dataset for Code Understanding and Generation. In *Proceedings of the Neural Information Processing
       Systems Track on Datasets and Benchmarks 1*.
[96]   Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNut: Combining
       Context-Aware Neural Translation Models Using Ensemble for Program Repair. In *Proceedings of the 29th ACM
       SIGSOFT International Symposium on Software Testing and Analysis*. 101–114.
[97]   Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri,
       Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean
       Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative Refinement with Self-Feedback. In *Thirty-
       seventh Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=S37hOerQLB
[98]   Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java. In *Proceedings of the 25th
       International Symposium on Software Testing and Analysis*. 441–444.
[99]   Matias Martinez and Martin Monperrus. 2018. Ultra-Large Repair Search Space with Automatically Mined Templates:
       The Cardumen Mode of Astor. In *Proceedings of the International Symposium on Search Based Software Engineering
       (SSBSE'18)*. Springer, 65–86.
[100] Ehsan Mashhadi and Hadi Hemmati. 2021. Applying Codebert for Automated Program Repair of Java Simple Bugs.
       In *Proceedings Companion of the 18th IEEE/ACM International Conference on Mining Software Repositories (MSR'21)*.
       505–509.
[101] Fairuz Nawer Meem, Justin Smith, and Brittany Johnson. 2024. Exploring Experiences with Automated Program
       Repair in Practice. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer
       Society, 870–870.
[102] Meta. 2024. LlaMa. URL: https://github.com/meta-llama/llama3. Lasted accessed: 2024-04-01.
[103] Mistral. 2024. Mistral-7B. URL: https://mistral.ai/news/announcing-mistral-7b/. Lasted accessed: 2024-04-01.
[104] Mohammad Mahdi Mohajer, Reem Aleithan, Nima Shiri Harzevili, Moshi Wei, Alvine Boaye Belle, Hung Viet Pham,
       and Song Wang. 2023. SkipanAlyzer: An Embodied Agent for Code Analysis with Large Language Models. *arXiv
       preprint arXiv:2310.18532* (2023).
[105] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *Comput. Surveys* 51, 1 (2018), 1–24.
[106] Martin Monperrus. 2022. *The Living Review on Automated Program Repair*. Technical Report hal-01956501.
       HAL/archives-ouvertes.fr.
[107] Emanuele Antonio Napoli and Valentina Gatteschi. 2023. Evaluating ChatGPT for Smart Contracts Vulnerability
       Correction. In *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 1828–1833.
[108] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-Based Prompt Selection for Code-Related Few-Shot
       Learning. In *Proceedings of the 45th International Conference on Software Engineering*. IEEE, 2450–2462.
[109] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong.
       2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *The Eleventh
       International Conference on Learning Representations*. https://openreview.net/forum?id=iaYcJKpY2B_
[110] Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2024. Is Self-
       Repair a Silver Bullet for Code Generation?. In *The Twelfth International Conference on Learning Representations*.
       https://openreview.net/forum?id=y0GJXRungR
[111] OpenAI. 2022. GPT-3.5. URL: https://platform.openai.com/docs/models/gpt-3-5. Lasted accessed: 2024-04-01.
[112] OpenAI. 2023. ChatGPT. URL: https://openai.com/blog/ChatGPT. Lasted accessed: 2024-04-01.
[113] OpenAI. 2023. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774* (2023).
[114] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris
       Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in Translation: A Study of Bugs
       Introduced by Large Language Models While Translating Code. In *2024 IEEE/ACM 46th International Conference on
       Software Engineering*. IEEE Computer Society, 866–866.

[115] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining Zero-Shot Vulnerability Repair with Large Language Models. In *2023 IEEE Symposium on Security and Privacy*. IEEE, 2339–2356.

[116] Yun Peng, Shuzheng Gao, Cuiyun Gao, Yintong Huo, and Michael Lyu. 2024. Domain Knowledge Matters: Improving Prompts with Fix Templates for Repairing Python Type Errors. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[117] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. 2015. Guidelines for Conducting Systematic Mapping Studies in Software Engineering: An Update. *Information and Software Technology* 64 (2015), 1–18.

[118] Julian Aron Prenner, Hlib Babii, and Romain Robbes. 2022. Can OpenAI's Codex Fix Bugs? An Evaluation on QuixBugs. In *Proceedings of the Third International Workshop on Automated Program Repair*. 69–75.

[119] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. GPT-1: Improving Language Understanding by Generative Pre-Training. URL: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf. Lasted accessed: 2024-04-01.

[120] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language Models Are Unsupervised Multitask Learners. *OpenAI blog* 1, 8 (2019), 9.

[121] Noam Raffel, Colinand Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *The Journal of Machine Learning Research* 21, 1 (2020), 5485–5551.

[122] Francisco Ribeiro, Rui Abreu, and João Saraiva. 2022. Framing Program Repair as Code Completion. In *Proceedings of the Third International Workshop on Automated Program Repair*. IEEE, 38–45.

[123] Francisco Ribeiro, José Nuno Castro de Macedo, Kanae Tsushima, Rui Abreu, and João Saraiva. 2023. GPT-3-Powered Type Error Debugging: Investigating the Use of Large Language Models for Code Repair. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering*. 111–124.

[124] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. *arXiv preprint arXiv:2308.12950* (2023).

[125] Fernando Vallecillos Ruiz, Anastasiia Grishina, Max Hort, and Leon Moonen. 2024. A Novel Approach for Automatic Program Repair Using Round-Trip Translation with Large Language Models. *arXiv preprint arXiv:2401.07994* (2024).

[126] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning Representations by Back-Propagating Errors. *Nature* 323, 6088 (1986), 533–536.

[127] Ahmadreza Saboor Yaraghi, Darren Holden, Nafiseh Kahani, and Lionel Briand. 2024. Automated Test Case Repair Using Language Models. *arXiv e-prints* (2024), arXiv–2401.

[128] Max Schafer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 1 (2024), 85–105.

[129] Jieke Shi, Zhou Yang, Bowen Xu, Hong Jin Kang, and David Lo. 2022. Compressing Pre-Trained Models of Code into 3 Mb. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[130] André Silva, Nuno Saavedra, and Martin Monperrus. 2024. GitBug-Java: A Reproducible Benchmark of Recent Java Bugs. *arXiv preprint arXiv:2402.02961* (2024).

[131] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An Analysis of the Automatic Bug Fixing Performance of ChatGPT. In *2023 IEEE/ACM International Workshop on Automated Program Repair*. 23–30.

[132] Chia-Yi Su and Collin McMillan. 2024. Distilled GPT for Source Code Summarization. *Automated Software Engineering* 31, 1 (2024), 22.

[133] Weisong Sun, Chunrong Fang, Yudu You, Yun Miao, Yi Liu, Yuekang Li, Gelei Deng, Shenghan Huang, Yuchen Chen, Quanjun Zhang, et al. 2023. Automatic Code Summarization Via ChatGPT: How Far Are We? *arXiv preprint arXiv:2305.12865* (2023).

[134] Yida Tao, Jindae Kim, Sunghun Kim, and Chang Xu. 2014. Automatically Generated Patches As Debugging Aids: A Human Study. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 64–74.

[135] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F Bissyandé. 2020. 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*. 981–992.

[136] Haoye Tian, Kui Liu, Yinghua Li, Abdoul Kader Kaboré, Anil Koyuncu, Andrew Habib, Li Li, Junhao Wen, Jacques Klein, and Tegawendé F Bissyandé. 2023. The Best of Both Worlds: Combining Learned Embeddings with Engineered Features for Accurate Prediction of Correct Patches. *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 1–34.

[137] Haoye Tian, Xunzhu Tang, Andrew Habib, Shangwen Wang, Kui Liu, Xin Xia, Jacques Klein, and TegawendÉ F BissyandÉ. 2022. Is This Change the Answer to That Problem? Correlating Descriptions of Bug and Code Changes for Evaluating Patch Correctness. In *37th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 1–13.

[138] Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Zhiyuan Liu, and Maosong Sun. 2024. DebugBench: Evaluating Debugging Capability of Large Language Models. *arXiv preprint arXiv:2401.04621* (2024).

[139] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *arXiv preprint arXiv:2302.13971* (2023).

[140] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open Foundation and Fine-tuned Chat Models. *arXiv preprint arXiv:2307.09288* (2023).

[141] YunDa Tsai, Mingjie Liu, and Haoxing Ren. 2023. RTLFixer: Automatically Fixing RTL Syntax Errors with Large Language Models. *arXiv preprint arXiv:2311.16543* (2023).

[142] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An Empirical Study on Learning Bug-Fixing Patches in the Wild Via Neural Machine Translation. *ACM Transactions on Software Engineering and Methodology* 28, 4 (2019), 1–29.

[143] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *Advances in Neural Information Processing Systems*. 5998–6008.

[144] Nalin Wadhwa, Jui Pradhan, Atharv Sonwane, Surya Prakash Sahu, Nagarajan Natarajan, Aditya Kanade, Suresh Parthasarathy, and Sriram Rajamani. 2023. Frustrated with Code Quality Issues? LLMs Can Help! *arXiv preprint arXiv:2309.12938* (2023).

[145] Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. URL: https://github.com/kingoflolz/mesh-transformer-jax. Lasted accessed: 2024-04-01.

[146] Deze Wang, Boxing Chen, Shanshan Li, Wei Luo, Shaoliang Peng, Wei Dong, and Xiangke Liao. 2023. One Adapter for All Programming Languages? Adapter Tuning for Code Search and Summarization. *arXiv preprint arXiv:2303.15822* (2023).

[147] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software Testing with Large Language Models: Survey, Landscape, and Vision. *IEEE Transactions on Software Engineering* (2024). Early Access, DOI: 10.1109/TSE.2024.3368208.

[148] Shangwen Wang, Mingyang Geng, Bo Lin, Zhensu Sun, Ming Wen, Yepang Liu, Li Li, Tegawendé F Bissyandé, and Xiaoguang Mao. 2023. Natural Language to Code: How Far Are We?. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 375–387.

[149] Simin Wang, Liguo Huang, Amiao Gao, Jidong Ge, Tengfei Zhang, Haitao Feng, Ishna Satyarth, Ming Li, He Zhang, and Vincent Ng. 2022. Machine/Deep Learning for Software Engineering: A Systematic Literature Review. *IEEE Transactions on Software Engineering* (2022).

[150] Shangwen Wang, Bo Lin, Zhensu Sun, Ming Wen, Yepang Liu, Yan Lei, and Xiaoguang Mao. 2023. Two Birds with One Stone: Boosting Code Generation and Code Search via a Generative Adversarial Network. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 486–515.

[151] Weishi Wang, Yue Wang, Shafiq Joty, and Steven CH Hoi. 2023. RAP-Gen: Retrieval-Augmented Patch Generation with CodeT5 for Automatic Program Repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 146–158.

[152] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 8696–8708.

[153] Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran, and Denys Poshyvanyk. 2022. A Systematic Literature Review on the Use of Deep Learning in Software Engineering Research. *ACM Transactions on Software Engineering and Methodology* 31, 2 (2022), 1–58.

[154] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 172–184.

[155] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 364–374.

[156] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. 2007. How Long Will It Take to Fix This Bug?. In *Fourth International Workshop on Mining Software Repositories*. IEEE, 1–1.

[157] Emily Winter, Vesna Nowack, David Bowes, Steve Counsell, Tracy Hall, Sæmundur Haraldsson, and John Woodward. 2022. Let's Talk with Developers, Not about Developers: A Review of Automatic Program Repair Research. *IEEE*

Transactions on Software Engineering 49, 1 (2022), 419–436.

[158] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

[159] Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. 2023. How Effective Are Neural Networks for Fixing Security Vulnerabilities. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*. Association for Computing Machinery, 1282–1294.

[160] Yonghao Wu, Zheng Li, Jie M Zhang, and Yong Liu. 2023. ConDefects: A New Dataset to Address the Data Leakage Concern for LLM-based Fault Localization and Program Repair. *arXiv preprint arXiv:2310.16253* (2023).

[161] Chunqiu Steven Xia, Yifeng Ding, and Lingming Zhang. 2023. The Plastic Surgery Hypothesis in the Era of Large Language Models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 522–534.

[162] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering*. IEEE Computer Society, 1482–1494.

[163] Chunqiu Steven Xia and Lingming Zhang. 2022. 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*. 959–971.

[164] Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational Automated Program Repair. *arXiv preprint arXiv:2301.13246* (2023).

[165] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 Out of 337 Bugs for $0.42 Each Using ChatGPT. *arXiv preprint arXiv:2304.00385* (2023).

[166] Yuan-An Xiao, Chenyang Yang, Bo Wang, and Yingfei Xiong. 2024. Accelerating Patch Validation for Program Repair with Interception-Based Execution Scheduling. *IEEE Transactions on Software Engineering* 01 (2024), 1–18.

[167] Danning Xie, Byungwoo Yoo, Nan Jiang, Mijung Kim, Lin Tan, Xiangyu Zhang, and Judy S Lee. 2023. Impact of Large Language Models on Generating Software Specifications. *arXiv preprint arXiv:2306.03324* (2023).

[168] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying Patch Correctness in Test-Based Program Repair. In *Proceedings of the 40th IEEE/ACM International Conference on Software Engineering*. 789–799.

[169] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering*. IEEE, 416–426.

[170] Zhuolin Xu, Yuanzhang Lin, Qiushi Li, and Shin Hwei Tan. 2023. Guiding ChatGPT to Fix Web Ui Tests Via Explanation-Consistency Checking. *arXiv preprint arXiv:2312.05778* (2023).

[171] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* 43, 1 (2016), 34–55.

[172] Yanming Yang, Xin Xia, David Lo, and John Grundy. 2022. A Survey on Deep Learning for Software Engineering. *Comput. Surveys* 54, 10s (2022), 1–73.

[173] Xufeng Yao, Haoyang Li, Tsz Ho Chan, Wenyi Xiao, Mingxuan Yuan, Yu Huang, Lei Chen, and Bei Yu. 2024. HDLdebugger: Streamlining HDL debugging with Large Language Models. *arXiv preprint arXiv:2403.11671* (2024).

[174] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2022. SelfAPR: Self-Supervised Program Repair with Test Execution Diagnostics. In *2022 37th IEEE/ACM International Conference on Automated Software Engineering*. IEEE.

[175] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural Program Repair with Execution-Based Backpropagation. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering*. 1506–1518.

[176] Wei Yuan, Quanjun Zhang, Tieke He, Chunrong Fang, Nguyen Quoc Viet Hung, Xiaodong Hao, and Hongzhi Yin. 2022. CIRCLE: Continual Repair across Programming Languages. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 678–690.

[177] Yuan Yuan and Wolfgang Banzhaf. 2018. ARJA: Automated Repair of Java Programs Via Multi-objective Genetic Programming. *IEEE Transactions on Software Engineering* 46, 10 (2018), 1040–1067.

[178] He Zhang, Muhammad Ali Babar, and Paolo Tell. 2011. Identifying Relevant Studies in Software Engineering. *Information and Software Technology* 53, 6 (2011), 625–637.

[179] Jialu Zhang, José Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. 2022. Repairing Bugs in Python Assignments Using Large Language Models. *arXiv preprint arXiv:2209.14876* (2022).

[180] Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-Edit: Fault-Aware Code Editor for Code Generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 769–787.

[181] Lyuye Zhang, Kaixuan Li, Kairan Sun, Daoyuan Wu, Ye Liu, Haoye Tian, and Yang Liu. 2024. ACFIX: Guiding LLMs with Mined Common RBAC Practices for Context-Aware Repair of Access Control Vulnerabilities in Smart Contracts. *arXiv preprint arXiv:2403.06838* (2024).

[182] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2023. A Survey of Learning-Based Automated Program Repair. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–69.

[183] Quanjun Zhang, Chunrong Fang, Weisong Sun, Yan Liu, Tieke He, Xiaodong Hao, and Zhenyu Chen. 2024. APPT: Boosting Automated Patch Correctness Prediction via Fine-Tuning Pre-Trained Models. *IEEE Transactions on Software Engineering* 50, 03 (2024), 474–494.

[184] Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang, Yun Yang, Weisong Sun, Shengcheng Yu, and Zhenyu Chen. 2023. A Survey on Large Language Models for Software Engineering. *arXiv preprint arXiv:2312.15223* (2023).

[185] Quanjun Zhang, Chunrong Fang, Bowen Yu, Weisong Sun, Tongke Zhang, and Zhenyu Chen. 2023. Pre-Trained Model-Based Automated Software Vulnerability Repair: How Far are We? *IEEE Transactions on Dependable and Secure Computing* (2023). Early Access, DOI: 10.1109/TDSC.2023.3308897.

[186] Quanjun Zhang, Chunrong Fang, Tongke Zhang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2023. GAMMA: Revisiting Template-based Automated Program Repair via Mask Prediction. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 535–547.

[187] Quanjun Zhang, Tongke Zhang, Juan Zhai, Chunrong Fang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2023. A Critical Review of Large Language Model on Software Engineering: An Example from ChatGPT and Automated Program Repair. *arXiv preprint arXiv:2310.08879* (2023).

[188] Quanjun Zhang, Yuan Zhao, Weisong Sun, Chunrong Fang, Ziyuan Wang, and Lingming Zhang. 2022. Program Repair: Automated vs. Manual. *arXiv preprint arXiv:2203.05166* (2022).

[189] Ting Zhang, Ivana Clairine Irsan, Ferdian Thung, David Lo, Asankhaya Sharma, and Lingxiao Jiang. 2023. Evaluating Pre-Trained Language Models for Repairing Api Misuses. *arXiv preprint arXiv:2310.16390* (2023).

[190] Yuntong Zhang, Xiang Gao, Gregory J. Duck, and Abhik Roychoudhury. 2022. Program Vulnerability Repair Via Inductive Inference. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 691–702.

[191] Yuntong Zhang, Ridwan Shariffdeen, Gregory J Duck, Jiaqi Tan, and Abhik Roychoudhury. 2023. Program Repair by Fuzzing over Patch and Input Space. *arXiv preprint arXiv:2308.00666* (2023).

[192] Qianhui Zhao, Fang Liu, Li Zhang, Yang Liu, Zhen Yan, Zhenghao Chen, Yufei Zhou, Jing Jiang, and Ge Li. 2024. Peer-aided Repairer: Empowering Large Language Models to Repair Advanced Student Assignments. *arXiv preprint arXiv:2404.01754* (2024).

[193] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A Survey of Large Language Models. *arXiv preprint arXiv:2303.18223* (2023).

[194] Zelin Zhao, Zhaogui Xu, Jialong Zhu, Peng Di, Yuan Yao, and Xiaoxing Ma. 2023. The Right Prompts for the Job: Repair Code-Review Defects with Large Language Model. *arXiv preprint arXiv:2312.17485* (2023).

[195] Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024. OpenCodeInterpreter: Integrating Code Generation with Execution and Refinement. *arXiv preprint arXiv:2402.14658* (2024).

[196] Li Zhong, Zilong Wang, and Jingbo Shang. 2024. LDB: A Large Language Model Debugger via Verifying Runtime Execution Step-by-step. *arXiv preprint arXiv:2402.16906* (2024).

[197] Xin Zhou, Kisub Kim, Bowen Xu, DongGyun Han, and David Lo. 2024. Out of Sight, Out of Mind: Better Automatic Vulnerability Repair by Broadening Input Ranges and Sources. In *2024 IEEE/ACM 46th International Conference on Software Engineering*. IEEE Computer Society, 872–872.

[198] Xin Zhou, Bowen Xu, Kisub Kim, DongGyun Han, Thanh Le-Cong, Junda He, Bach Le, and David Lo. 2023. PatchZero: Zero-Shot Automatic Patch Correctness Assessment. *arXiv preprint arXiv:2303.00202* (2023).

[199] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A Syntax-Guided Edit Decoder for Neural Program Repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 341–353.

[200] Qihao Zhu, Zeyu Sun, Wenjie Zhang, Yingfei Xiong, and Lu Zhang. 2023. Tare: Type-Aware Neural Program Repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering*. IEEE, 1443–1455.

[201] Yuqi Zhu, Jia Li, Ge Li, YunFei Zhao, Zhi Jin, and Hong Mei. 2024. Hot or Cold? Adaptive Temperature Sampling for Code Generation with Large Language Models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 437–445.

[202] Armin Zirak and Hadi Hemmati. 2024. Improving Automated Program Repair with Domain Adaptation. *ACM Transactions on Software Engineering and Methodology* 33, 3 (2024), 1–43.