No Man is an Island: Towards Fully Automatic Programming by Code Search, Code Generation and Program Repair

QUANJUN ZHANG, CHUNRONG FANG, YE SHANG, TONGKE ZHANG, SHENGCHENG YU, and ZHENYU CHEN, State Key Laboratory for Novel Software Technology, Nanjing University, China

Automatic programming attempts to minimize human intervention in the generation of executable code and has been a long-standing challenge in the software engineering community. To advance automatic programming, researchers are focusing on three primary directions: (1) code search that reuses existing code snippets from external databases; (2) code generation that produces new code snippets from natural language; and (3) program repair that refines existing code snippets by fixing detected bugs. Despite significant advancements, the effectiveness of state-of-the-art techniques is still limited, such as the usability of searched code and the correctness of generated code.

Motivated by the real-world programming process, where developers usually use various external tools to aid their coding processes, such as code search engines and code testing tools, in this work, we propose Cream, an automatic programming framework that leverages recent large language models (LLMs) to integrate the three research areas to address their inherent limitations. Our insight is that the integration of three research areas can overcome their inherent limitations: the code generator can benefit from the valuable information retrieved by the code searcher, while the code repairer can refine the quality of the generated code with external feedback. In particular, our framework first leverages different code search strategies to retrieve similar code snippets, which are then used to further guide the code generation process of LLMs. Our framework further validates the quality of generated code by compilers and test cases, and constructs repair prompts to query LLMs for generating correct patches. We conduct preliminary experiments to demonstrate the potential of our framework, *e.g.,* helping CodeLlama solve 267 programming problems with an improvement of 62.53%. As a generic framework, Cream can integrate various code search, generation, and repair tools, combining these three research areas together for the first time. More importantly, it demonstrates the potential of using traditional SE tools to enhance the usability of LLMs in automatic programming.

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

Additional Key Words and Phrases: Software testing, Machine translation, Metamorphic testing

ACM Reference Format:
Quanjun Zhang, Chunrong Fang, Ye Shang, Tongke Zhang, Shengcheng Yu, and Zhenyu Chen. 2025. No Man is an Island: Towards Fully Automatic Programming by Code Search, Code Generation and Program Repair. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2025), 18 pages. https://doi.org/1

1 Introduction

Software automation has long been a vision of **Software Engineering (SE)**, with one of the significant challenges being the task of automatic programming [23, 29]. Automatic programming

Authors' Contact Information: Quanjun Zhang, quanjun.zhang@smail.nju.edu.cn; Chunrong Fang, fangchunrong@nju.edu.cn; Ye Shang, 201250032@smail.nju.edu.cn; Tongke Zhang, 201250032@smail.nju.edu.cn; Shengcheng Yu, 201250032@smail.nju.edu.cn; Zhenyu Chen, zychen@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China.

ACM 1557-7392/2025/1-ART1
https://doi.org/1

attempts to handle high-level specifications (*e.g.*, natural language and test cases) into correct source code without direct human intervention [36]. It effectively reduces manual coding effort, improves efficiency, and minimizes programming errors, thus enabling a more robust software development pipeline. Besides, it democratizes programming by making it accessible to individuals with varying levels of programming expertise. This is particularly significant as software permeates various industries in modern society, allowing domain-specific experts, who may not be proficient in programming, to undertake programming tasks tailored to their specific needs, such as AI applications in Science [34].

In the SE literature, to advance automatic programming effectively, researchers are focusing on three primary directions:

- **Code Search.** This task [14, 18, 31, 39] involves developing sophisticated algorithms to search and retrieve existing code snippets from vast databases or the internet. Code search is able to accelerate development and promote best practices by enabling the reuse of code.
- **Code Generation.** This task [5, 22, 37, 40–42, 51] explores the automatic creation of code based on high-level specifications or requirements. It harnesses advanced machine learning models and artificial intelligence to translate human intentions into functional programs.
- **Program Repair.** [9, 15, 24, 43, 44] This task involves fixing bugs or vulnerabilities in existing code during the software maintenance phase. Program repair can be seen as automatic code generation at a micro-scale by generating correct code from buggy code.

Over the last decade, considerable research efforts have been devoted to advancing the state-of-the-art in the three areas. Despite being promising, existing research in these domains still suffers from several limitations. First, prior studies [10, 31] find that code snippets retrieved by code search techniques cannot be directly reused and require manual adaptation, consuming a significant amount of time. Second, code generation techniques often struggle to produce syntactically and semantically correct code that can pass both the compiler and test cases. Recent studies [6, 20] show that even the latest **Large Language Models (LLMs)** still tend to generate code that contains errors and vulnerabilities. Third, research on program repair [43, 44] is mostly confined to semantic bugs introduced by developers, while overlooking the rapidly emerging field of auto-generated code [49]. Therefore, addressing the aforementioned issues can help enhance the effectiveness and usability of code search, code generation, and program repair tools when applied in real-world automatic programming scenarios.

To that end, our insights come from the limitations of existing techniques:

- **Limitation of Code Search.** Code search is an effective method for finding usable code from external codebases. However, the retrieved code typically cannot be deployed directly due to several reasons, such as project context, software bugs, and library dependencies [31]. Thus, developers need to adapt retrieved code to specific requirements, including adjusting variable names, optimizing performance or efficiency, fixing bugs or securities, and including necessary dependencies. When applied in practice, although code search tools can significantly accelerate development by providing useful code snippets, developers will often expend considerable effort to customize and validate the retrieved code before integrating it into their projects. To address this issue, a feasible direction is to refine the retrieved code to meet certain requirements automatically. In this regard, **program repair is promising to adapt the code retrieved by code search tools with minor modifications**, as the retrieved code may already be very similar to the desired output.
- **Limitation of Code Generation.** Code generation is the focus of LLMs in the SE community, and has achieved continuous progress. However, LLMs are trained on vast datasets up to a certain cutoff point, making it difficult to acquire and update up-to-date knowledge.

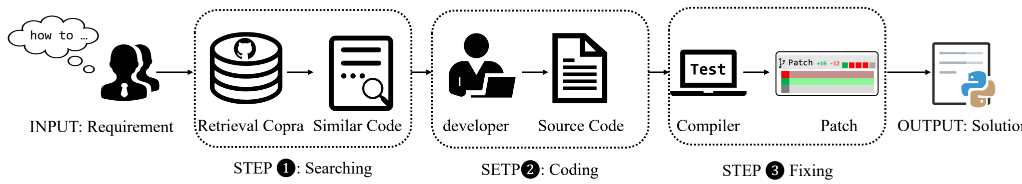

Fig. 1. A common programming scenario during software development

Although fine-tuning remains a possible solution, it is impractical to frequently update LLMs with the latest information due to the vast number of model parameters and computational resources. Thus, when generating code, LLMs may suffer from outdated knowledge and project-specific context. Particularly, LLMs are unaware of new knowledge (such as libraries or frameworks) after the last training update and fail to incorporate information about project-specific requirements, dependencies, or evolving codebases, thus limiting the effectiveness of code generation. To address this issue, a viable approach is to dynamically search for valuable information to augment the code generator. **In this regard, code search can provide useful hints, which can guide LLMs to avoid invalid results during code generation.**

- **Limitation of Program Repair.** As a crucial phase of automatic programming, program repair has achieved significant progress in terms of the number of correctly-fixed bugs [45]. However, existing repair research is mainly limited to fixing bugs detected by functional test cases from well-constructed benchmarks [13, 48]. Recently, LLMs have demonstrated impressive capabilities in automatically generating source code. However, the reliability and quality of auto-generated code are usually imperfect [20], making it difficult to deploy such code directly into projects. In fact, LLMs may generate source code with syntax and semantic errors as the generation process is static without external validation tools, such as compilers. This concern raises a significant question: *can we automatically refine the code generated by LLMs to make it sufficiently trusted for integration into software systems*? Therefore, combining test-driven repair with LLMs can provide dynamic feedback, allowing LLMs to iteratively generate accurate code. **In this regard, program repair is promising to help LLMs perform self-debugging with test-driven feedback during the code generation phase.**

Our analysis motivates us to leverage the complementary strengths of code search, code generation, and program repair techniques to achieve mutual improvement. In real-world programming scenarios, as shown in Fig. 1, developers typically follow a three-step process. First, they construct an appropriate search query (such as natural language descriptions) and use search engines (such as GitHub) to find similar code. Second, they generate their own code by imitating the retrieved code instead of coding from scratch. Third, they validate the generated code through a compiler to ensure it meets specifications, such as test cases. Taking inspiration from the developer practice, we can integrate the three research areas into the programming process, *i.e.,* code search retrieves similar code for code generation, and program repair provides dynamic feedback to refine the generated code. However, the key challenge lies in emulating the human developer role to seamlessly connect the three steps, *i.e.,* how to integrate retrieved code for generation and how to refine generated code based on test feedback. Fortunately, thanks to the powerful natural language and programming language understanding capabilities of LLMs, we can fully automate this process through prompt engineering. Prior studies demonstrate that LLMs can perform code-related tasks in a manner similar to human conversation, and thus, we are motivated to leverage such capabilities of LLMs to connect code search, code generation, and program repair in a unified programming pipeline.

This Work. We propose a framework CREAM, which leverages code sear**C**h, code gene**R**ation, and program rep**A**ir to push forward the boundaries of automatic progr**AM**ming in the ear of LLMs. Our work is motivated by the potential to automatically emulate the common developer practice with the help of LLMs' powerful natural language understanding and programming language generation capabilities. Particularly, given a programming requirement, CREAM follows three steps: (1) code search: an information retrieval (IR) or deep learning (DL)-based technique searches for relevant code from an external database of previous code snippets that may fit the programming context. (2) code generation: an LLM-based code generator synthesizes a ranked list of code candidates based on both the programming requirement and the retrieved external code knowledge. (3) program repair: an LLM-based code repairer slightly refines the token sequence of generated code from the previous step by constructing dynamic prompts with test case feedback. This framework attempts to integrate three well-known research domains that are often developed in isolation, so as to benefit the whole programming pipeline. More importantly, the integration not only broadens the application scope of these three research areas but also boosts the capabilities of recent LLMs in resolving programming problems effectively.

Preliminary Results. We conduct a preliminary experiment to evaluate the effectiveness of CREAM by implementing it with two retrieval strategies as the code searcher and an open-source LLM CodeLLama with 7 billion parameters as the code generator and program repairer. The experimental results on the MBPP benchmark demonstrate that (1) CREAM is able to help CodeLLama in solving programming problems significantly, *e.g.,* generating 267 correct solutions with an improvement of 32.18%; (2) the three phases positively contribute to the performance of CREAM, *e.g.,* code search and program repair improves CodeLlama by 24.75% and 14.85%, respectively. To evaluate the potential of CREAM in a more real-world programming scenario, we illustrate several case studies from the CoderEval benchmark by utilizing GitHub Search API to search for relevant code and a black-box LLM ChatGPT to generate and repair code.

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

- **New Dimension.** We open a new direction for integrating LLMs with traditional SE research areas for more powerful automatic programming. To the best of our knowledge, this is the first work to reveal the potential of LLMs in bridging the gap between three long-standing yet studied-separately research topics, *i.e.,* code search, program repair and program repair.
- **Novel Framework.** We propose CREAM, a three-stage automatic programming framework built on top of LLMs with code search, code generation, and program repair. CREAM is a conceptually generic framework that can be easily integrated with various LLMs, code search, generation and repair tools.
- **Preliminary Evaluation.** We demonstrate CREAM's ability to generate correct solutions for competitive programming problems. Besides, we present case studies to indicate the potential of CREAM in real-world programming scenarios.

2 Background & Related Work

2.1 Large Language Models

In recent years, LLMs have attracted increasing attention from the industry and academia for their impressive capability of processing natural and programming languages [45]. LLMs are generally based on the Transformer architecture that has two key components: the encoder and the decoder. The former encodes the input into a vector representation for the model to understand the input, while the latter transforms the encoded representation to the output sequence. Such LLMs have shown outstanding performance in code-related tasks, such as program repair and code generation [35]. For example, ChatGPT [27] and GPT-4 [26] released by OpenAI are known for their ability of conducting dialogues with human beings. They can take prompts in natural language and generate relevant code accordingly. CodeLlama [30] is a family of open-source LLMs specifically trained for source code and can solve programming problems in a zero-shot situation. Details about the application of LLMs in SE can be found in recent survey papers [35, 45]. However, when programming, such LLMs typically struggle to learn the latest knowledge and interact with external coding tools. In this work, we aim to improve the programming capabilities of off-the-shelf LLMs by integrating them with code search, code generation, and program repair techniques.

2.2 Code Search

A common action that developers take while programming is searching for existing code with similar requirements to reuse. A variety of code search techniques have been proposed to facilitate the retrieval process, and they can be generally classified into two types: IR-based and DL-based [31]. IR-based code search techniques usually involve indexing the codebases and using scoring algorithms to calculate the similarity between the query and the target code. For example, Lucene [8] is a typical IR-based search engine whose default scoring algorithm is BM25, which considers the word frequency and the lengths of documents to rank candidates in the retrieval corpus. DL-based techniques leverage deep learning models to encode code snippets into vectors and retrieve similar code based on the cosine similarity between the vectors. For example, GraphSearchNet [19] is a neural network framework based on bidirectional GGNN to map queries and source code by learning the structural information from them. For a more comprehensive study on code search, please refer to the work of Sun *et al.* [31]. In this work, we implement Cream with two simple yet effective code searchers, *i.e.,* IR-based and DL-based sterategies.

2.3 Code Generation

Code generation is a popular task that LLMs are applied to because of its great potential to improve the coding efficiency of developers. For example, AceCoder [17] retrieves similar code and removes redundant retrieval results to boost the effectiveness of code generation. SkCoder [16] simulates developers' coding behavior by constructing code sketches from the retrieved similar code and turning the sketch into complete code with an encoder-decoder model. CodeAgent [32] proposes a novel repo-level code generation framework that integrates different programming tools, including information retrieval tools, with the purpose of gathering relevant resources so that LLMs can better understand the problems. Please refer to the of Jiang *et al.* [11] for a more comprehensive survey. In this work, we construct prompts augmented by the code searcher to query CodeLlama and ChatGPT as the code generator.

2.4 Program Repair

Program repair aims to automatically fix software bugs, thereby reducing the efforts for manual debugging [45]. Existing repair techniques can be broadly categorized into traditional and learning-based ones. Traditional program repair approaches include heuristic-based, constraint-based, and template-based techniques. With recent advancements in DL, a variety of learning-based repair approaches have been proposed [21, 33, 50]. Such learning-based techniques leverage Neural Machine Translation (NMT) models to understand the semantics of the bugs and transform them into the correct code. For example, CoCoNut [21] utilizes a context-aware NMT architecture to represent the buggy source code and its surrounding context separately.

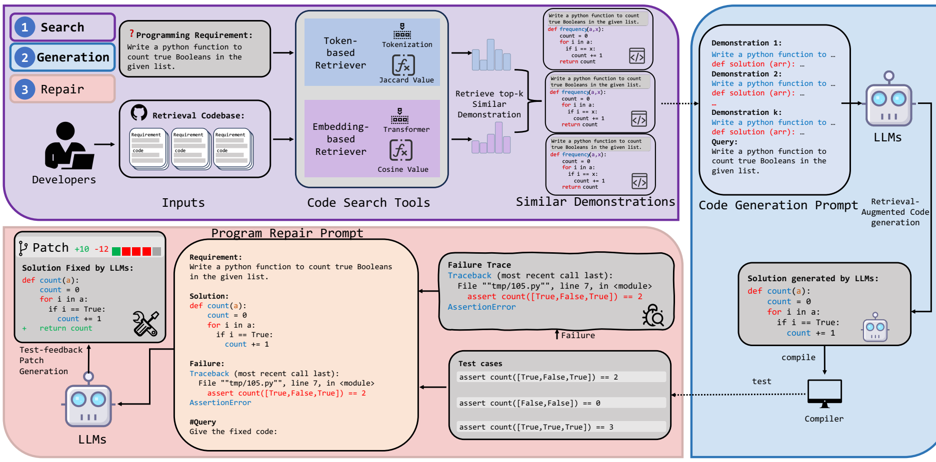

Fig. 2. The overall workflow of this paper

Recently, LLMs are increasingly being utilized for repair tasks [12, 38, 46, 47]. For example, Zhang *et al.* [46] investigate the potential of fine-tuning LLMs in repairing security vulnerabilities. Xia *et al.* [12] evaluate the fixing capabilities of LLMs for Java single-hunk semantic bugs. Detailed summarization of program repair studies can be found in recent work [43, 44]. However, unlike traditional repair techniques, LLMs' powerful natural language capabilities enable them to incorporate external runtime information, thus facilitating iterative patch generation [4, 25]. In this work, motivated by the self-critical capability of LLMs, we leverage execution feedback to integrate program repair into the programming process.

3 Framework and Implementation

3.1 Overview

Cream takes a programming requirement and external codebases as inputs and automatically returns source code that meets the requirements. Fig. 2 illustrates an overview of Cream, which is divided into three phases: code search, code generation, and program repair.

- **In the code search phase**, given the input as a query, Cream searches a database of source code to find similar code applied previously in a similar context. This phase is motivated by the redundancy assumption [3] that the required code can be found or refactored from other projects with similar contexts. In the motivating example in Fig. 2, given the programming requirement "`write a python function to count true Booleans in the given list`" as query, Cream retrieves a code snippet that addresses a similar programming problem from the codebase: "`Write a python function to count true Booleans in the given list`".
- **In the code generation phase**, we first concatenate the retrieved code along with the original programming requirement to construct an augmented input. We then query off-the-shelf LLMs, as mentioned in Section 4, to generate code. Fig. 2 illustrates the generation phase that produces a token sequence for "`def count(a): count = 0 for i in a: if i == True: count += 1)`", which is close to the intended code. However, as LLMs generate code tokens from a probabilistic perspective without the ability to dynamically execute and validate the generated code, the generated code may still be imperfect. For example, in Fig. 2, the generated code lacks a return statement, resulting in output values that do not

match the expected results and failing to pass the test cases. Although the generated code can be quite similar to the intended output, developers still need to spend manual efforts to inspect and modify the generated code.

- **In the program repair phase**, we attempt to capture the minor modifications and further refine the generated code with dynamic test feedback. Our key insight is that, due to the powerful code generation capabilities of LLMs, the code returned in the last phase is already close to perfect, which can naturally be seen as a program repair task. In particular, we utilize the self-critical ability of LLMs by compiling and executing the generated code, then returning the error information to LLMs to guide them in generating more accurate code. In the case of Fig 2, this step explicitly adds the lacked returned statment "`return count`", resulting in the correct patch

3.2 Code Search

In this phase, given a program specification that needs to be implemented as a query, Cream retrieves relevant code snippets from external databases that are applied before in a similar code context. Cream utilizes two types of retrieval strategies: an IR-based retriever and a DL-based retriever, to consider the lexical and semantic similarity, respectively.

3.2.1 IR-based Retriever. Cream employs the sparse strategy IR as the token-based retriever to search for a code snippet that is similar to the query programming requirement based on lexical matching. Suppose $\mathcal{D} = (r_i, c_i)_{i=1}^{|\mathcal{D}|}$ be an external retrieval database consisting of $|\mathcal{D}|$ previous code pairs, where r_i is the i-th program specification and c_i is its corresponding source code. Given a query requirement q, Cream tokenizes all requirements in the retrieval dataset $\mathcal{D}$ and the query and removes duplicate tokens for an efficient retrieval process. Cream then calculates the lexical similarity between q with all requirements in $\mathcal{D}$ based on the Jaccard coefficient. Jaccard is a widely-used similarity coefficient, to measure the similarity between two sparse vector representations based on their overlapping and unique tokens. Formula 1 defines the calculation of Jaccard similarity, where $S(q)$ and $S(r_i)$ are the sets of code tokens of two requirements, q and r_i, respectively. The value varies from 0% to 100%, and a higher value indicates a higher similarity. In this work, considering that all requirements are natural language descriptions, we adopt space to tokenize each requirement instead of a sub-word level tokenizer.

$$Jaccard(q, r_i) = \frac{|S(q) \cap S(r_i)|}{|S(q) \cup S(r_i)|} \tag{1}$$

3.2.2 DL-based Retriever. Cream employs the pre-trained CodeBERT [7] as the embedding-based retriever to search for similar code snippets based on semantic similarity. In particular, Cream first splits all requirements in $\mathcal{D}$ and the query q into a list of tokens and exploits CodeBERT to transform the tokenized tokens into vector representations. CodeBERT Cream prepends a special token of [CLS] into its tokenized sequence and calculates the final layer hidden state of the [CLS] token as the contextual embedding. Cream then calculates the Cosine similarity between the embeddings of two requirements to measure their semantic relevance. Cosine similarity is widely adopted in previous studies to measure the semantic relevance of two dense vectors. Given two vectors, Cosine similarity is calculated based on the cosine of the angle between them, which is the dot product of the vectors divided by the product of their lengths. Formula 2 defines the calculation of Cosine similarity, where $E(q)$ and $E(r_i)$ denote the embeddings of two requirements q and r_i.

$$Cosine(q, r_i) = \frac{E(q) \cdot E(r_i)}{\|E(q)\| \|E(r_i)\|} \tag{2}$$

Algorithm 1 Pseudo Code of Code Search in Cream

Input: q: a query as a program specification that needs to be implemented

Input: $\mathcal{D}$: an external retrieval database for code search

Input: k: the number of similar code snippets to be retrieved

Output: $retrievedResult$: retrieved most similar codes to q' from $\mathcal{D}$

1: $similarityList \leftarrow []$

2: **for** each code requirement r_i in $\mathcal{D}$ **do**

3: $\quad similarityScore \leftarrow$ calculateSimilarity(q, r_i)

4: $\quad similarityList.append(\{requirement : r_i, code : c_i, score : similarityScore\})$

5: **end for**

6: $sortedList \leftarrow$ customSort$(similarityList)$

7: $retrievedResult \leftarrow [c$ for $\{requirement, c, score\}$ in $sortedList[: k]]$

8: **return** $retrievedResult$

9: **function** calculateSimilarity(q, r_i) ▷ Calculate similarity score between q and r_i

10: $\quad \mathcal{T}(q) \leftarrow$ extractToken(q) OR $\mathcal{E}(q') \leftarrow$ extractEmbedding(q')

11: $\quad \mathcal{T}(r_i) \leftarrow$ extractToken(r_i) OR $\mathcal{E}(r_i) \leftarrow$ extractEmbedding(r_i)

12: $\quad similarityScore \leftarrow$ computeJaccardSimilarity$(\mathcal{T}(q), \mathcal{T}(r_i))$ OR $similarityScore \leftarrow$ computeCosineSimilarity$(\mathcal{E}(q), \mathcal{E}(r_i))$

13: $\quad$ **return** $similarityScore$

14: **end function**

15: **function** customSort$(similarityList)$ ▷ Sort list of tuples $similarityList$ in descending order by similarity score

16: $\quad$ **for** $i \leftarrow 0$ to $length(similarityList) - 1$ **do**

17: $\quad\quad$ **for** $j \leftarrow 0$ to $length(similarityList) - i - 1$ **do**

18: $\quad\quad\quad$ **if** $similarityList[j][score] < similarityList[j+1][score]$ **then**

19: $\quad\quad\quad\quad temp \leftarrow similarityList[j]$

20: $\quad\quad\quad\quad similarityList[j] \leftarrow similarityList[j+1]$

21: $\quad\quad\quad\quad similarityList[j+1] \leftarrow temp$

22: $\quad\quad\quad$ **end if**

23: $\quad\quad$ **end for**

24: $\quad$ **end for**

25: $\quad$ **return** $similarityList$

26: **end function**

Algorithm 1 presents the detailed workflow of the search strategy in our work. The algorithm starts by taking three inputs: a query representing the program specification to be implemented (q'), an external database containing code snippets ($\mathcal{D}$), and the number of similar code snippets to be retrieved (top-k). The algorithm initializes an empty list called "similarityList" to store the similarity scores between the query and each code requirement in the database (Line 1). For each code requirement in the database, the algorithm calculates the similarity score between the query and the code requirement using a function named calculateSimilarity (Line 3). This function extracts tokens or embeddings from both the query and the code requirement, then calculates the similarity score using either Jaccard similarity for tokens or cosine similarity for embeddings (Line 9~14). The resulting similarity score, along with the corresponding code requirement and code snippet, is appended to the "similarityList" (Line 4). Once all similarity scores are calculated,

Input

```
# Requirement
Write a python function to check whether a list of numbers contains only one
distinct element or not.

# Test Case
assert unique_Element([1,1,1]) == True
assert unique_Element([1,2,1,2]) == False
assert unique_Element([1,2,3,4,5]) == False

# source code
def unique_Element(arr):
    s = set(arr)
    return len(s) == 1
```

Demonstration 1

Demonstration 2

...

Demonstration *k*

```
# Requirement
#Write a python function to check whether the two numbers differ at one bit
position only or not.

# Test case
assert differ_At_One_Bit_Pos(13,9) == True
assert differ_At_One_Bit_Pos(15,8) == False
assert differ_At_One_Bit_Pos(2,4) == False
assert differ_At_One_Bit_Pos(2, 3) == True
assert differ_At_One_Bit_Pos(5, 1) == True
assert differ_At_One_Bit_Pos(1, 5) == True

# Source code
```

Query

Output

```
def is_Power_Of_Two (x):
    return x and (not(x & (x - 1)))
def differ_At_One_Bit_Pos(a,b):
    return is_Power_Of_Two(a ^ b)",
```

Fig. 3. Retrieval-Augmented Code Generation Prompt Template

the algorithm sorts "similarityList" in descending order based on the similarity scores using a function named customSort (Line 6). This function employs a bubble sort algorithm to ensure the list is ordered from the highest to the lowest similarity score (Line 15~26). After sorting, the algorithm extracts the top-*k* most similar code snippets from the sorted list and stores them in "retrievedResult" (Line 7). Finally, the algorithm returns "retrievedResult" as the output, which contains the code snippets that are most similar to the given query (Line 8). Through this systematic approach, the algorithm effectively retrieves the most relevant code snippets from an external database based on the provided query.

3.3 Code Generation

In the code generation phase, we leverage LLMs to generate code based on the programming requirement and the retrieved programming solutions. Particularly, we build a prompt by composing (a) relevant code demonstrations, (b) the programming query, and (c) natural language instructions. To select demonstrations, we take examples from the demonstration retriever component. Then we select a task-specific template and combine these three elements to build the final prompt.

Fig. 3 illustrates a prompt template for code generation. The input prompt mainly contains two sections: code demonstrations and the query. Each code demonstration section consists of the programming requirement, its test cases, and the expected code. The query section contains the natural language instruction beginning with # in the template, followed by the test cases. As it is an autocomplete task, the comment `Write a python function` is used to signal the model to

Input

```
# Requirement
# Write a function to count the number of characters in a string that occur
at the same position in the string as in the English alphabet (case
insensitive).

def count_char_position(string):
    result = sum(string[i] == chr(i+97) for i in range(len(string)))
    return result

#test cases:
assert count_char_position(\"xbcefg\") == 2,
assert count_char_position(\"ABcED\") == 3,
assert count_char_position(\"AbgdeF\") == 5
```

```
#The above code fails to pass the test cases, with the fail message:
Traceback (most recent call last):
  File ""tmp/165.py"", line 6, in <module>
    assert count_char_position(""ABcED"") == 3
AssertionError

#Give the fixed code:
```

Error Message

Output

```
def count_char_position(string):
    result = sum(string[i].lower() == chr(i+97) for i in range(len(string)))
    return result
```

Reference

```
def count_char_position(str1):
    count_chars = 0
    for i in range(len(str1)):
        if ((i == ord(str1[i]) - ord('A')) or
            (i == ord(str1[i]) - ord('a'))):
            count_chars += 1
    return count_chars ",
```

Fig. 4. Test-Driven Program Repair Prompt Template

generate a correct code snippet. Finally, the expected output for a given prompt is a multi-line code snippet passing the test cases.

3.4 Program Repair

After obtaining generated code snippets by LLMs, this phase attempts to refine them further by fixing syntax and semantic errors. Particularly, we compile the programs and dynamically execute them against all available test cases. The test cases provide valuable information about the correctness of the generated code, associated with the error messages if the code does not pass the tests. For the failing functions, we input them to the LLMs again, together with their requirements and error messages, in an attempt to have them fixed. To this end, we conduct a dynamic prompt for LLMs in the program repair stage, as shown in Fig. 4. The prompt The input prompt contains two sections: the programming requirement. its test cases, the generated code, and the failure information. Each section is denoted by a natural language description that begins with the comment symbol # in the template. The query contains the context of test execution followed by the instruction. This is essentially an auto-complete task where the query is an incomplete example used to prompt the model. Finally, the expected output for a given prompt is a fixed version that addresses the reported failure.

4 Preliminary Results

We conduct two preliminary experiments to evaluate the performance of our search-generation-repair framework for automatic programming. We first investigate the performance of Cream in solving competitive programming problems. We then explore the potential of Cream in solving real-world programming problems.

4.1 Evaluation 1: The Effectiveness of Three Phases in Solving Competitive Programming Problems

Experimental Design. In this RQ, we investigate the effectiveness of the code search, code generation, and program repair phases in our automatic programming framework.

We choose Mostly Basic Programming Problems (MBPP) released by Austin *et al.* [2] as the benchmark. It contains 974 easy Python programming problems, their test cases and code solutions obtained from crowdsourcing. There is also a sanitized version of MBPP with 427 programming problems manually verified and extracted from the full dataset. We focus on the sanitized MBPP and investigate how effective our framework is in solving problems in it. We use the same dataset for code retrieval (except the query) since the coding styles within the same dataset are similar, which helps LLMs better learn what the target code looks like.

We choose CodeLlama as the researched LLM in this RQ and use it to generate Python code for 427 programming problems from the MBPP dataset. CodeLlama is a series of large language models for code generation, and we select the CodeLlama-7b-hf model, which is one of the foundation models with 7B parameters.

We conduct experiments under programming four scenarios to explore how each phase influences the correctness of the generated code:

- **Only code generation.** We construct basic prompts only with programming problem descriptions and test cases, and query LLMs to generate Python functions to solve the problems.
- **Code search + code generation.** When using LLMs to generate code, we not only provide basic information of the programming problems, but also additional functions with similar requirements that are retrieved from the same dataset.
- **Code generation + program repair.** We generate code with the basic prompts, and then we use LLMs again to fix the incorrect code it generates.
- **Code search + code generation + program repair (Cream).** We combine all the phases by retrieving similar code, generating code according to the programming requirements as well as the similar code, and repairing the code that does not pass all the tests.

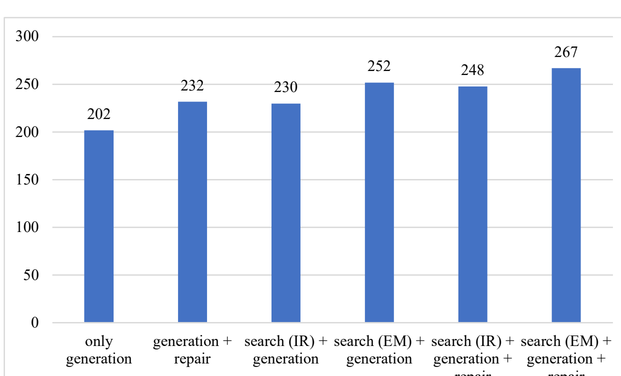

Fig. 5. The number of MBPP problems correctly solved by CodeLlama

Results. Fig. 5 presents the number of problems that CodeLlama successfully solves under four different scenarios. In Fig. 5, "search (IR)" and "search (EM)" denote the IR and embedding retrievers, respectively. When CodeLlama directly performs the code generation task, 202 out of 427 generated solutions are correct, resulting in a correctness rate of 47.31Besides, when integrating program repair, code search-IR, and code search-embedding into the programming process, 31, 28, and 50 additional correct solutions are generated, respectively. Finally, CodeLlama achieves the best performance when code search, code generation and program repair are combined, producing 248 (IR retriever) and 251 (embedding retriever) correct solutions. The improvement against the code generation-only scenario yields 58.08% and 62.53%, respectively. These results demonstrate that the three-phase pipeline—comprising searching, coding, and repairing—continuously enhances the programming capabilities of LLMs.

4.2 Evaluation 2: The Potential of Three Phases in Solving Real-world Programming Problems

Experimental Design. In RQ1, we investigate the performance of Cream in generating functions for competitive programming problems. In this RQ, we further explore how effective our approach is in a real-world programming scenario.

We select CoderEval [41]. Compared to MBPP and other popular code generation benchmarks, which only include standalone functions, CoderEval contains programming tasks extracted from real-world projects as well as separate platforms to execute them, so we can evaluate our automatic programming approach in a real-world scenario.

We simulate real programming scenarios by selecting three functions that need to be implemented from the dataset. First, we use the GitHub search engine to find similar code, then call ChatGPT to generate the code, providing feedback on the test results for corrections. In the following, we present three real-world examples to illustrate the search-generation-repair capabilities of Cream. For all three examples, ChatGPT fails to directly generate correct code based solely on their specifications, *i.e.*, docstring. However, Cream successfully queries ChatGPT to produce the correct code for the first example with code search, the second example with program repair, and the third example with both code search and program repair.

Case 1. Fig. 6 illustrates an example that is not correctly generated by ChatGPT, but correct code can be produced by providing relevant information from code search. This example attempts to identify whether a given string starts with a specified case-insensitive prefix, as shown in the blue part of Fig. 6. ChatGPT first attempts to generate the solution directly, which ignores some boundary conditions and leads to a 'NullPointerException' if either "str" or "prefix" is null. Cream then retrieves a similar solution that provides additional context and correct implementation details, with which ChatGPT properly handles "null" values and checks the length of the strings before performing the comparison, thus avoiding the "NullPointerException". Overall, with the guidance of retrieved code, LLMs produce higher-quality code that adheres to best practices and avoids common pitfalls.

Case 2. Fig. 7 illustrates an example that fails in the generation-only scenario but succeeds after program repair. This example requires completing the logic to call "String.trim()" on each element in a given string array. Similar to the generate-with-retrieval case, CHatGPT overlooks boundary conditions and fails the null-input test case. When provided with the dynamic error message, ChatGPT fixes the bug related to missing boundary checks and passes all test cases.

Case 3. Fig. 8 illustrates an example only generated corrected with code search and program repair. This example involves implementing logic that returns true when the given value is "true", returns false when the given value is "false", and otherwise returns a specified default value. ChatGPT first attempts to generate the solution while ignoring the possibility of a null input, thus

```
# DocString
/**
 * Returns  {@code true} if the given string starts with the specified case-insensitive prefix,
{@code false} otherwise.
 * @param str the String to check
 * @param prefix the prefix to look for
 * @return {@code true} if the given string starts with the specified case-insensitive prefix,
{@code false} otherwise.
 * @see java.lang.String#startsWith
 */

# Signature
public static boolean startsWithIgnoreCase(String str,String prefix){
```

Generation

```
# Generated Code
public static boolean startsWithIgnoreCase(String str, String prefix) {
    return str.toLowerCase().startsWith(prefix.toLowerCase());
}
-----------------------------------------------------------------------------------------------
# Error Message
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.toLowerCase()"
because "<parameter1>" is null
       at io.jsonwebtoken.lang.Strings.startsWithIgnoreCase(Strings.java:291)
       at
io.jsonwebtoken.lang.Strings_startsWithIgnoreCase.main(Strings_startsWithIgnoreCase.java:8)
```

Search + Generation

```
# Reference Code
public static boolean startsWithIgnoreCase(@Nullable String str, @Nullable String prefix) {
    return (str != null && prefix != null && str.length() >= prefix.length() &&
            str.regionMatches(true, 0, prefix, 0, prefix.length()));
}
-----------------------------------------------------------------------------------------------
# Generated Code
public static boolean startsWithIgnoreCase(String str, String prefix) {
    if (str == null || prefix == null) {
        return false;
    }
    if (str.length() < prefix.length()) {
        return false;
    }
    return str.regionMatches(true, 0, prefix, 0, prefix.length());
}
```

Fig. 6. Example-1 generated by ChatGPT

causing a NullPointerException. After retrieving the reference code, ChatGPT identifies the error and corrects the faulty logic. However, it encounters a new issue where it fails to return the correct value in the "false" condition for the same reason. Finally equipped with program repair, ChatGPT generates the correct code and finally passes the test.

5 Discussion and Future Work

The primary technical innovation in this work is the introduction of a unified automatic programming paradigm that leverages advanced LLMs to integrate three long-explored research areas, *i.e.,* code search code generation and program repair. The preliminary experiments highlight the potential of Cream in competitive programming problems and real-world programming scenarios. Particularly, we demonstrate that (1) code search can help code generators produce more accurate code; (2) program repair serves as an effective post-processing step, even after retrieval-augmented code generation; (3) a unified programming pipeline, incorporating the above three phases together, is highly effective in generating code, especially when equipped with LLMs.

As a unified programming pipeline, we believe Cream has significant potential for the SE community and can be extended in the following aspects.

Deployment Scenarios. It is promising to adapt Cream to more programming scenarios during deployment. First, there are some domain-specific areas that require developers to possess both programming skills and domain expert knowledge, such as hardware code [1]. Cream can fully automate such programming process by retrieving similar code within the domain and iteratively refining it, thereby reducing the programming barrier for developers. Second, Cream takes natural

```
public static boolean startsWithIgnoreCase(String str,String prefix){
# DocString
/**
 * Trim the elements of the given String array, calling <code>String.trim()</code> on each of
them.
 * @param array the original String array
 * @return the resulting array (of the same size) with trimmed elements
 */
# Signature
public static String[] trimArrayElements(String[] array){
```

Generation

```
# Generated Code
public static String[] trimArrayElements(String[] array) {
   String[] trimmedArray = new String[array.length];
   for (int i = 0; i < array.length; i++) {
      trimmedArray[i] = array[i].trim();
   }
   return trimmedArray;
}
----------------------------------------------------------------------------------------------
# Error Message
Exception in thread "main" java.lang.NullPointerException: Cannot read the array length because
"<parameter1>" is null
       at io.jsonwebtoken.lang.Strings.trimArrayElements(Strings.java:827)
       at io.jsonwebtoken.lang.Strings_trimArrayElements.main(Strings_trimArrayElements.java:12)
```

Generation + Repair

```
# Repair Code
public static String[] trimArrayElements(String[] array) {
   if (array == null) {
      return new String[0];
   }
   String[] trimmedArray = new String[array.length];
   for (int i = 0; i < array.length; i++) {
      if (array[i] != null) {
         trimmedArray[i] = array[i].trim();
      }
   }
   return trimmedArray;
}
```

Fig. 7. Example-2 generated by ChatGPT

language descriptions as inputs currently, but it can be implemented with other query formats, such as test cases. Considering that Cream treats queries as tokens or embeddings without taking any specific code features into account, Cream can be applied to the other input formats in a drop-in fashion. For example, the potential of Cream in the well-known test-driven development [28] is worth investigating, *i.e.*,, retrieving similar solutions based on test cases. Third, Cream is generic to other code-related tasks, such as test generation, code translation and program repair.

Technical Designs. The effectiveness of Cream can be further optimized by improving the quality of retrieved code and refining the prompt engineering for program repair. First, Cream employs a straightforward retriever that can be either token-based or embedding-based, depending on whether it focuses on syntax or semantics. Given the rapid advancements in code search, optimizing these retrieval strategies for greater efficiency is essential. Second, Cream directly appends error messages to the generated code as prompts and then queries LLMs for repair. In the future, more advanced prompt engineering techniques, such as chain-of-thought, can be utilized.

Evaluation Experiments. In this study, preliminary experiments demonstrate the effectiveness of Cream. We only conduct case studies in Section 4.2 due to the limitation of real-world programming datasets and retrieval corpora. Large-scale evaluations with quantitative analysis are necessary in the future. Besides, the opinions and experiences of developers are crucial for assessing the utility of such programming tools. We plan to conduct user studies to evaluate Cream in real-world programming scenarios. Furthermore, the experiments can be extended in the future with more studied LLMs, benchmarks, and programming languages.

```
# DocString
/**
 * If <code>value</code> is "true", then <code>true</code> is returned. If <code>value</code>
is "false", then <code>false</code> is returned. Otherwise, <code>default</code> is returned.
<p>Case of value is unimportant.
 */
# Signature
public static boolean toBoolean(String value,boolean dEfault){
```

Generation

```
# Generated Code
public static boolean toBoolean(String value, boolean dEfault) {
    if (value.equalsIgnoreCase("true")) {
        return true;
    } else if (value.equalsIgnoreCase("false")) {
        return false;
    } else {
        return dEfault;
    }
}
-----------------------------------------------------------------------------------------------
# Error Message
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"String.equalsIgnoreCase(String)" because "<parameter1>" is null
        at org.apache.log4j.helpers.OptionConverter.toBoolean(OptionConverter.java:129)
        at
org.apache.log4j.helpers.OptionConverter_toBoolean.main(OptionConverter_toBoolean.java:9)
```

Search + Generation

```
# Reference Code
private Boolean toBoolean(String value, Boolean defaultValue) {
    if (StrKit.isBlank(value))
        return defaultValue;
    value = value.trim().toLowerCase();
    if ("1".equals(value) || "true".equals(value))
        return Boolean.TRUE;
    else if ("0".equals(value) || "false".equals(value))
        return Boolean.FALSE;
    throw new ActionException(400, renderManager.getRenderFactory().getErrorRender(400), "Can
not parse the parameter \"" + value + "\" to Boolean value.");
}
-----------------------------------------------------------------------------------------------
# Generated Code
public static boolean toBoolean(String value, boolean dEfault) {
    if (value == null || value.isEmpty()) {
        return dEfault;
    }
    value = value.toLowerCase();
    if ("true".equals(value)) {
        return true;
    } else if ("false".equals(value)) {
        return true;
    } else {
        return dEfault;
    }
}
-----------------------------------------------------------------------------------------------
# Error Message
Test case 2 failed: input='FalsE', expected=false, actual=true
```

Search + Generation + Repair

```
public static boolean toBoolean(String value, boolean dEfault) {
    if (value == null || value.isEmpty()) {
        return dEfault;
    }
    value = value.toLowerCase();
    if ("true".equals(value)) {
        return true;
    } else if ("false".equals(value)) {
        return false;
    } else {
        return dEfault;
    }
}
```

Fig. 8. Example-3 generated by ChatGPT

6 Conclusion

In this paper, we propose a novel automatic programming framework, Cream, which leverages advanced large language models (LLMs) to integrate three well-established areas: code search, code generation, and program repair. Preliminary experiments indicate the potential of our framework to enhance the problem-solving capabilities of existing LLMs in programming tasks. Besides, our framework demonstrates the preliminary benefits of combining LLMs with traditional software engineering (SE) areas. In the future, more advanced technologies, such as intelligent agents, can be employed to further integrate various SE techniques within the programming framework more effectively.

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 Trans. Inf. Forensics Secur.* 19 (2024), 4043–4057. https://doi.org/10.1109/TIFS.2024.3374558

[2] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR* abs/2108.07732 (2021). arXiv:2108.07732 https://arxiv.org/abs/2108.07732

[3] Earl T. Barr, Yuriy Brun, Premkumar T. Devanbu, Mark Harman, and Federica Sarro. 2014. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 306–317. https://doi.org/10.1145/2635868.2635898

[4] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching Large Language Models to Self-Debug. *CoRR* abs/2304.05128 (2023). https://doi.org/10.48550/ARXIV.2304.05128 arXiv:2304.05128

[5] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating Large Language Models in Class-Level Code Generation. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024.* ACM, 81:1–81:13. https://doi.org/10.1145/3597503.3639219

[6] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated Repair of Programs from Large Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023.* IEEE, 1469–1481. https://doi.org/10.1109/ICSE48619.2023.00128

[7] 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: EMNLP 2020, Online Event, 16-20 November 2020 (Findings of ACL, Vol. EMNLP 2020)*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, 1536–1547. https://doi.org/10.18653/V1/2020.FINDINGS-EMNLP.139

[8] Apache Software Foundation. 2024. Apache Lucene. https://lucene.apache.org. Accessed: 2024-07-29.

[9] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Trans. Software Eng.* 45, 1 (2019), 34–67. https://doi.org/10.1109/TSE.2017.2755013

[10] Luca Di Grazia and Michael Pradel. 2023. Code Search: A Survey of Techniques for Finding Code. *ACM Comput. Surv.* 55, 11 (2023), 220:1–220:31. https://doi.org/10.1145/3565971

[11] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A Survey on Large Language Models for Code Generation. *CoRR* abs/2406.00515 (2024). https://doi.org/10.48550/ARXIV.2406.00515 arXiv:2406.00515

[12] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023.* IEEE, 1430–1442. https://doi.org/10.1109/ICSE48619.2023.00125

[13] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, Corina S. Pasareanu and Darko Marinov (Eds.). ACM, 437–440. https://doi.org/10.1145/2610384.2628055

[14] Kisub Kim, Sankalp Ghatpande, Dongsun Kim, Xin Zhou, Kui Liu, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2024. Big Code Search: A Bibliography. *ACM Comput. Surv.* 56, 1 (2024), 25:1–25:49. https://doi.org/10.1145/3604905

[15] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65. https://doi.org/10.1145/3318162

[16] Jia Li, Yongmin Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. 2023. SkCoder: A Sketch-based Approach for Automatic Code Generation. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023.* IEEE, 2124–2135. https://doi.org/10.1109/ICSE48619.2023.00179

[17] Jia Li, Yunfei Zhao, Yongmin Li, Ge Li, and Zhi Jin. 2024. AceCoder: An Effective Prompting Technique Specialized in Code Generation. *ACM Trans. Softw. Eng. Methodol.* (jul 2024). https://doi.org/10.1145/3675395 Just Accepted.

[18] Chao Liu, Xin Xia, David Lo, Cuiyun Gao, Xiaohu Yang, and John C. Grundy. 2022. Opportunities and Challenges in Code Search Tools. *ACM Comput. Surv.* 54, 9 (2022), 196:1–196:40. https://doi.org/10.1145/3480027

[19] Shangqing Liu, Xiaofei Xie, Jing Kai Siow, Lei Ma, Guozhu Meng, and Yang Liu. 2023. GraphSearchNet: Enhancing GNNs via Capturing Global Dependencies for Semantic Code Search. *IEEE Trans. Software Eng.* 49, 4 (2023), 2839–2855. https://doi.org/10.1109/TSE.2022.3233901

[20] Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach Dinh Le, and David Lo. 2024. Refining ChatGPT-Generated Code: Characterizing and Mitigating Code Quality Issues. *ACM Trans. Softw. Eng. Methodol.* 33, 5 (2024), 116:1–116:26. https://doi.org/10.1145/3643674

[21] 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 *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 101–114. https://doi.org/10.1145/3395363.3397369

[22] Antonio Mastropaolo, Luca Pascarella, Emanuela Guglielmi, Matteo Ciniselli, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota. 2023. On the Robustness of Code Generation Techniques: An Empirical Study on GitHub Copilot. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2149–2160. https://doi.org/10.1109/ICSE48619.2023.00181

[23] Hong Mei and Lu Zhang. 2018. Can big data bring a breakthrough for software automation? *Sci. China Inf. Sci.* 61, 5 (2018), 056101:1–056101:3. https://doi.org/10.1007/S11432-017-9355-3

[24] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* 51, 1 (2018), 17:1–17:24. https://doi.org/10.1145/3105906

[25] Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Is Self-Repair a Silver Bullet for Code Generation?. In *The Twelfth International Conference on Learning Representations*.

[26] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023). https://doi.org/10.48550/ARXIV.2303.08774 arXiv:2303.08774

[27] OpenAI. 2024. ChatGPT. URL: https://openai.com/blog/ChatGPT.

[28] Yahya Rafique and Vojislav B. Misic. 2013. The Effects of Test-Driven Development on External Quality and Productivity: A Meta-Analysis. *IEEE Trans. Software Eng.* 39, 6 (2013), 835–856. https://doi.org/10.1109/TSE.2012.28

[29] Abhik Roychoudhury and Yingfei Xiong. 2019. Automated program repair: a step towards software automation. *Sci. China Inf. Sci.* 62, 10 (2019), 200103:1–200103:3. https://doi.org/10.1007/S11432-019-9947-6

[30] 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. *CoRR* abs/2308.12950 (2023). https://doi.org/10.48550/ARXIV.2308.12950 arXiv:2308.12950

[31] Weisong Sun, Chunrong Fang, Yifei Ge, Yuling Hu, Yuchen Chen, Quanjun Zhang, Xiuting Ge, Yang Liu, and Zhenyu Chen. 2024. A Survey of Source Code Search: A 3-Dimensional Perspective. *ACM Trans. Softw. Eng. Methodol.* (apr 2024). https://doi.org/10.1145/3656341 Just Accepted.

[32] Daniel Tang, Zhenghan Chen, Kisub Kim, Yewei Song, Haoye Tian, Saad Ezzini, Yongfeng Huang, Jacques Klein, and Tegawendé F. Bissyandé. 2024. CodeAgent: Collaborative Agents for Software Engineering. *CoRR* abs/2402.02172 (2024). https://doi.org/10.48550/ARXIV.2402.02172 arXiv:2402.02172

[33] 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 Trans. Softw. Eng. Methodol.* 28, 4 (2019), 19:1–19:29. https://doi.org/10.1145/3340544

[34] Hanchen Wang, Tianfan Fu, Yuanqi Du, Wenhao Gao, Kexin Huang, Ziming Liu, Payal Chandak, Shengchao Liu, Peter Van Katwyk, Andreea Deac, Anima Anandkumar, Karianne Bergen, Carla P. Gomes, Shirley Ho, Pushmeet Kohli, Joan Lasenby, Jure Leskovec, Tie-Yan Liu, Arjun Manrai, Debora S. Marks, Bharath Ramsundar, Le Song, Jimeng Sun, Jian Tang, Petar Velickovic, Max Welling, Linfeng Zhang, Connor W. Coley, Yoshua Bengio, and Marinka Zitnik. 2023. Scientific discovery in the age of artificial intelligence. *Nature* 620, 7972 (2023), 47–60. https://doi.org/10.1038/S41586-023-06221-2

[35] 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 Trans. Software Eng.* 50, 4 (2024), 911–936. https://doi.org/10.1109/TSE.2024.3368208

[36] 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, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 375–387. https://doi.org/10.1145/3611643.3616323

[37] Xiaokai Wei, Sujan Kumar Gonugondla, Shiqi Wang, Wasi Uddin Ahmad, Baishakhi Ray, Haifeng Qian, Xiaopeng Li, Varun Kumar, Zijian Wang, Yuchen Tian, Qing Sun, Ben Athiwaratkun, Mingyue Shang, Murali Krishna Ramanathan,

Parminder Bhatia, and Bing Xiang. 2023. Towards Greener Yet Powerful Code Generation via Quantization: An Empirical Study. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 224–236. https://doi.org/10.1145/3611643.3616302

[38] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1482–1494. https://doi.org/10.1109/ICSE48619.2023.00129

[39] Yutao Xie, Jiayi Lin, Hande Dong, Lei Zhang, and Zhonghai Wu. 2024. Survey of Code Search Based on Deep Learning. *ACM Trans. Softw. Eng. Methodol.* 33, 2 (2024), 54:1–54:42. https://doi.org/10.1145/3628161

[40] Dapeng Yan, Zhipeng Gao, and Zhiming Liu. 2023. A Closer Look at Different Difficulty Levels Code Generation Abilities of ChatGPT. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 1887–1898. https://doi.org/10.1109/ASE56229.2023.00096

[41] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 37:1–37:12. https://doi.org/10.1145/3597503.3623316

[42] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2023. Large Language Models Meet NL2Code: A Survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, 7443–7464. https://doi.org/10.18653/V1/2023.ACL-LONG.411

[43] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2024. A Survey of Learning-based Automated Program Repair. *ACM Trans. Softw. Eng. Methodol.* 33, 2 (2024), 55:1–55:69. https://doi.org/10.1145/3631974

[44] Quanjun Zhang, Chunrong Fang, Yang Xie, Yuxiang Ma, Weisong Sun, Yun Yang, and Zhenyu Chen. 2024. A Systematic Literature Review on Large Language Models for Automated Program Repair. *CoRR* abs/2405.01466 (2024). https://doi.org/10.48550/ARXIV.2405.01466 arXiv:2405.01466

[45] 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. *CoRR* abs/2312.15223 (2023). https://doi.org/10.48550/ARXIV.2312.15223 arXiv:2312.15223

[46] Quanjun Zhang, Chunrong Fang, Bowen Yu, Weisong Sun, Tongke Zhang, and Zhenyu Chen. 2024. Pre-Trained Model-Based Automated Software Vulnerability Repair: How Far are We? *IEEE Trans. Dependable Secur. Comput.* 21, 4 (2024), 2507–2525. https://doi.org/10.1109/TDSC.2023.3308897

[47] 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 *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 535–547. https://doi.org/10.1109/ASE56229.2023.00063

[48] 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. *CoRR* abs/2310.08879 (2023). https://doi.org/10.48550/ARXIV.2310.08879 arXiv:2310.08879

[49] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. *CoRR* abs/2404.05427 (2024). https://doi.org/10.48550/ARXIV.2404.05427 arXiv:2404.05427

[50] 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 *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 341–353. https://doi.org/10.1145/3468264.3468544

[51] Yuqi Zhu, Jia Li, Ge Li, Yunfei Zhao, Jia Li, Zhi Jin, and Hong Mei. 2024. Hot or Cold? Adaptive Temperature Sampling for Code Generation with Large Language Models. In *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*, Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan (Eds.). AAAI Press, 437–445. https://doi.org/10.1609/AAAI.V38I1.27798