Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

A search-and-fill strategy to code generation for complex software requirements

Yukun Dong [*], Lingjie Kong, Lulu Zhang, Shuqi Wang, Xiaoshan Liu, Shuai Liu, Mingcheng Chen

Qingdao Institute of Software, College of Computer Science and Technology, China University of Petroleum(East China), Shandong, China

ARTICLE INFO

Dataset link: https://doi.org/10.5281/zenodo.13292164

Keywords:
Low-Code Development
Complex software requirements
Search-and-Fill strategy
Code generation
Natural language processing

ABSTRACT

Context: The realm of software development has seen significant transformations with the rise of Low-Code Development (LCD) and the integration of Artificial Intelligence (AI), particularly large language models, into coding practices. The proliferation of open-source software also offers vast resources for developers.
Objective: We aim to combine the benefits of modifying retrieved code with the use of an extensive code repository to tackle the challenges of complex control structures and multifunctional requirements in software development.
Method: Our study introduces a Search-and-Fill strategy that utilizes natural language processing (NLP) to dissect complex software requirements. It extracts control structures and identifies atomic function points. By leveraging large-scale pre-trained models, the strategy searches for these elements to fill in the automatically transformed program structures derived from descriptions of control structures. This process generates a code snippet that includes program control structures and the implementations of various function points, thereby facilitating both code reuse and efficient development.
Results: We have validated the effectiveness of our strategy in generating code snippets. For natural language requirements involving multifunctional complex structures, we constructed two datasets: the Basic Complex Requirements Dataset (BCRD) and the Advanced Complex Requirements Dataset (ACRD). These datasets are based on natural language descriptions and Python code that were randomly extracted and combined. For the code snippets to be generated, we achieved the best results with the ACRD dataset, with BLEU-4 scores reaching up to 0.6326 and TEDS scores peaking at 0.7807.
Conclusion: The Search-and-Fill strategy successfully generates a comprehensive code snippets, integrating essential control structures and functions to streamline the development process. Experimental results substantiate our strategy's efficacy in optimizing code reuse by effectively integrating preprocessing and selection optimization approach. Future research will focus on enhancing the recognition of complex software requirements and further refining the code snippets.

1. Introduction

Since 2014, the software development area has witnessed a significant transformation with the advent of LCD [1]. LCD is distinguished as a method that abstracts and partially automates each phase of the software development lifecycle, thereby expediting delivery across various systems within a designated domain. Its adoption has seen a steady increase, with industry analysts, such as Gartner Incorporated, forecasting that by 2024, "more than 65% of application development activity will involve Low-Code Developmen" [2].

The field of LCD has particularly benefited from advancements in AI, especially in enhancing the efficiency of LCD workflows. A noteworthy area of AI application in LCD is programming driven by natural language requirements, which currently sees groundbreaking AI methodologies employing language models for automating code generation.

The burgeoning number of open-source software projects on the Internet is further complemented by platforms like GitHub,[1] which hosted over 3.3 billion code repositories by 2024, covering a wide array of programming languages, frameworks, and libraries. Similarly, corporate giants like Google have amassed internal code repositories exceeding 2 billion lines of code [3]. Leveraging such vast code resources and the collective wisdom available online is crucial, as it significantly boosts the efficiency and quality of software development endeavors.

[*] Corresponding author.
E-mail address: dongyk@upc.edu.cn (Y. Dong).

[1] https://github.com/.

https://doi.org/10.1016/j.infsof.2024.107584
Received 21 April 2024; Received in revised form 12 August 2024; Accepted 17 September 2024
Available online 20 September 2024

The realm of automatic code generation has been explored in mainly two directions. One direction attempts to generate complete, executable code directly from a natural language query [4–6], a method facing challenges due to the vast solution spaces encountered, especially in pre-trained models [7]. Enhancements in code generation have been realized through retrieval-enhanced methods, which assist in narrowing down the solution space and improving code quality by integrating external code knowledge databases. For instance, Hashimoto et al. [8] proposed a method that simplifies code generation by retrieving and editing relevant code based on similarity calculations between the input natural language and the model's training data. Similarly, the DocPromptin model [9] dynamically incorporates document libraries to generate code for previously unseen tasks, showcasing significant performance improvements with retrieval-based approaches.

Another direction focuses on generating critical code components, such as API sequences [10], achieving high accuracy but often overlooking essential code structure elements necessary for comprehensive development tasks.

In reality, developers face complex software requirements that often extend beyond the reuse of individual functionalities to include multiple functionalities and the need for specific execution logic between them. These complex software requirements also lead to an increase in program length, which has a significant negative impact on the performance of automatic code generation.

We refer to the "action" components within atomic business rules, characterized by their clarity, conciseness, and independence, as atomic functional points. As these functional points are processed, their relationships significantly impact the logical structure and execution flow of programs. Although there may be multiple types of relationships between atomic functional points, this article primarily considers three key relationships: dependency, mutual exclusion, and feedback. Dependency implies that the output of one functional point directly becomes the input to another, commonly seen in sequential executions; mutual exclusion indicates that only one operation within a functional point can be executed at any given time, typically seen in path selection scenarios; feedback involves the output of a functional point affecting its subsequent execution, which is particularly important in loops or iterative operations.

To assess the complexity of software requirements, this paper proposes a symbolic complexity calculation model. This model considers both the number of atomic functional points and their interrelationships, defined and calculated as follows:

- R: Represents a set of software requirements composed of multiple atomic functional points, expressed as $R = \{r_1, r_2, \dots, r_n\}$.
- $D_{\text{cnt}}(R)$: Represents the complexity of the count of atomic function points, calculated as $D_{\text{cnt}}(R) = |R|$.
- $D_{\text{rel}}(R)$: Represents the complexity of relationships between atomic function points, calculated as $D_{\text{rel}}(R) = \sum_{i=1}^{n} \sum_{j=1,j\neq i}^{n} k(r_i, r_j)$, where the function $k(r_i, r_j)$ quantifies the dependency, mutual exclusion, or feedback relationships between the functional points.
- $D(R)$: Represents the total complexity of the software requirements, integrating both the number and relational complexities of the atomic functional points, expressed as $D(R) = \alpha \cdot D_{\text{cnt}}(R) + \beta \cdot D_{\text{rel}}(R)$, where α and β are coefficients that adjust for the impact of different factors.

Through this symbolic complexity calculation method, we are not only able to accurately assess the structure and potential execution challenges of software requirements but also gain a deeper understanding of their complexity.

When faced with highly complex requirements, developers typically begin with an architectural design, followed by modular design and development. Our Search-and-Fill strategy mirrors this systematic approach, adopting the same architectural and modular development processes to efficiently and effectively address complex software requirements. Our strategy focuses not only on leveraging a vast amount of existing code resources but also on effectively understanding and reconstructing them to meet these complex demands. It emphasizes editing existing code over generating new code from scratch, treating program control structures and functions as key elements in completing code projects.

Specifically, we extract and process descriptions of control structures and atomic function points from the requirements, where control structure descriptions align with the program's sequence, selection, and loop structures. Moreover, atomic function point descriptions, which denote specific operational details related to individual tasks, serve as inputs for external library searches. The search results reuse methods from the code repository, filling them into the program control structure according to varied execution logic, culminating in a comprehensive and adaptable code snippet. This strategy not only streamlines the development process but also enhances development efficiency, marking a notable progression in addressing complex software requirements.

Our contributions are outlined as follows:

- We leverage NLP techniques to intricately analyze complex software requirements provided by developers. Innovatively, we distinguish between control structure descriptions and function descriptions within these requirements, extracting both control structure descriptions and atomic function point descriptions as starting points for generation. We emphasize searching within an extensive repository for relevant implementations of various function points and filling these into the program control structure. This approach effectively generates code snippets for complex software requirements.
- Through the optimization of description preprocessing and search approaches, we achieve more precise program control structures and candidate functions. These serve as an intermediary representation that can be transformed into code snippets for various programming languages and UML activity diagrams
- We curated a dataset for validation by randomly selecting data from the CodeSearchNet [11] dataset, combining them, and manually annotating to evaluate the efficacy of generating code for complex software requirements involving multiple function calls and diverse control structures. The dataset has 1009 entries split into two sections: BCRD with 186 entries for simpler tasks, and ACRD with 822 entries for more complex tasks. BCRD entries average 42.44 words, with code featuring 3.43 function calls and 1.43 control structures. ACRD entries are more complex, averaging 86.69 words and including 6.53 function calls and 2.94 control structures in their code.

The remainder of the paper is structured as follows: Section 2 delves into the technical details of our approach; Section 3 outlines the experimental design, setup, and results analysis; Section 4 discusses our work's significance; Section 5 addresses potential validity threats; Section 6 reviews related literature; and Section 7 concludes the paper and suggests directions for future research.

2. Our approach

2.1. Strategy overview

The Search-and-Fill strategy ingeniously employs NLP techniques and pre-trained models to dissect and comprehend complex software requirements, enabling the construction of efficient code snippets. It is predicated on a two-pronged approach: (1) searching for pertinent implementations of function points based on an accurate understanding and extraction of input requirements, and (2) filling the identified placeholders within the program control structure with these implementations to craft a code snippet. **Operational Phases of the Search-and-Fill strategy:**

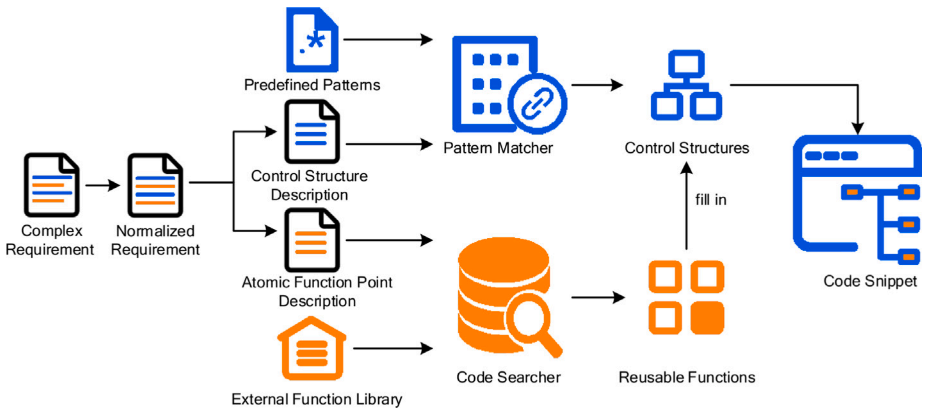


Fig. 1. The workflow of Search-and-Fill strategy.

(1) **Normalization (N):** This stage preprocesses the input requirements (R) into a more analyzable form (R'). The stage involves:

- Sentence tokenization
- Pronoun replacement
- Lemmatization

This can be represented as $N(R) \rightarrow R'$.

(2) **Extraction (E):** In this stage, the strategy generates a rough code snippet (CS) and extracts atomic function point descriptions (A) from R'. This involves:

- Parsing R' into control structure types, conditions, and content, represented as $(Type, Condition, Content)$. This serves as the basis for further transforming into CS.
- Isolating atomic function points for each $Content$.

This can be represented as $E(R') \rightarrow \{CS, A\}$.

(3) **Synthesis (S):** The final stage synthesizes the extracted elements into a comprehensive code snippet CS', employing a function search and selection algorithm to:

- Map A to corresponding implementations of functions (F) from a repository
- Place each F into its specified placeholder within CS.

This process is formalized as $S(\{CS, F\}) \rightarrow CS'$.

This process is illustrated in Fig. 1.

2.2. Normalization process

Real software requirements are often complex, rather than simple descriptions of individual functions. For instance, the left part of Fig. 2 illustrates a segment containing four function points and a selection structure. In the selection structure, there is also a loop structure. In the preprocessing stage, we have taken the following three key steps to ensure a more accurate understanding of the requirements in subsequent stages.

(1) **Sentence Tokenization:** In this step, we break down the original requirements into individual sentences, considering specific logical connection keywords to determine whether to merge two sentences. If two sentences each have leading and trailing keywords, they are combined into one sentence. Table 1 presents the logical connection keywords. This process aids in partitioning lengthy requirements into distinct sentences, facilitating a more precise understanding and extraction of the necessary content in subsequent processing. In the provided example, the original requirement is broken down into three sentences. These are then grouped into two parts, where the second part is formed by merging the second and third sentences, as shown in the first step of Fig. 2.

(2) **Pronoun Replacement:** In complex software requirements, instances of unclear pronoun references may occur. Pronoun replacement involves substituting pronouns in sentences with specific words or phrases to reduce ambiguity and enhance sentence clarity. In the example above, after sentence tokenization, we replace “them” with “the query results” in this step, as illustrated in the second step of Fig. 2.

(3) **Lemmatization:** Lemmatization involves reducing words to their base form to minimize word variation and enhance sentence consistency. This process aids in better understanding sentences during subsequent processing and analysis. For instance, in the last step of Fig. 2, “results” is further lemmatized to “result”, and the initial capitalization is normalized.

Table 1
Logical connected keywords.

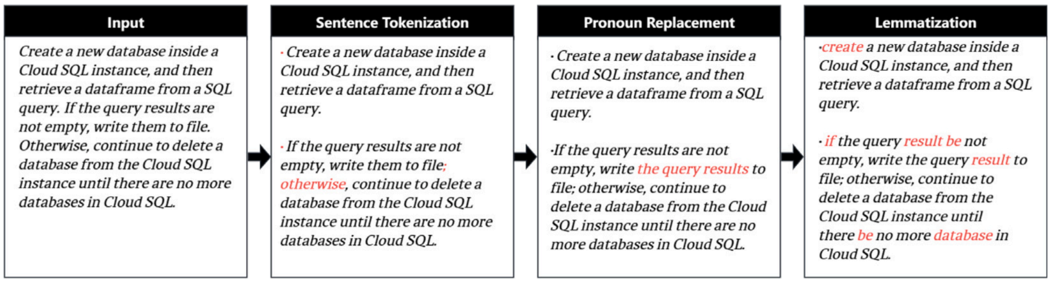
Leading keywords	Trailing keywords
if	if not
in the case of	else
on the condition that	otherwise
–	however
when	repeat
while	repeatedly
for each	–
to all	–

Through these three steps, we break down a complex software requirement into a set of sub-requirements. Each item in this preprocessed requirement set undergoes standardized processing, making it more accessible for understanding and further analysis. This preprocessing lays a favorable foundation for the subsequent extraction of control structure descriptions and atomic function point descriptions.

2.3. Extraction mechanisms

The process of control structure extraction involves identifying different control structures — such as sequences, selections, and loops — and pinpointing the exact locations within the code snippet meant for filling with functional points. This step is crucial because it creates a solid foundation for precise code filling, ensuring that each function point is properly filled.

The process of atomic function point extraction identifies function points within content segments, further decomposing these points into atomic function points to ensure that each result to be searched is focused solely on accomplishing a single task. This prepares for accurate searches for corresponding implementations of functions.



Input

Create a new database inside a Cloud SQL instance, and then retrieve a dataframe from a SQL query. If the query results are not empty, write them to file. Otherwise, continue to delete a database from the Cloud SQL instance until there are no more databases in Cloud SQL.

Sentence Tokenization

· Create a new database inside a Cloud SQL instance, and then retrieve a dataframe from a SQL query.

· If the query results are not empty, write them to file; otherwise, continue to delete a database from the Cloud SQL instance until there are no more databases in Cloud SQL.

Pronoun Replacement

· Create a new database inside a Cloud SQL instance, and then retrieve a dataframe from a SQL query.

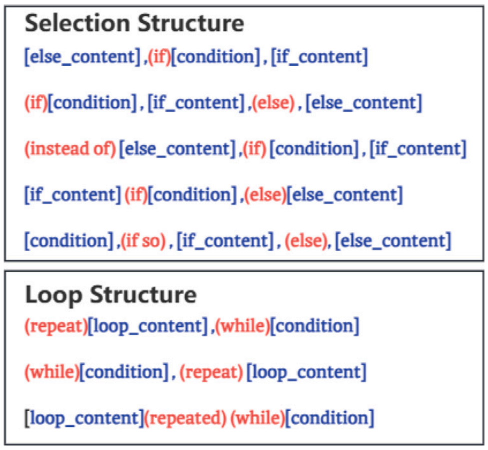
· If the query results are not empty, write the query results to file; otherwise, continue to delete a database from the Cloud SQL instance until there are no more databases in Cloud SQL.

Lemmatization

· create a new database inside a Cloud SQL instance, and then retrieve a dataframe from a SQL query.

· if the query result be not empty, write the query result to file; otherwise, continue to delete a database from the Cloud SQL instance until there be no more database in Cloud SQL.

Fig. 2. Normalization process.



Selection Structure

[else_content] ,(if)[condition] , [if_content]

(if)[condition] , [if_content] ,(else) , [else_content]

(instead of) [else_content] ,(if) [condition] , [if_content]

[if_content] (if)[condition] ,(else)[else_content]

[condition] ,(if so) , [if_content] , (else), [else_content]

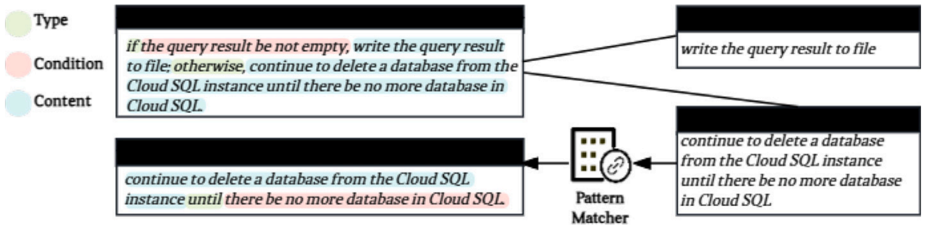
Loop Structure

(repeat)[loop_content] ,(while)[condition]

(while)[condition] , (repeat) [loop_content]

[loop_content](repeated) (while)[condition]

Fig. 3. Basic predefined patterns.



if the query result be not empty, write the query result to file; otherwise, continue to delete a database from the Cloud SQL instance until there be no more database in Cloud SQL.

write the query result to file

continue to delete a database from the Cloud SQL instance until there be no more database in Cloud SQL

continue to delete a database from the Cloud SQL instance until there be no more database in Cloud SQL.

Fig. 4. An example of matching predefined patterns.

2.3.1. *Control structure extraction*

In the preprocessing stage, the complex software requirement is parsed into a series of sub-requirements, each adhering to a sequential structure denoted as $SR = \{SR_1, SR_2, \dots, SR_n\}$. This structured decomposition is pivotal for the systematic identification of intrinsic control structures, including sequences, selections, and loops. The process involves matching each SR_i against a curated set of predefined regular expression patterns, meticulously demonstrated in Fig. 3. These patterns are designed to highlight keywords or phrases indicative of control structures. For instance, within the context of selection structures, keywords and phrases such as "if", "in the case of", and "on the condition that" are pivotal in signaling the true branch. In Fig. 4, one of these keywords or phrases is selected and encapsulated in parentheses, while the divided condition and content parts are distinctly marked with square brackets, offering a clear delineation of the control structure's components.

This identification phase segments each SR_i into a tripartite structure, represented as: $TS_{i,k} = (Type_{i,k}, Condition_{i,k}, Content_{i,k})$, where:

- i **:** Index of the original sub-requirement in SR.
- k **:** Recursive depth at which control structures are identified in SR_i.
- $Type$ **:** Identifies the control structure as a sequence, selection (e.g., **if-else** statements), or loop (e.g., **while** loops), essential for applying correct control logic.
- $Condition$ **:** Defines the criteria for execution paths within the control structure, such as the conditions for executing the true branch of an **if** statement or continuing a **while** loop.
- $Content$ **:** Specifies the set of operations to be executed if the conditions are met, including function calls, or more complex nested structures.

It ensures that each identified structure $TS_{i,k}$ is standardized into a representation delineating the control structure type, its governing condition, and a detailed analysis of the content.

The recursive integration of these identified control structures into a code structure, CS, is formalized as $CS = \bigcup_{i=1}^{m} \bigcup_{k=1}^{n_i} \{TS_{i,k}\}$, where m signifies the total number of sub-requirements, and n_i is the count of control structures identified within each SR_i. This formulation ensures that each $TS_{i,k}$ is integrated into CS.

The process unfolds through the following steps:

- **Step 1: Initialization** - Establish $CS = \emptyset$.
- **Step 2: Identification and Segmentation** - For each $SR_i \in SR$, match SR_i against regular expressions to identify $TS_{i,k} = (Type_{i,k}, Condition_{i,k}, Content_{i,k})$.

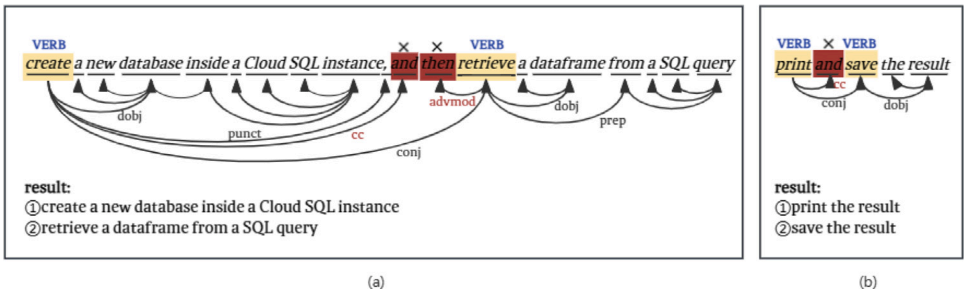

VERB
create a new database inside a Cloud SQL instance, and then retrieve a dataframe from a SQL query
dobj
punct
cc
conj
advmod
dobj
prep
result:
①create a new database inside a Cloud SQL instance
②retrieve a dataframe from a SQL query
(a)
print and save the result
cc
conj
dobj
result:
①print the result
②save the result
(b)

Fig. 5. An illustration of part-of-speech tagging process.

- **Step 3: Integration** - Amend CS by integrating $CS = CS \cup \{TS_{i,k}\}$.
- **Step 4: Recursive Structuring** - For each $Content_{i,k}$ housing additional control structures, reapply **Step 2** and **Step 3** recursively, utilizing i and k to denote the source and depth of decomposition and integration, respectively.

To illustrate this methodology further, an existing example is introduced, as depicted in Fig. 4. This example demonstrates the steps of this methodology in addressing complex software requirements and extracting their control structures.

2.3.2. Atomic function point extraction

Following the identification of control structures in Section 2.3.1, where control structures within the sub-requirements SR are identified and structured into $CS = \bigcup_{i=1}^{m} \bigcup_{k=1}^{n_i} \{TS_{i,k}\}$, the process proceeds to isolate atomic function points from the content segments $Content_{i,k}$ of each identified control structure $TS_{i,k}$. This process employs the SpaCy NLP toolkit,[2] utilizing part-of-speech tagging and dependency parsing, with a particular focus on verbs as indicators of operations. The methodology refines function point extraction by expanding upon verbs through dependency relationships and omitting non-essential modifiers. Additionally, it meticulously examines conjunct relationships within the content to accurately extract function descriptions. The steps include:

(1) **Part-of-speech tagging:** All elements within $Content_{i,k}$ undergo grammatical role tagging, with a particular emphasis on verbs that denote operational actions. This foundational step is crucial for identifying the primary actions within each $Content_{i,k}$, as illustrated with tagged verbs highlighted in Fig. 5.
(2) **Dependency parsing:** This phase refines the identification of verbs through dependency analysis, eliminating extraneous modifiers to focus squarely on the core operational essence. Fig. 5(a) exemplifies the pruning of modifiers like "then", deemed irrelevant to the core operation, thereby streamlining the extraction process.
(3) **Conjunct Relationship Analysis:** A meticulous examination of conjunct relationships within $Content_{i,k}$ ensures the extraction of accurate and comprehensive atomic function point descriptions. Fig. 5(a) and (b) shows different types of conjunct relationships, including instances of shared or not shared objects between verbs, thereby necessitating careful analysis and adaptation in the extraction process.

To precisely represent the capability of a single $Content_{i,k}$ to be decomposed into multiple atomic function point descriptions, the output structure is refined to: $A_{ik} = \{A_{ik1}, A_{ik2}, \dots, A_{ikn}\}$. Each element in the set A_{ik} represents an independent atomic function point description. This structuring ensures that every atomic function point description is directly linked to its originating content segment. Algorithm 1 outlines the extraction process.

Algorithm 1 extract atomic function point

```
    Input:$S$ (String) - Function point description.
    Output:$T$ (List) - A list of atomic function points descriptions.
 1: $T \leftarrow [\,]$.
 2: $D \leftarrow$ spaCyNLP($S$)
 3: $visited \leftarrow [\,]$.
 4: function TraverseAndPrune($Node$)
 5:     visited ← visited $\cup \{Node\}$
 6:     for each $child_m$ of $Node.children$ do
 7:         if $child_l \in$ PruneConditions then
 8:             $Node.children \leftarrow Node.children \setminus \{child_m\}$
 9:         else
10:             $Node.children[m] \leftarrow$ TraverseAndPrune($child_m$)
11:         end if
12:     end for
13:     return $Node$
14: end function
15: for each $token_i$ in $D$ do
16:     if $token_i.pos\_$ = "VERB" and $token_i \notin$ visited then
17:         visited ← visited $\cup \{token_i\}$
18:         $Tree \leftarrow$ TraverseAndPrune($token_i$)
19:         $T \leftarrow T \cup \{Tree\}$
20:     end if
21: end for
22: for each $tree_j$ in $T$ do
23:     for each $child_k$ in $tree_j.head.children$ do
24:         if "obj" $\subseteq child_k.dep\_$ then
25:             $Object = child_k.children$
26:         end if
27:     end for
28:     for each $child_l$ in $tree_j.head.children$ do
29:         if $child_l.dep\_$ = "conj" and $child_l.pos\_$ = "VERB" then
30:             $tree_j.children \leftarrow tree_j.children \setminus \{child_l\}$
31:             $T \leftarrow T \cup \{child_l\}$
32:             if $\forall child \in child_l.children, \neg(child.dep\_$
    = "obj") then
33:                 $child_k.children \leftarrow child_k.children \cup \{Object\}$
34:             end if
35:         end if
36:     end for
37: end for
38: return T
```

2.4. Code snippet synthesis

The core of the Search-and-Fill strategy involves searching for the suitable implementations of function points that match our identified atomic function points A, and then filling these function calls into the

[2] https://spacy.io/.

predefined placeholders within the structured code snippets CS. This strategy not only makes the development process more efficient but also ensures that each component is precisely positioned.

The interaction between CS and A during the Search-and-Fill process operates as follows:

Search: With A as the directive, the search phase scours a repository for function calls that correspond to each atomic function point. This is facilitated by a function matching and selection approach, aimed at securing the optimal implementations of function points.

Fill: Upon locating the appropriate function calls via the search approach, they are then allocated to their designated slots within CS. This allocation process is shaped by the layout of the program control structure, ensuring that each code snippet is precisely positioned to maintain the intended control flow.

Subsequent sections will provide a detailed exploration of this matching and selection approach, detailing the fill process and the different transformations of the fill outcomes.

2.4.1. Searching for optimal reusable functions

In the context of the Search-and-Fill strategy, function matching and selection is a critical step in the synthesis of code snippets. For each atomic function point description A_i derived from any given content segment, the strategy initiates a search across an extensive code library. This search action is implemented using GraphCodeBERT [12] to pinpoint corresponding functions, thereby assembling a function set F. The process is mathematically articulated as:

$$F_i = \mathrm{SortDescending}(f_{i1}, f_{i2}, \ldots, f_{in}) \tag{1}$$

Here, F_i represents the function set correlated with the ith atomic description A_i within a content segment. The function set is sorted in descending order of relevance scores, with each function f_{ij} accompanied by a corresponding relevance score $score_{ij}$ within the collection $Score_i = \{score_{i1}, score_{i2}, \ldots, score_{in}\}$.

This strategy acknowledges scenarios where executing a single function might be more efficient than multiple functions for achieving multiple functionalities. To address this, an original description A_0, which is undecomposed, is introduced for each content before searching. This strategy ensures the consideration of overall relevance when identifying the best-matching function. $Score_0$ is the set of scores for the candidate functions corresponding to A_0. As the length of the function description l increases, the relevance of the function tends to decrease. Therefore, an adjusted relevance score is calculated using the formula:

$$score' = \mathrm{sigmoid}(k \times l + b) \times score \tag{2}$$

where $score$ is the original relevance score, l is the description length, k and b are parameters derived from experimentation.

Then, the set $Score'_i$ is defined as the collection of adjusted relevance scores $score'$ for the top k candidate functions corresponding to a specific content description A_i. These scores are derived from the formula given above, where each $score'$ represents the adjusted relevance of a candidate function within the function set F_i. Specifically, $Score'_i$ is composed as follows: $Score'_i = \{score'_{i1}, score'_{i2}, \ldots, score'_{ik}\}$

The adjusted scores are used to evaluate both the original and the extracted atomic function point descriptions. By employing a Cartesian product approach, individual scores from $Score'_1$ to $Score'_n$ for each description are combined to calculate the total score for every possible combination. This total is then augmented by the score of the undivided descriptions, $Score'_0$, to facilitate an overall ranking. The top k items with the highest cumulative scores are selected as the final search results. The formula for this calculation is outlined below:

$$S = Score'_1 \times Score'_2 \times \cdots \times Score'_n \tag{3}$$

$$r' = \left\{ \frac{\sum s}{n} \mid s \in S \right\} \tag{4}$$

$$r = Score'_0 + r' \tag{5}$$

$$\mathrm{Result} = \mathrm{top}_k(r) \tag{6}$$

The steps for acquiring the optimal matching function are illustrated in Fig. 6.

2.4.2. Filling reusable functions into code snippet

The integration process concentrates on mapping the selected functions F directly into the corresponding atomic function point descriptions A within the code snippet CS. This process can be succinctly represented as:

$$\forall A \in CS, \exists F : M(A \rightarrow F) \rightarrow CS' \tag{7}$$

Here, $M(A \rightarrow F)$ represents the operation of mapping functions F selected from atomic function point descriptions A to their specific placeholders within the control structure. CS' represents the transformed version of CS.

This meticulous mapping lays the groundwork for the transformation of CS', which emerges as an intermediary representation (as shown in Fig. 7). CS' is capable of evolving into code snippets compatible with multiple programming languages. Specifically, the Python code snippet, denoted as CS'_{python}, is exemplified (as shown in Fig. 8), and the pseudocode, denoted as CS'_{pseudo}, is also exemplified (as shown in Fig. 9).

Using visualization tools like PlantUML,[3] CS'_{pseudo} is transformed into UML activity diagrams $CS'_{\mathrm{UML}} = U(CS'_{\mathrm{pseudo}})$. For example, the result of this transformation of CS'_{pseudo} in Fig. 9 is displayed in Fig. 10. This graphical representation significantly enhances the code snippet's comprehensibility by visually depicting the interconnections among its various components.

Thus, CS' not only serves as a bridge between abstract design and tangible code but also provides developers with a versatile intermediary representation that can be transformed into multilingual code and visual displays, streamlining the transition from conceptual design to executable software solutions.

3. Experiment

3.1. Dataset

We utilized the widely recognized CodeSearchNet [11] dataset as a foundation to construct a specialized dataset tailored for code generation research targeting complex software requirements. CodeSearchNet encompasses a variety of programming languages and contains a rich collection of code snippets and natural language descriptions, providing the necessary complexity and diversity for our study. We carefully filtered and reorganized this dataset to ensure the selected data accurately reflects complex software requirements.

Specifically, our constructed dataset includes 1009 entries, comprising natural language descriptions (**nl**) and their corresponding programming language implementations (**pl**), referred to as **nl-pl** pairs. Each **pl** component is crafted from a random sample within the CodeSearchNet dataset, combined through various logical structures, and the corresponding **nl** component is manually annotated as the requirement description for the **pl**, ensuring they precisely mirror the control structures and function points, accurately reflecting complex software requirements.

The dataset is divided into two parts to showcase programming tasks of varying complexities. The first part, the BCRD(Basic Complexity Requirements Data), includes 186 entries with descriptions averaging 40.81 words, encompassing 3.42 function calls and 1.46 control structures, primarily focusing on less complex software requirements challenges. Table 2 presents the top five entries of BCRD,

[3] https://plantuml.com/.

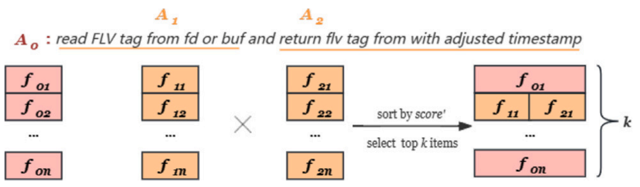
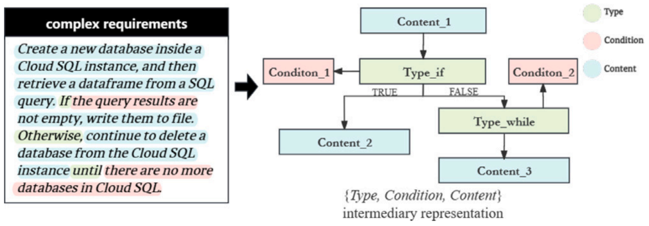


Fig. 6. The steps for acquiring the optimal matching function.


complex requirements

Create a new database inside a Cloud SQL instance, and then retrieve a dataframe from a SQL query. If the query results are not empty, write them to file. Otherwise, continue to delete a database from the Cloud SQL instance until there are no more databases in Cloud SQL.

Fig. 7. Intermediary representation.

```
""""
Create a new database inside a Cloud SQL instance, and then retrieve a
 dataframe from a SQL query.
If the query results are not empty, write them to the new database.
 Otherwise, continue to delete a
database from the Cloud SQL instance until there are no more databases
 in Cloud SQL.
"""

if __name__ == "__main__":
  create_database(self, instance_id, database_id, ddl_statements,
 project_id=None)
  get_pandas_df(self, hql, schema='default')

  if the query results are not empty:
    write_object_to_file(self,query_results,filename,fmt="csv",
,coerce_to_timestamp=False,
                 record_time_added=False)
  else:
    while True:
      if there are no more databases in Cloud SQL:
        break
      delete_database(self, instance_id, database_id, project_id=None)
```

Fig. 8. Intermediary representation to code.

```
@startuml
start
  :create a new database inside a Cloud SQL instance;
  :retrieve a dataframe from a SQL query;

  if (the query results are not empty) then (yes)
    :write the query results to file;
  else (no)
    repeat
      :continue to delete a database from the Cloud SQL instance;
    repeat while (there be no more database in Cloud SQL) is (True)
  endif
stop
@enduml
```

Fig. 9. Intermediary representation to pseudocode.

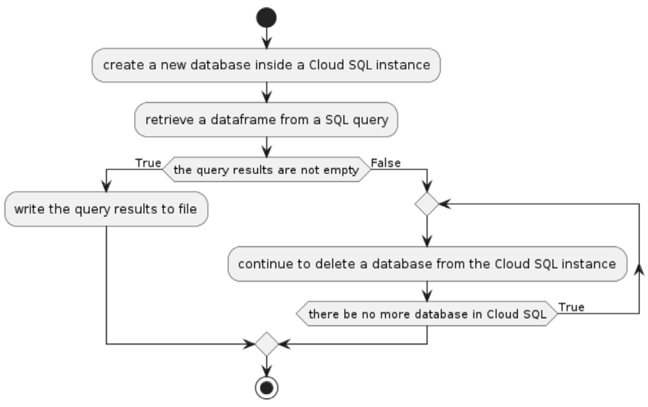

create a new database inside a Cloud SQL instance
retrieve a dataframe from a SQL query
True
the query results are not empty
False
write the query results to file
continue to delete a database from the Cloud SQL instance
there be no more database in Cloud SQL
True

Fig. 10. Visualization of intermediary representation as UML activity diagrams.

Table 2
Top five entries of BCRD.

nl	pl
When processing the data, instead of recursively iterating over all schema sub-fields, if a certain condition is met, get available filters from the dataset you've selected.	`while(if(get_filters;)else(iterate_schema;))`
If a tag-specific condition is met, execute the function for the node constructor to be used in tags, if not, creates or returns a group, and then proceed to get deposits.	`if(for_tag;)else(_all_create_or_get_group;get;)`
If a valid pair is needed, repeat getting deposits and making a nice string representation of a pair of numbers until deposits retrieval is in progress. Otherwise, retrieve the first Signature ordered by mangling descendant.	`if(while(get;nice_pair;))else(first;)`
When the conversion process is in progress, convert Lisp collections into Python collections, and when the data categorization and tomography basis generation is underway, cut a numeric vector into categorical buckets, then generate a TomographyBasis object, and finally, associate the given client data with the item at position n.	`while(to_py;while(cut;tomography_basis;set_data;))`
When the span conversion and copying process is ongoing, repeat converting encoded spans to a different encoding and returning a full copy of the object, optionally renaming it, if key generation is initiated, generate the key associated with the specified address; otherwise, close an existing connection, which cannot be used again once closed.	`while(convert_spans;copy;if(get_key_for;)else(close;))`

illustrating the dataset's focus on the generation of structurally simpler code snippets.

The second part, the ACRD(Advanced Complexity Requirements Data), comprises 822 entries with more detailed descriptions averaging 84.02 words, 6.79 function calls, and 2.98 control structures, addressing more complex programming scenarios. Table 3 showcases the top five entries of ACRD, further illustrating the capacity of our dataset to handle complex software requirements.

This division into BCRD and ACRD is crucial for testing the capabilities of our proposed Search-and-Fill strategy, designed to translate complex software requirements into code snippets. By encompassing a wide variety of functional requirements and programming structures, the dataset lays a solid foundation for assessing how well the Search-and-Fill strategy can understand and address tasks of varying complexity. The complete dataset, including both BCRD and ACRD parts, is available for access.[4]

Although no existing dataset perfectly meets all our requirements for high complexity and function reuse, our approach ensures that our dataset not only meets our research needs but also can be broadly applied to code generation research. Future research may benefit from the creation of more specialized datasets, which will further advance code generation technology. Through this meticulous data preparation and strategic implementation, our study provides a solid foundation for understanding and resolving programming tasks of different complexities.

3.2. Experimental design

Our experiments are designed to scrutinize the efficacy of the proposed Search-and-Fill strategy in generating code snippets from complex software requirements. The evaluation is structured around two pivotal research questions (RQs):

RQ1: How does the Search-and-Fill strategy perform in generating accurate code snippets across datasets with varying levels of complexity compared to other language models?

RQ2: Does preprocessing of requirements and optimization of search approaches significantly enhance the code snippet generation process?

To address **RQ1**, we compare our method with three Large Language Models (LLMs): ChatGPT-4o mini,[5] Code-GeeX [13], and Gemini 1.5 Pro [14]. These models were selected because they represent advanced technology in the NLP field, capable of handling complex tasks, particularly excelling in code generation when confronted with intricate requirements. LLMs have been extensively used to tackle NLP problems and are known to excel at generating code when faced with complex requirements. Due to token limitations, method reuse is not feasible; therefore, the comparison excludes the search step.

For **RQ2**, we employ an ablation study to evaluate the contributions of our preprocessing and optimization approaches. We utilize manually annotated descriptions as input and compare the generated code snippets with the predetermined **pl** within our dataset.

3.3. Evaluation metrics

The evaluation of our strategy's performance employs the BLEU metric [15], renowned for its efficacy in the domain of code generation. This metric evaluates the coherence between the generated code and the reference code by measuring the overlap of n-grams, with an emphasis on ensuring both accuracy and fairness in the evaluation process. Typically, n is set to 4 in code generation scenarios. Ensuring the accuracy and impartiality of evaluations, BLEU scoring integrates penalty mechanisms for n-gram operations, manifesting as the product of weighted n-gram sums and penalty terms. The BLEU score is calculated as:

$$BLEU = BP \times \exp\left(\sum_{n=1}^{N} \omega_n \log p_n\right) \tag{8}$$

where:

- BP represents the penalty term, defined as:
$$BP = \begin{cases} 1 & \text{if } c \geq r \\ e^{(1-r/c)} & \text{if } c < r \end{cases} \tag{9}$$
- c denotes the length of the output code.
- r represents the length of the reference code.
- ω_n are the weights assigned to each n-gram.
- p_n is the precision of n-grams.

We distill the generated code snippet, selecting the most congruent function calls as outputs, and subsequently streamline them to align with the **pl** structure within the dataset. Given our study's emphasis on structural insights and function invocation patterns, these facets are distinctly embodied within the Abstract Syntax Tree nodes. Transforming the code snippets and subsequently pruning them, we leverage Tree-Edit-Distance-based Similarity (TEDS) [16] to validate the precision of the generated code snippets. Calculating TEDS involves determining the minimum number of tree editing operations required to transform one tree structure into another. The trees being compared are denoted as T_a and T_b, the TEDS calculation formula is as follows:

$$TEDS(T_a, T_b) = 1 - \frac{\text{EditDist}(T_a, T_b)}{\max(|T_a|, |T_b|)} \tag{10}$$

where $|T_a|$ and $|T_b|$ represent the total number of nodes in trees T_a and T_b respectively. The function EditDist(T_a, T_b) calculates the minimum number of edit operations required to transform tree T_a into tree T_b. This formula provides a normalized measure of similarity, with a higher TEDS value indicating greater similarity between the two trees.

[4] DOI:10.5281/zenodo.13292164.

[5] https://platform.openai.com/docs/models/gpt-4o-mini.

Table 3
Top five entries of ACRD.

nl	pl
In the case of the interface map is available for HType generation, generate a flattened register map for HStruct, otherwise, return a new Frame that fills NA along a given axis and along a given direction with a maximum fill length. Additionally, it receives a dictionary of `connection_params` to set up a connection to the database. Get info for all filters. While the initial setup is incomplete,calculate a subcircuit that implements this unitary, write API tokens to a file, and produce an appropriate `_ProductsForm` subclass for the given render type.	`resized_crop;if(_wrap_handling;) else ( while (children;_build_cryptographic_parameters;)) while (_collapse_leading_ws;)`
Crop the given PIL Image and resize it to desired size. in the case of resizing is complete, starts running a queue handler and creates a log file for the queue,however, while child elements exist, browse node's children within the browse node tree and build a CryptographicParameters struct from a dictionary. while the input is a header, preserve the newlines,all others need not.	`while(omim; while(safe_sum;))send_email;if(_pos_changed;) else(create_random_seq;kalman_transition;get_train_op;)`
Execute a code object where the inputs and behavior should match those of `eval_` and `exec_` until code execution is no longer in progress. on the condition that the content is ready to copy, then return a full copy of the object optionally renaming it. Otherwise, generate the key associated with the specified address. Repeat the process of returning the training job information associated with `job_name` and printing CloudWatch logs until the training job logging is completed. Then, display the Scout dashboard.to all feature, make the features beat-synchronous and create a new instance of a rule by merging two dictionaries.	`while(execute;)if(copy;) else(get_key_for;) while(describe_training_job_with_log;)index; while(compute_beat_sync_features;from_yaml;)`
Instead of overriding config options with user-specified options and getting the train operation for the given loss, in the case of the validation check is passed, convert a file object with JSON-serialized pyschema records to a stream of pyschema objects until the data processing loop is initiated. In the case of leading whitespace, the description header must preserve newlines, otherwise, get course information using the Ecommerce course API. Given a LuminosoClient pointing to the root of the API, and a filename to read JSON lines from, create a project from the documents in that file.	`if( while(mr_reader;)) else(__add_user_options;get_train_op;)if(_collapse_leading_ws;) else(_get_course_content_from_ecommerce;)upload_docs;`
Gather only metadata statements and return them. In the case of metadata validation is true, make a function that applies a list of Bijectors forwards until the transformation sequence is initiated. Otherwise, return the string name for this dtype. Get a chunk from the input data converts it to a number and encodes that number. Reads FLV tags from fd or buf and returns them with adjusted timestamps in the case of the encoding process starts, if not, get document from index with its `id_GET772210180J` and tell postgres to encrypt this field using PGP. At last, wraps a function with reporting to errors backend.	`gather_metadata_statements;if( while(forward_transform_fn;)) else(name;)_encode_chunk;if(iter_chunks;) else(do_GET;get_placeholder;)wrap_function;`

Table 4
Performance comparison on BCRD and ACRD datasets.

Approach	BCRD BLEU-4	BCRD Average TEDS	ACRD BLEU-4	ACRD Average TEDS
CodeGeeX [13]	0.7384	0.7377	0.7934	0.6646
ChatGPT-4o mini[a]	0.7822	0.7353	0.7653	0.6409
Gemini 1.5 Pro [14]	**0.9043**	0.8548	0.8156	0.6559
Search-and-Fill Strategy	0.8910	**0.8603**	**0.9166**	**0.8128**

[a] https://platform.openai.com/docs/models/gpt-4o-mini.

Table 5
BLEU-4 scores under different conditions.

Approach	BCRD	ACRD
No preprocessing or selection	0.3797	0.4749
Only preprocessing	0.5000	0.5782
Only selection optimization	0.4187	0.5176
Preprocessing and selection	**0.5403**	**0.6326**

Table 6
Average TEDS under different conditions.

Approach	BCRD	ACRD
No preprocessing or selection	0.6586	0.5373
Only preprocessing	0.7464	0.7297
Only selection optimization	0.6879	0.6972
Preprocessing and selection	**0.7723**	**0.7807**

3.4. Experimental results

Addressing RQ1: Table 4 provides a comprehensive comparison, showcasing the BLEU-4 and Average TEDS scores generated by different approaches across two datasets with varying complexity levels (BCRD and ACRD), Fig. 11 respectively visualize these results. These scores measure the accuracy of the generated code snippets. Overall, the results demonstrate that our method outperforms other LLMs. Although Gemini 1.5 Pro [14] achieves slightly higher BLEU-4 scores than our strategy on the BCRD dataset, its effectiveness significantly decreases on the more complex ACRD dataset. This contrast underscores the adaptability and effectiveness of our strategy in addressing complex coding challenges.

Addressing RQ2: Tables 5 and 6 provide detailed insights into the BLEU-4 and TEDS scores achieved under various conditions, with corresponding visualization in Fig. 12. The application of both preprocessing and selection optimization approaches led to the highest BLEU-4 scores, with 0.5342 for BCRD and 0.6326 for ACRD, indicating significant improvements in the strategy's ability to generate accurate code snippets, especially in the more complex ACRD dataset. This suggests that these approaches are particularly effective for challenging software requirements. Additionally, these methods increased the Average TEDS to 0.7699 for BCRD and 0.7807 for ACRD, showing that the generated code snippets have become more structurally similar to the reference code, thus confirming the effectiveness of the combined approaches in enhancing precision. A comparative analysis reveals that the synergistic use of preprocessing and selection optimization offers substantial improvements in both BLEU-4 and TEDS metrics, greatly enhancing the strategy's capacity to comprehend and accurately translate complex software requirements into code snippets.

In summary, the experimental results clearly demonstrate the superior performance of our method in generating accurate and structurally sound code snippets, particularly for complex software requirements. The combined use of preprocessing and selection optimization has proven to be especially effective, significantly enhancing the adaptability and precision of our approach. These results confirm our method's efficacy in addressing diverse coding challenges.

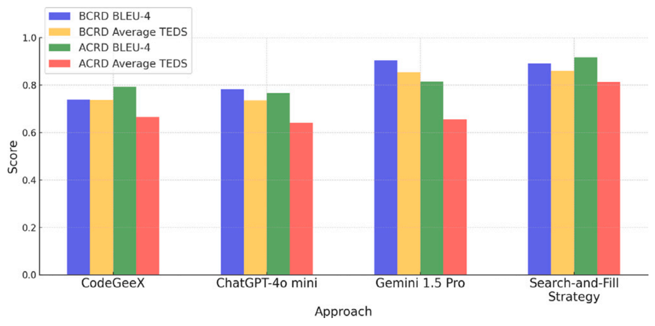


Fig. 11. Performance comparison on BCRD and ACRD.

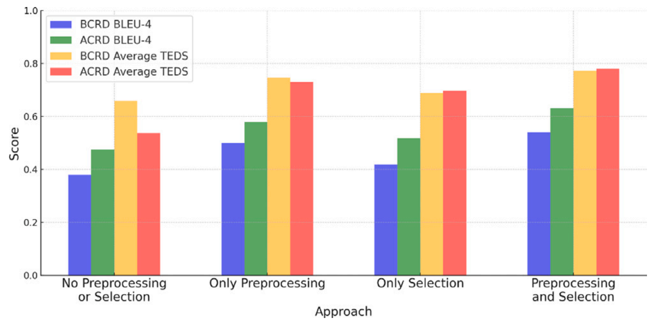


Fig. 12. Comparison under different conditions.

Upon examining the generated outcomes, it was observed that the limitations associated with the external resources used and the accuracy of the search model significantly impacted the results. As illustrated in Fig. 13, despite our approach's accurate identification of the requirements, both the BLEU-4 scores and TEDS were not as high as expected. In this example, three out of four atomic function descriptions failed to retrieve the expected functions. This suggests that even with an accurate understanding of the requirements, the quality of the final outcomes is still influenced by the quality of external resources and the precision of the search model.

Regarding the running cost, our primary focus was on how to implement our search strategy using the search model, without optimizing the search model itself. Therefore, the running time and resource consumption of our strategy mainly depend on the performance of the search model itself. In our experiments, we used the existing search model, GraphCodeBERT [12], to complete the search steps. GraphCodeBERT has already proven its efficiency and effectiveness in code search tasks. Our strategy is designed to reduce the time required for manually extracting search inputs and integrating search results, and it has successfully achieved this goal.

These findings underscore the interplay between strategy effectiveness and external dependencies, indicating a need for enhanced search models and better integration with code repositories to improve result accuracy in future work.

4. Discussion

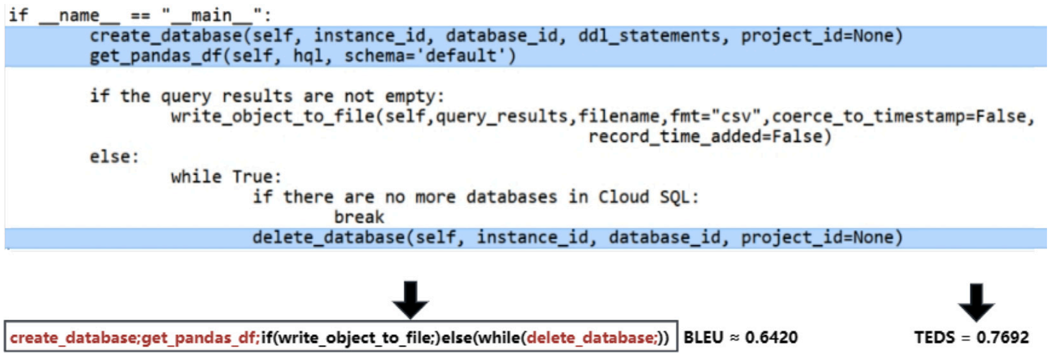
In this study, we introduced a novel Search-and-Fill strategy designed to generate code snippets for complex software requirements. Leveraging NLP techniques and large-scale pre-trained models, the Search-and-Fill strategy demonstrates a significant stride towards automating the software development process, particularly in the context of LCD. Our findings underscore the potential of integrating the concept of "editing retrieved code is more effective than direct code generation" with the practice of leveraging a vast repository of reusable code resources. This integration is pivotal in addressing the nuances of complex software requirements, thereby enhancing the efficiency and quality of software development.

The Search-and-Fill strategy's efficacy is attributed to its methodical approach, which includes preprocessing input requirements, extracting control structure and atomic function point descriptions, and synthesizing these components into a structured code snippet. This process not only streamlines the development cycle but also aligns closely with the practical real complex software requirements.

Our experimental results reveal that the Search-and-Fill strategy significantly improves code reuse by facilitating the generation of code snippets that are closely aligned with the predefined structures of complex software requirements. Furthermore, the integration of preprocessing and optimization approaches has proven to be crucial in enhancing the strategy's performance. These approaches enable the Search-and-Fill strategy to more effectively interpret complex software requirements and identify the most relevant function calls for reuse, thereby reducing the time and effort required for development.

Typically, code generated through semi-automated processes can be more time-consuming to correct than writing code from scratch, especially when dealing with complex requirements. Frequent errors in such generated code often lead developers to spend more time making corrections than they would if they had written the code from scratch. However, although our method also falls under semi-automated code generation, requiring the generation of the program's control structures and the integration of searched reusable functions for code filling, it effectively reduces the time spent on corrections compared to traditional methods. This is because our strategy precisely distinguishes between control structures and function descriptions, accurately extracting and searching for relevant function implementations to effectively generate code snippets for complex software requirements. Additionally, the intermediary representation proposed can not only be transformed into code snippets for various programming languages but also can be converted into UML activity diagrams, providing developers with a more intuitive method for code review.

In the future, we should focus on enhancing the accuracy and reliability of our strategy. This involves further refining the search algorithms and optimizing the integration of external code repositories to ensure that the retrieved functions are the most suitable for



```
if __name__ == "__main__":
        create_database(self, instance_id, database_id, ddl_statements, project_id=None)
        get_pandas_df(self, hql, schema='default')

        if the query results are not empty:
                write_object_to_file(self,query_results,filename,fmt="csv",coerce_to_timestamp=False,
                                                      record_time_added=False)
        else:
                while True:
                        if there are no more databases in Cloud SQL:
                                break
                        delete_database(self, instance_id, database_id, project_id=None)
```

```
create_database;get_pandas_df;if(write_object_to_file;)else(while(delete_database;))
```

Fig. 13. An illustration of function retrieval limitations.

the given requirements. Additionally, implementing more sophisticated validation techniques that can automatically verify the correctness of the generated code before it reaches the developers will be crucial. Such validation could involve static analysis tools or automated testing frameworks.

Meanwhile, this study also presents a significant limitation. One significant limitation is that the current strategy only generates a rough outline of codes. It does not detail intricate elements such as the conditions for loops or branches, declarations of function parameters, or their returns. There is also an opportunity to include explanatory comments in the code. Therefore, future work should concentrate on employing additional AI techniques to enhance the utility of these code snippets. These could involve using code generation techniques to fill in missing details and refining the approach to handle a wider range of complex software requirements more effectively.

Moreover, our dataset was limited to using only Python, and the amount of data was not extensive. Expanding the dataset to include more diverse programming tasks and constructs could enable a more comprehensive evaluation of the methodology.

In conclusion, the Search-and-Fill strategy represents a promising advancement in the field of LCD, offering a pragmatic solution to the challenges of code reuse and the efficient handling of complex software requirements. By automating the process of generating code snippets, the strategy not only enhances the efficiency of software development but also contributes to the ongoing evolution of software engineering practices.

5. Threats to validity

Internal Validity: In the process of extracting control structures and atomic function points from requirements using the Search-and-Fill strategy, we relied on a set of predefined patterns and rules. These were formulated based on common expressions of requirements and programming practices, which may not encompass all potential ways of expressing requirements or programming habits. During the validation process, we employed a manually annotated dataset for verification. While the dataset includes a diverse array of control structures and randomly selected functions, the subjective nature of manual annotations might not fully capture the complexity and diversity present across the broader domain of software development.

External Validity: The conversion of complex software requirements into program control structures and atomic function points may not cover all aspects of complexity found in actual software development scenarios. The accuracy of generating code snippets largely depends on the quality of function calls available in external libraries and the precision with which large language models retrieve relevant code. If these external resources do not adequately reflect the diversity of real-world software requirements, the applicability of the Search-and-Fill strategy could be limited. Additionally, the dynamic nature of software development, characterized by evolving requirements and programming paradigms, poses a challenge to maintaining the strategy's relevance and effectiveness over time.

Construct Validity: This study's construct validity may be questioned regarding the accuracy of the metrics used to evaluate the strategy's performance. While the BLEU metric and TEDS metric are widely recognized and employed in the field of code, their ability to fully capture the nuances of code quality and functional correctness in the context of code snippets remains a subject for further investigation.

Addressing these threats involves ongoing refinement of the strategy, the inclusion of broader and more varied datasets, and the exploration of alternative evaluation metrics and statistical approaches that can more accurately reflect the effectiveness of the Search-and-Fill strategy in generating code snippets.

6. Related work

The evolution of AI in programming driven by natural language requirements has significantly reduced the reliance on manual coding, driving software development methods towards partial automation. This shift is profoundly influencing various stages of software development, marking a major transformation in digital and software development processes [17]. In the field of NLP, the introduction of pre-trained models represents a critical advancement, ushering in the era of code generation from natural language descriptions. The advent of transformer-based pre-trained models, such as BERT [18], has established a benchmark in achieving top results in numerous NLP tasks, highlighting AI's potential in understanding and interpreting programming languages.

The exploration of pre-trained models' effects on source code has led to the development of several influential models. For instance, Feng et al. [19] introduced CodeBERT, which integrates contrastive learning methods with BERT's training on code texts, thus improving the model's efficiency in code-related tasks. Similarly, Kanade et al. [20] focused on training BERT models with Python code, resulting in the CuBERT model, while Guo et al. [12] enhanced the Transformer's capabilities in pre-training code by incorporating code structures into the training process, producing the GraphCodeBERT model. The effectiveness of these models in code understanding tasks was further confirmed by Ishtiaq et al. [21], demonstrating the strong performance of transformer-based pre-trained language models in code comprehension.

With the rapid advancement of large pre-trained language model technologies, methods for code generation and understanding based on these models have made significant strides. Nijkamp et al. [22] open-sourced CodeGEN, a large model with 16.1 billion parameters, introducing a conversational program generation method where users provide natural language inputs in multiple rounds and receive system feedback, culminating in code generation. Experimental results showed that this multi-turn dialogue approach effectively enhances code generation performance. Zan et al. [23] proposed the CERT model, which uses a continual pre-training method based on sketches to solve library-oriented code generation problems, dividing the task into generating code templates and filling in details according to those templates, significantly boosting performance in library-oriented code generation. Fried et al. [24] introduced InCoder, the first large-scale code generation model capable of filling any code region, which greatly improved code generation performance through random masking and bidirectional prediction. Models like CodeT5 [25] and CoTexT [26], which utilize the Transformer's encoder–decoder structure, especially emphasize learning from identifiers extracted from code segments during their pre-training phase, treating code generation and summarization as dual tasks using <Text, Code> dual-modal data, thereby enhancing performance in various software engineering tasks. Guo et al. [27] proposed the UniXcoder model, which is compatible with encoder–decoder, encoder-only, and decoder-only architectures. It utilizes code semantic information for multi-modal contrastive learning and cross-modal generation, significantly enhancing code comprehension. Moreover, several large pre-trained code generation models such as OpenAI's CodeX [7], DeepMind's AlphaCode [28], Huawei's PanGu-Coder [29], and Tsinghua University's CodeGeeX [13] have been launched, demonstrating robust code generation performance in practical applications, providing convenience for developers. Moreover, Google's Gemini 1.5 Pro [14] is Google's best performing model in code to date, showcasing its capability in handling a variety of coding tasks with advanced techniques. In March 2023, OpenAI unveiled GPT-4, a model distinguished by its superior reasoning and contextual learning capabilities. [6] Experimental results showing that GPT-4 further enhances zero-shot generation capabilities. Overall, these efforts have achieved important progress in large-scale code data pre-training, conversational program generation, fill-in code generation, and multi-task learning, continually advancing technologies related to large language models for code.

However, these models primarily excel in understanding rather than generating code, with their application in code generation revealing limitations due to the expansive solution space [7]. Despite advancements in these models, they face limitations related to processing length, which hamper their ability to effectively utilize extensive code repositories. This limitation is especially problematic when addressing complex software requirements, as holistic approaches employed by these models do not align well with traditional modular programming strategies, which are typically more effective for such tasks. Simultaneously, the vast number of code snippets available on the internet means that other developers have likely had similar needs for the majority of user requirements.

Coding, inherently an open-domain problem, necessitates the integration of external knowledge bases and previous works to supplement the original training models, a concept that diverges from the closed-book task approach solely reliant on training data patterns [30]. Hayati and colleagues [31] pioneered the integration of retrieval technologies into code generation with the introduction of the RECODE model. Following this innovation, Xu et al. [32] leveraged the TranX model [33], enriching it with external data from Stack Overflow and API Documentation to mitigate the shortage of annotated code data. They devised a distribution of real-world <Text, Code> pairs, using a methodology similar to Yin et al. [34] for harvesting data from Stack Overflow, and refined the API Documentation data to more accurately mirror real-world coding scenarios. This method facilitated an effective pre-training regime, subsequently fine-tuned on the CoNaLa dataset, markedly enhancing the model's capacity for code generation tasks. Additionally, Zhou et al. [9] unveiled DocPrompting, an innovative strategy for updating a dynamic documentation library with previously unseen code, further demonstrating the value of integrating external knowledge into AI models for programming applications.

Research shows that the complexity and length of programming tasks significantly hinder the effectiveness of automatic code generation systems, making it challenging to produce high-quality code from complex requirements [35]. Due to the frequent occurrence of errors in code generated by these systems, developers often hesitate to use it, noting that correcting such inaccuracies typically takes more time than manual coding.

In response to these impediments, employing a strategy focused on generating specific code elements, such as API sequences, has been identified as a more efficacious alternative compared to attempts at producing complete programs [10]. This approach not only augments the predictability of the code generation process but also equips developers with more practical code elements, thereby facilitating subsequent development tasks and enhancing overall productivity. However, this strategy limits the scope of code reuse, potentially leading to underutilization of existing comprehensive codebases.

Against this backdrop, our research introduces the Search-and-Fill strategy, engineered to refine the code snippet generation process by adeptly navigating complex, multifunctional, and multi-control structure requirements. This strategy synthesizes the use of large-scale models for external library retrieval with a meticulous focus on the precise integration of single-function implementations. Such a strategy markedly elevates development efficiency amidst complex software requirements, substantially diminishing both development time and workload, while simultaneously safeguarding code quality and adaptability.

7. Summary and prospect

This paper presents the Search-and-Fill strategy, designed to improve software development efficiency by generating code snippets from complex software requirements. Utilizing NLP and pre-trained models, the Search-and-Fill strategy analyzes complex software requirements, synthesizing control structures and atomic function points into comprehensive code snippets. Our experiments with the ACRD and BCRD datasets demonstrate the strategy's effectiveness; it showcases a novel strategy to combining code retrieval and modification, offering advancements in LCD for complex tasks.

Future work on the Search-and-Fill strategy aim to deeper its understanding of complex software requirements and addressing current limitations in code snippets generation. This includes generating detailed code elements like loop conditions and function parameters, and integrating explanatory comments for better clarity. Advanced AI techniques will be employed to enhance the precision of code retrieval and integration, making the code snippets more accurate and adaptable to diverse tasks. Additionally, expanding the dataset to include a wider range of programming languages and constructs will allow for a more robust evaluation of the methodology. Continuous updates to the algorithms will ensure the strategy stays relevant and effective in evolving programming environments, ultimately streamlining software development through efficient code reuse.

CRediT authorship contribution statement

Yukun Dong: Writing – review & editing, Validation, Supervision, Funding acquisition. **Lingjie Kong:** Writing – original draft, Methodology, Data curation, Conceptualization. **Lulu Zhang:** Writing – review & editing, Validation. **Shuqi Wang:** Writing – review & editing, Validation. **Xiaoshan Liu:** Writing – review & editing, Validation. **Shuai Liu:** Writing – review & editing, Validation. **Mingcheng Chen:** Writing – review & editing.

Declaration of competing interest

We declare that we have no financial and personal relationships with other people or organizations that can inappropriately influence our work, and there is no professional or other personal interest of any nature or kind in any product, service, and/or company that could be construed as influencing the position presented in, or the review of, the manuscript entitled.

[6] https://openai.com/research/gpt-4.

Data availability

The data that support the findings of this study are openly available at https://doi.org/10.5281/zenodo.13292164.

Acknowledgments

This work was supported by the Shandong Provincial Natural Science Foundation (ZR2021MF058).

References

[1] C. Richardson, J.R. Rymer, C. Mines, A. Cullen, D. Whittaker, New Development Platforms Emerge for Customer-Facing Applications, Vol. 15, Forrester, Cambridge, MA, USA, 2014.
[2] Gartner magic quadrant for low-code application platforms, 2023, https://www.mendix.com/resources/gartner-magic-quadrant-for-low-code-application-platforms/.
[3] R. Potvin, J. Levenberg, Why google stores billions of lines of code in a single repository, Commun. ACM 59 (7) (2016) 78–87.
[4] Y. Wang, H. Le, A.D. Gotmare, N.D. Bui, J. Li, S.C. Hoi, Codet5+: Open code large language models for code understanding and generation, 2023, arXiv preprint arXiv:2305.07922.
[5] R. Li, L.B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, et al., Starcoder: may the source be with you!, 2023, arXiv preprint arXiv:2305.06161.
[6] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, D. Jiang, Wizardcoder: Empowering code large language models with evol-instruct, 2023, arXiv preprint arXiv:2306.08568.
[7] M. Chen, J. Tworek, H. Jun, Q. Yuan, H.P.d.O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al., Evaluating large language models trained on code, 2021, arXiv preprint arXiv:2107.03374.
[8] T.B. Hashimoto, K. Guu, Y. Oren, P.S. Liang, A retrieve-and-edit framework for predicting structured outputs, Adv. Neural Inf. Process. Syst. 31 (2018).
[9] S. Zhou, U. Alon, F.F. Xu, Z. Wang, Z. Jiang, G. Neubig, Docprompting: Generating code by retrieving the docs, 2022, arXiv preprint arXiv:2207.05987.
[10] J. Martin, J.L. Guo, Deep api learning revisited, in: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, 2022, pp. 321–330.
[11] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, M. Brockschmidt, Codesearchnet challenge: Evaluating the state of semantic code search, 2019, arXiv preprint arXiv:1909.09436.
[12] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, et al., Graphcodebert: Pre-training code representations with data flow, 2020, arXiv preprint arXiv:2009.08366.
[13] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, Z. Wang, L. Shen, A. Wang, Y. Li, et al., Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x, 2023, arXiv preprint arXiv:2303.17568.
[14] M. Reid, N. Savinov, D. Teplyashin, D. Lepikhin, T. Lillicrap, J.-b. Alayrac, R. Soricut, A. Lazaridou, O. Firat, J. Schrittwieser, et al., Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context, 2024, arXiv preprint arXiv:2403.05530.
[15] K. Papineni, S. Roukos, T. Ward, W.-J. Zhu, Bleu: a method for automatic evaluation of machine translation, in: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, 2002, pp. 311–318.
[16] X. Zhong, E. ShafieiBavani, A. Jimeno Yepes, Image-based table recognition: data, model, and evaluation, in: European Conference on Computer Vision, Springer, 2020, pp. 564–580.
[17] Z. Yan, The impacts of low/no-code development on digital transformation and software development, 2021, arXiv preprint arXiv:2112.14073.
[18] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, Bert: Pre-training of deep bidirectional transformers for language understanding, 2018, arXiv preprint arXiv:1810.04805.
[19] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al., Codebert: A pre-trained model for programming and natural languages, 2020, arXiv preprint arXiv:2002.08155.
[20] A. Kanade, P. Maniatis, G. Balakrishnan, K. Shi, Learning and evaluating contextual embedding of source code, in: International Conference on Machine Learning, PMLR, 2020, pp. 5110–5121.
[21] A.A. Ishtiaq, M. Hasan, M.M.A. Haque, K.S. Mehrab, T. Muttaqueen, T. Hasan, A. Iqbal, R. Shahriyar, Bert2code: Can pretrained language models be leveraged for code search? 2021, arXiv preprint arXiv:2104.08017.
[22] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, C. Xiong, Codegen: An open large language model for code with multi-turn program synthesis, 2022, arXiv preprint arXiv:2203.13474.
[23] D. Zan, B. Chen, D. Yang, Z. Lin, M. Kim, B. Guan, Y. Wang, W. Chen, J.-G. Lou, CERT: continual pre-training on sketches for library-oriented code generation, 2022, arXiv preprint arXiv:2206.06888.
[24] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, M. Lewis, Incoder: A generative model for code infilling and synthesis, 2022, arXiv preprint arXiv:2204.05999.
[25] Y. Wang, W. Wang, S. Joty, S.C. Hoi, Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, 2021, arXiv preprint arXiv:2109.00859.
[26] L. Phan, H. Tran, D. Le, H. Nguyen, J. Anibal, A. Peltekian, Y. Ye, Cotext: Multi-task learning with code-text transformer, 2021, arXiv preprint arXiv:2105.08645.
[27] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, J. Yin, Unixcoder: Unified cross-modal pre-training for code representation, 2022, arXiv preprint arXiv:2203.03850.
[28] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, et al., Competition-level code generation with alphacode, Science 378 (6624) (2022) 1092–1097.
[29] F. Christopoulou, G. Lampouras, M. Gritta, G. Zhang, Y. Guo, Z. Li, Q. Zhang, M. Xiao, B. Shen, L. Li, et al., Pangu-coder: Program synthesis with function-level language modeling, 2022, arXiv preprint arXiv:2207.11280.
[30] D. Drain, C. Hu, C. Wu, M. Breslav, N. Sundaresan, Generating code with the help of retrieved template functions and stack overflow answers, 2021, arXiv preprint arXiv:2104.05310.
[31] S.A. Hayati, R. Olivier, P. Avvaru, P. Yin, A. Tomasic, G. Neubig, Retrieval-based neural code generation, 2018, arXiv preprint arXiv:1808.10025.
[32] F.F. Xu, Z. Jiang, P. Yin, B. Vasilescu, G. Neubig, Incorporating external knowledge through pre-training for natural language to code generation, 2020, arXiv preprint arXiv:2004.09015.
[33] P. Yin, G. Neubig, TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation, 2018, arXiv preprint arXiv:1810.02720.
[34] P. Yin, B. Deng, E. Chen, B. Vasilescu, G. Neubig, Learning to mine aligned code and natural language pairs from stack overflow, in: Proceedings of the 15th International Conference on Mining Software Repositories, 2018, pp. 476–486.
[35] H. Liu, M. Shen, J. Zhu, N. Niu, G. Li, L. Zhang, Deep learning based program generation from requirements text: Are we there yet? IEEE Trans. Softw. Eng. 48 (4) (2020) 1268–1289.