LLM-Based Method Name Suggestion with Automatically Generated Context-Rich Prompts

WASEEM AKRAM, Beijing Institute of Technology, China
YANJIE JIANG*, Peking University, China
YUXIA ZHANG, Beijing Institute of Technology, China
HARIS ALI KHAN, Beijing Institute of Technology, China
HUI LIU*, Beijing Institute of Technology, China

Accurate method naming is crucial for code readability and maintainability. However, manually creating concise and meaningful names remains a significant challenge. To this end, in this paper, we propose an approach based on Large Language Model (LLMs) to suggest method names according to function descriptions. The key of the approach is *ContextCraft*, an automated algorithm for generating context-rich prompts for LLM that suggests the expected method names according to the prompts. For a given query (functional description), it retrieves a few best examples whose functional descriptions have the greatest similarity with the query. From the examples, it identifies tokens that are likely to appear in the final method name as well as their likely positions, picks up pivot words that are semantically related to tokens in the according method names, and specifies the evaluation results of the LLM on the selected examples. All such outputs (tokens with probabilities and position information, pivot words accompanied by associated name tokens and similarity scores, and evaluation results) together with the query and the selected examples are then filled in a predefined prompt template, resulting in a context-rich prompt. This context-rich prompt reduces the randomness of LLMs by focusing the LLM's attention on relevant contexts, constraining the solution space, and anchoring results to meaningful semantic relationships. Consequently, the LLM leverages this prompt to generate the expected method name, producing a more accurate and relevant suggestion. We evaluated the proposed approach with 43k real-world Java and Python methods accompanied by functional descriptions. Our evaluation results suggested that it significantly outperforms the state-of-the-art approach *RNN-att-Copy*, improving the chance of exact match by 52% and decreasing the edit distance between generated and expected method names by 32%. Our evaluation results also suggested that the proposed approach worked well for various LLMs, including ChatGPT-3.5, ChatGPT-4, ChatGPT-4o, Gemini-1.5, and Llama-3.

CCS Concepts: • **Software and its engineering → Development frameworks and environments**.

Additional Key Words and Phrases: Method Name, Large Language Model, Functional Description

ACM Reference Format:
Waseem Akram, Yanjie Jiang, Yuxia Zhang, Haris Ali Khan, and Hui Liu. 2025. LLM-Based Method Name Suggestion with Automatically Generated Context-Rich Prompts. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE036 (July 2025), 22 pages. https://doi.org/10.1145/3715753

*Corresponding author.

Authors' Contact Information: Waseem Akram, School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China, waseemakramcui@gmail.com; Yanjie Jiang, School of Computer Science, Peking University, Beijing, China, jiangyanjiese@gmail.com; Yuxia Zhang, School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China, yuxiazh@bit.edu.cn; Haris Ali Khan, School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China, hariskhatter@gmail.com; Hui Liu, School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China, liuhui08@bit.edu.cn.

ACM 2994-970X/2025/7-ARTFSE036
https://doi.org/10.1145/3715753

1 Introduction

The quality of method names is critical for the source code's readability and maintainability [33]. High-quality method names enhance comprehension and collaboration among developers, thereby reducing the cognitive load associated with understanding and modifying source code [9, 16, 25]. However, despite their importance, it is often challenging and time-consuming for software engineers, especially inexperienced ones, to manually construct a concise and meaningful high-quality method name from scratch. Ideally, method names should be both concise and expressive, which significantly increases the difficulty in method name construction. Additionally, developers' linguistic backgrounds [19, 28] could make this task even harder. As a result, manual naming can be inconsistent and error-prone.

To facilitate the naming of methods, many approaches have been proposed to suggest method names. The first category of such approaches suggests method names according to the implementation of methods [28][5][6]. Such approaches have significantly improved method naming. However, these approaches require method bodies to be implemented before naming. The second category of such approaches suggests method names according to functional descriptions of the methods [23]. Notable, describing the functionality of methods in natural language, e.g., English, is much easier than coining concise, meaningful, and well-formed method names because the functional descriptions have little limitation on their length and few constraints on their structures. In contrast, method names have much stronger limitations and constraints concerning their length, structures, semantics, and even spelling [19, 28]. Consequently, it is beneficial if developers specify only functional descriptions and automated approaches can generate high-quality method names. Although a few approaches have been proposed to generate method names according to functional descriptions, their performance deserves further improvement, as suggested in the evaluation section.

Advances in large language models (LLMs), such as GPT-3 and GPT-4, make them promising tools for natural language processing tasks [1, 8]. However, fine-tuning the LLMs for specific tasks poses computational [41] and overfitting challenges, which can potentially lead to biased performance assessments [17, 46]. Although effective prompts with high-quality examples can mitigate these issues, LLMs often struggle with the complexities of natural language prompts [53, 55], such as contextual differences, linguistic variations, and semantic nuances, which can hinder their performance. Non-expert users, in particular, may find it difficult to craft effective prompts and often lack the necessary feedback for refinement. It is also tricky to dynamically select the appropriate few-shot examples based on input descriptions [22, 29, 52]. Prompt engineering has emerged as a crucial technique to address these challenges of LLMs, enabling users to craft more effective prompts that yield better results from LLMs [15, 30, 47].

To address these challenges, this paper introduces the first LLM-based approach to suggesting method names according to function descriptions. The key of the approach is an automated algorithm called *ContextCraft* that generates context-rich prompts according to a given query (functional description of a method). The context-rich prompt is fed into an LLM to generate the expected method names. For a given query, it retrieves a few best examples from the corpus according to their semantic similarity with the given query. From the retrieved best examples, it identifies tokens that are likely to appear in the final method name and their most likely appearing position, e.g., the first, middle or last token of the method name. It also picks up pivot words, i.e., tokens in the functional descriptions that are semantically similar yet lexically distinct from the tokens in the corresponding method names. Once presented in the prompt, such pivot words and their associated method name tokens may help the LLMs generate the associated method name tokens. Besides, *ContextCraft* evaluates the selected LLM with the retrieved best examples and generates

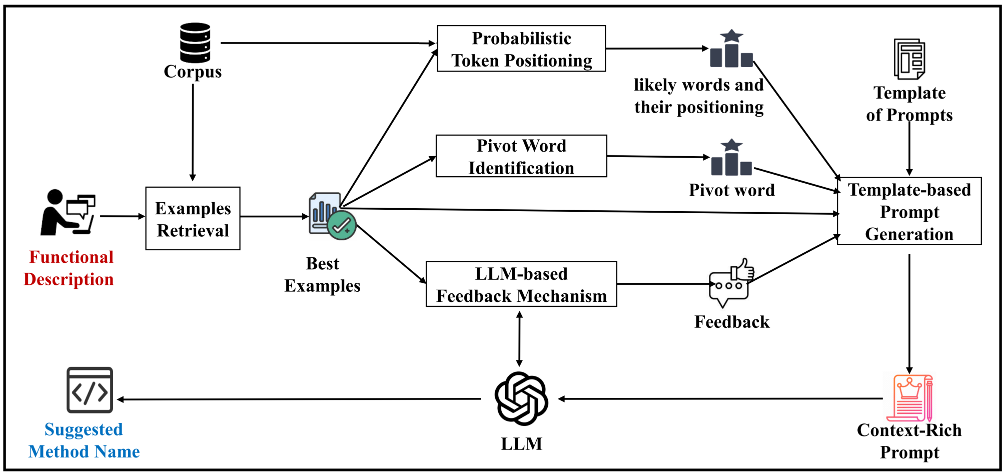

Corpus
Probabilistic Token Positioning
likely words and their positioning
Template of Prompts
Pivot Word Identification
Pivot word
Template-based Prompt Generation
Functional Description
Examples Retrieval
Best Examples
LLM-based Feedback Mechanism
Feedback
Suggested Method Name
LLM
Context-Rich Prompt

Fig. 1. Overview of the Proposed Approach

feedback specifying how the generated names differ from the expected ones. Such feedback may help LLMs understand their divergence limitations [7]. All such outputs (i.e., tokens with their probabilistic positions, pivot words accompanied by associated name tokens and similarity scores, and evaluation feedback) together with the query and the retrieved examples are then filled in a predefined prompt template, resulting in a context-rich prompt. This context-aware prompt helps to reduce the randomness of LLMs by guiding the LLM to focus on relevant information. Narrowing down the possible solutions and anchoring the results to meaningful semantic relationships enables the LLM to generate method names that are more accurate and relevant[32, 34]. The LLM then leverages the resulting prompt to generate the expected method name. We evaluated the proposed approach with 43,000 real-world Java and Python methods accompanied by functional descriptions. Our evaluation results suggested that it significantly improved the state of the art by increasing the chances of an exact match by 52%. Our evaluation results also indicated that the proposed approach worked well with various LLMs, including ChatGPT-3.5, ChatGPT-4, ChatGPT-4o, Gemini-1.5, and Llama-3. It also demonstrated strong performance with both Java and Python. This paper makes the following contributions:

- A novel LLM-based approach to suggesting method names according to functional descriptions. The key of the approach is an automated algorithm to generate context-rich prompts, which serves as the deep customization of LLMs for the given task. The proposed approach substantially enhances the performance of state-of-the-art LLMs without the need for costly fine-tuning or the involvement of domain experts. To the best of our knowledge, our approach is the first LLM-based approach in this line.
- Implementation and evaluation of the proposed approach. The datasets, implementation, and replication package are publicly available [12].

2 Approach

2.1 Overview

An overview of the proposed approach is presented in Fig. 1. Overall, it works as follows:

- For a given functional description fd, it retrieves a set of top examples from a corpus whose functional descriptions are highly similar to fd.
- The selected examples undergo three processes: Probabilistic Token Positioning (PTP), Pivot Word Identification (PWI), and LLM-based Feedback Mechanism (LFM). PTP identifies the most likely prefixes, infixes, and suffixes (of the expected method name) from functional descriptions; PWI returns pivotal tokens from functional descriptions that are highly important for method name generation; and the LFM evaluates the LLM with the retrieved examples, generating a textual message to explain to what extent the generated names are different from the expected ones.
- The best examples and output from the preceding processes are fed into a predefined structured prompt template, resulting in a context-rich prompt.
- The resulting prompt is fed into the selected LLM to generate method name as requested.

2.2 Example Retrieval

For a given functional description (fd), example retrieval is to retrieve a few closely related examples from a predefined corpus. The key of the retrieval is the measurement of contextual similarity between the query (i.e., fd) and the examples in the corpus. Notably, every example e in the corpus comprises two parts: The functional description of a method (denoted as $D(e)$) and the name of the method (denoted as $N(e)$). While computing the similarity between fd and e, we only compare $D(e)$ against fd because both are functional descriptions:

$$S(fd, e) = TS(fd, D(e)) \tag{1}$$

where $S(fd, e)$ is the similarity between the input query fd and the example e from the corpus. $TS(fd, D(e))$ is the text similarity between two functional descriptions fd and $D(e)$. The Text similarity is computed using cosine similarity between vector representations of fd and $D(e)$ as follows:

$$\begin{aligned} TS(fd, D(e)) &= \cos(vec(fd), vec(D(e))) \\ &= \frac{vec(fd) \cdot vec(D(e))}{\|vec(fd)\| \|vec(D(e))\|} \end{aligned} \tag{2}$$

where $vec(fd)$ and $vec(D(e))$ are the vector representations of fd and $D(e)$, respectively. "$\|\|$" denotes the magnitude of the vectors and "$\cdot$" denotes the dot product. To compute the similarity between fd and $D(e)$, we convert the tokens into fixed-length vectors using different embedding models, including text-embedding-ada-003 [39], All-mpnet-base-v2 [43], GraphCodeBERT [27], CodeBERT [20], and RoBERTa [35]. We individually evaluate these models to determine which improves the overall results of ContextCraft, as detailed in Section 3.5. The retrieved examples are leveraged to generate context-rich prompts. To account for the limited context window size of LLMs, we limit retrieved examples to the top ten based on similarity, as using more examples increases prompt length. However, the setting is subject to the changes of employed LLMs.

2.3 Probabilistic Token Positioning (PTP)

Method names are concise, so only a subset of tokens from the functional description is used in the method names. Consequently, it would be beneficial to know how likely a token in functional descriptions may appear in the corresponding method names. Method names follow patterns; therefore, the probability of a token appearing in a specific position (prefix, infix, or suffix) varies. Consequently, we compute how likely a given token (from functional descriptions) would appear in different positions of method names to model such position-specific probability. Details of the process are presented in the following paragraphs.

Algorithm 1 Ranking Algorithm for Probabilistic Token Positioning

```
1: procedure PTPRanking($bestExamples$)
2:     $PTP = \emptyset$                                   ▷ Initialize the output parameter
3:     for each $example$ in $bestExamples$ do                   ▷ Iteration on each example
4:         $singPTP$ =Ranking4SingleExample( $example$)       ▷ Invoke $Ranking4SingleExample$
5:         $PTP$.append($singPTP$)       ▷ Selected tokens and their position info add into $PTP$
6:     end for
7:     return $PTP$
8: end procedure
9: procedure Ranking4SingleExample($example$)
10:    $maxPrefixProb$=0                                  ▷ Initialize Probability variables
11:    $maxInfixProb$=0
12:    $maxSuffixProb$=0
13:    $Prefix$=null
14:    $Infix$=null
15:    $Suffix$=null
16:    $descTokenList$=Tokens($example.functionalDescription$)      ▷ Tokenize the description
17:    for each $token$ in $descTokenList$ do                 ▷ Iteration on each unique token
18:        $token.P_{prefix}, token.P_{infix}, token.P_{suffix}=findPositionalProb(token)$
19:        if $token.P_{prefix} > maxPrefixProb$ then              ▷ Find highest prefix prob.
20:            $maxPrefixProb = token.P_{prefix}$
21:            $Prefix=token$
22:        end if
23:        if $token.P_{infix} > maxInfixProb$ then                ▷ Find highest infix prob.
24:            $maxInfixProb = token.P_{infix}$
25:            $Infix=token$
26:        end if
27:        if $token.P_{suffix} > maxSuffixProb$ then              ▷ Find highest suffix prob.
28:            $maxSuffixProb = token.P_{suffix}$
29:            $Suffix=token$
30:        end if
31:    end for
32:    $singPTP$=<$Prefix, maxPrefixProb, Infix, maxInfixProb, Suffix, maxSuffixProb$>
33:    return $singPTP$   ▷ Return tokens with the highest prefix, infix, and suffix prob. in name
34: end procedure
```

We retrieve all texts from the given corpus of examples (i.e., pairs of <functional description, method name>) and decompose them into token sequences. Functional descriptions are split by whitespace or punctuation, and method names are split by camelCase or underscores based on programming language.

For each unique token, we calculate its likelihood of appearing as a prefix, infix, or suffix in the method name from the training dataset (detailed in Section 3.3). The probability is defined and computed as follows:

- **Prefix probability**, noted as $P_{\text{prefix}}(t)$, represents the likelihood of a given token t (from a functional description) appearing as the first token in the training dataset's method names and also existing in their functional descriptions:

$$P_{\text{prefix}}(t) = \frac{\text{Occurrences of } t \text{ as prefixes of names}}{\text{Occurrences of } t \text{ in descriptions}} \tag{3}$$

- **Infix probability**, noted as $P_{\text{infix}}(t)$, represents the likelihood of a given token t appearing in the middle part of the method names:

$$P_{\text{infix}}(t) = \frac{\text{Occurrences of } t \text{ in the middle of names}}{\text{Occurrences of } t \text{ in descriptions}} \tag{4}$$

- **Suffix probability**, noted as $P_{\text{suffix}}(t)$, represents the likelihood of a given token t appearing as the last token of the method names:

$$P_{\text{suffix}}(t) = \frac{\text{Occurrences of } t \text{ as suffixes of names}}{\text{Occurrences of } t \text{ in descriptions}} \tag{5}$$

With the preceding computation, we create a mapping of $M : t \rightarrow < P_{\text{prefix}}(t), P_{\text{infix}}(t), P_{\text{suffix}}(t) >$ that maps each unique token in functional descriptions into a sequence of probabilities. With the mapping M, we rank the unique tokens in the functional descriptions of the selected best examples. The ranking is presented in Algorithm 1. This algorithm takes the *bestExamples* as input. On line 2, it initializes the *PTP*, which will be returned as the final output. The algorithm enumerates each element in *bestExamples* with another procedure *Ranking4SingleExample* (Lines 3–4). This procedure initializes a sequence of variables (Lines 10–15) that should be returned. The algorithm retrieves a list of unique tokens, *descTokenList*, from the description in the given example on line 16. *FOR* iteration on lines 17–31 is the major body of the algorithm, where each iteration handles a unique token in *descTokenList*. Line 18 calculates the prefix, infix and suffix probabilities of the *token* from the training dataset. Line 19 compares the *prefix probability* of the token, i.e., $P_{\text{prefix}}(token)$ against *maxPrefixProbability*, to validate whether its prefix probability is greater than any of the enumerated tokens. If yes, the algorithm temporarily marks it as the to-be-returned token by assigning it to *maxPrefixProbability* on line 20. It also records its prefix token by assigning it to the variable *Prefix* on line 21. The *IF* statement on lines 23-26 validates whether the current *token* has the maximal *infix probability* in the same way, whereas the validation of maximal suffix probability is accomplished by the *IF* statement on lines 27-30. Finally, on line 32, the algorithm returns three tokens (i.e., *Prefix*, *Infix*, and *Suffix*) with the greatest probability of appearing as the method name's prefix, infix, and suffix, respectively. It also returns the corresponding probability scores. This output is appended to *PTP* on line 5, and the resulting *PTP* is returned as the final output after all examples are enumerated by *Ranking4SingleExample*.

2.4 Pivot Word Identification (PWI)

While PTP identifies tokens for direct copying, some functional description words aid in name generation without direct copying. We propose a dedicated process called pivot word identification (PWI) to identify such valuable words. The rationale of PWI is that if a word (w) in the functional description is semantically similar to words (nw) in the corresponding method name but not an exact copy, this word (i.e., w) could be employed by LLMs to generate the associated name tokens (i.e., nw).

Algorithm 2 outlines PWI, which takes best examples as input (i.e., parameter *bestExamples*) and returns a list of pivot words (pair of method name's token and description's token with their similarity score) for each example. First, it initializes the output parameter *PWI* with an empty set on line 2. The primary body of the algorithm is an iteration on each of the examples (i.e., the FOR statement on lines 3-16). Each iteration retrieves the tokens in the functional description (Line 4) and method name (Line 5). On line 6, initializes *PWL* as a list of pivot words of an example. The

Algorithm 2 Pivot Word Identification (PWI)

```
1:  procedure PivotWordIdentification($bestExamples$)
2:      Initialize $PWI$ as empty structure          ▷ To store examples' pivot words with score
3:      for each $example$ in $bestExamples$ do
4:          $descriptionTokens$ = tokenizer($example.description$)      ▷ Tokenize the description
5:          $methodNameTokens$ = tokenizer($example.methodName$)      ▷ Tokenize the name
6:          Initialize $PWL$ as empty structure          ▷ List of pivot words
7:          for each $m_t$ in $methodNameTokens$ do
8:              for each $d_t$ in $descriptionTokens$ do
9:                  if $m_t \neq d_t$ then          ▷ Ensure tokens are not the same
10:                     $similarity = \text{Sim}(m_t, d_t)$          ▷ Calculate similarity between tokens
11:                     if $similarity \geq threshold$ then
12:                         $PWL.Append(m_t, d_t, similarity)$
13:                     end if
14:                 end if
15:             end for
16:         end for
17:         $PWI.Append(example, PWL)$
18:     end for
19:     return $PWI$
20: end procedure
```

nested loop on lines 7 and 8 takes each token of the method name noted as m_t and of description as d_t and ensure tokens are not the same on line 9. If they are not, the algorithm passes them to a function *Sim* on line 10. Using the embedding model, this function calculates the cosine similarity between m_t and d_t and stores the result in the variable *similarity*. On line 12, It appends m_t, d_t and *similarity* into the list PWL if they have a *similarity* score greater than or equal to *threshold*. We set a threshold of 0.5 for semantic similarity to achieve a balance between precision and recall, ensuring that only moderately similar token pairs are identified while minimizing noise. Once all pivot words of an example are appended into PWL, it appends the *example* and corresponding PWL into PWI on line 17 and returns this PWI on line 19. On line 10, the algorithm employs $similarity = \text{Sim}(m_t, d_t)$ to calculate the cosine similarity between two tokens m_t and d_t. We convert the tokens into fixed-length vectors using a text embedding model (i.e.,text-embedding-ada-003, OpenAI's next-generation embedding model at the time of selection [39]) to compute the similarity. With the resulting embedding, we measure the similarity between m_t and d_t by computing the cosine similarity between their corresponding vectors. To demonstrate the application of the PWI, there are awesome examples:

- Example 1: <Functional description: *"Persist the user's configuration preferences to ensure changes are not lost."*, Method name:*"saveSettings"*>
- Example 2: <Functional description: *"Verify if the current user has the authority to access the requested resource."*, Method name: *"checkAccess"*>
- Example 3: <Functional description: *"Create a copy of this object."*, Method name: *"clone"*>

For the first example, the semantic similarities between functional descriptions and method names reveal notable correlations: the word "*save*" from the method name "*saveSettings*" achieves a similarity score of 0.564 with "*Persist*" from the description as both terms imply the action of data maintaining or conserving. Additionally, "*settings*" matches with "*preferences*" with a similarity score of 0.590, suggesting a strong relation since both terms refer to user-configurable parameters.

For the second example, the semantic analysis between the functional description and method name vividly illustrates the contextual alignment of the terms used: "*check*" from the method name "*checkAccess*" shows a significant semantic correspondence with "*verify*" from the description, achieving a similarity score of 0.623, as both terms are commonly employed to confirm or ascertain validity. Furthermore, the word "*access*" is identically used in both contexts with a similarity score of 0.580. Notably, the similarity between two occurrences of "*access*" is not one because they have different contexts and thus have different representing vectors.

For the third example, the semantic similarity analysis reveals a notable connection: "*clone*" in the method name corresponds with "*object*" from the functional description with a similarity score of 0.618. This is logical because "*clone*" in source code usually refers to the action of creating an exact copy of an "*object*". This example highlights how semantic similarity analysis can effectively establish a connection between a method's action and its target, even when the direct linguistic connection seems obscure (i.e., the two words are not lexically similar).

2.5 LLM-Based Feedback Mechanism (LFM)

PTP and PWI focus on token identification, while LFM offers quantitative feedback by evaluating LLM performance with the best examples. Unlike token-based processing, the LFM proposed in this section tries to help by evaluating the to-be-employed LLM (i.e., ChatGPT-4o as it presents the OpenAI's latest LLM [38]) with the retrieved best examples and providing quantitative feedback on the evaluation. For each example, this process works as follows:

- It feeds the LLM a prompt with a functional description from the example and asks it to generate a method name according to the functional description.
- The LFM compares the name generated by the LLM and the ground truth, i.e., the method name in the example. This comparison is based on character-level analysis, calculating the edit distance between the generated and the original method name.
- It generates message *msg* specifying how the generated names differ from the ground truth.

We take the following example to illustrate the process:

Example: <Function description: *"casts this class object to represent a subclass of the class represented by the specified object."*, Method name: *"asSubclass"*>. In this example, it requests the model to generate a method name with the following prompt:

> *"Suggest a Java method name according to the given functional description: casts this class object to represent a subclass of the class represented by the specified object."*

The model generates a method name "*castToSubclass*". By comparing it against the ground truth (i.e., "*asSubclass*"), our approach calculates the edit distance score as 4, indicating that there are 4 single-character edits required to transform "*castToSubclass*" into "*asSubclass*". Consequently, it generates the following messages as feedback:

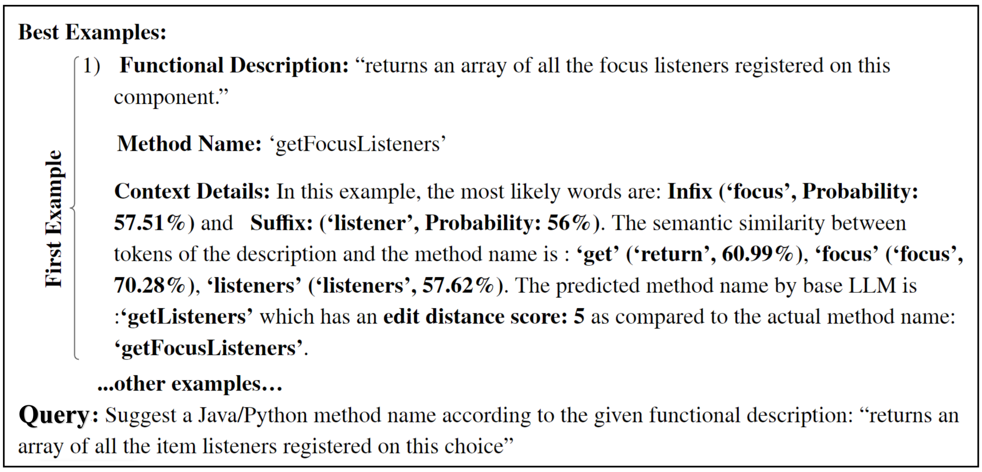
> *"The predicted method name by base LLM:'castToSubclass' which has an edit distance score: 04 as compared to the actual method name: 'asSubclass'"*

The LFM provides feedback on all retrieved top examples, which is then used in the next process.

2.6 Template-Based Prompt Generation

With the outputs from processes introduced in the preceding sections, we gathered all the information to create a context-rich prompt. The prompt is composed of five parts:

- Best examples retrieved from the corpus (copied from Section 2.1);



Best Examples:

First Example

1) **Functional Description:** "returns an array of all the focus listeners registered on this component."

Method Name: 'getFocusListeners'

Context Details: In this example, the most likely words are: **Infix ('focus', Probability: 57.51%)** and **Suffix: ('listener', Probability: 56%)**. The semantic similarity between tokens of the description and the method name is : **'get' ('return', 60.99%)**, **'focus' ('focus', 70.28%)**, **'listeners' ('listeners', 57.62%)**. The predicted method name by base LLM is :**'getListeners'** which has an **edit distance score: 5** as compared to the actual method name: **'getFocusListeners'**.

...other examples...

Query: Suggest a Java/Python method name according to the given functional description: "returns an array of all the item listeners registered on this choice"

Fig. 2. Example Prompt

- Probabilistic token positioning (copied from Section 2.3);
- Pivot words (copied from Section 2.4);
- LLM-based feedback (copied from Section 2.5);
- Query.

The preceding sections generated the best examples, probabilistic token positioning, pivot words, and LLM-based feedback. Together, PTP and PWI offer a balanced prompt where PTP handles token positioning and syntactic structure, and PWI ensures semantic relevance, improving overall flexibility and effectiveness in method name generation. Additionally, the LFM integrated into the prompt addresses inherent divergence in LLM behavior, ensuring that generated method names are both relevant and contextually aligned with input descriptions. The *query* comprises a predefined text (i.e., a command to the LLM) and the input of the proposed approach (i.e., the functional description). An example of a generated prompt by *ContextCraft* is presented in Figure 2.

2.7 LLM-Based Method Name Suggestion

The automatically generated query is fed into an LLM to generate the method name. In the default setting, we employ *ChatGPT-4o* as the LLM because it presents the state of the art at the time of selection. However, as validated in Section 3.7, the proposed approach works well with various LLMs, e.g., *Gemini-1.5* and *Llama-3*.

3 Evaluation

3.1 Research Questions

- ***RQ1:*** *Does the proposed approach improve the state of the art in functional description-based method name generation?*
- ***RQ2:*** *What is the impact of different embedding models on the performance of the proposed approach?*
- ***RQ3:*** *How do the different components of the proposed approach contribute to the overall performance?*
- ***RQ4:*** *How well does the proposed approach work with different large language models?*
- ***RQ5:*** *How accurate is the proposed approach on private projects?*

Table 1. Datasets

Metrics	*JavaData*	*PythonData*
Size	23,739	20,000
Average Length of Descriptions	12.85 tokens	18.51 tokens
Max Length of Descriptions	100 tokens	100 tokens
Average Length of Method Names	2.45 tokens	2.62 tokens
Max Length of Method Names	10 tokens	8 tokens
Programming Language	Java	Python
Natural Language	English	English

RQ1 concerns the proposed approach's performance compared to the state-of-the-art (SOTA) in functional description-based method name suggestion. RQ2 examines the impact of different embedding models while selecting the best examples, improving the proposed approach's overall performance. RQ3 examines the contribution of different components of the proposed approach and validates their importance and indispensability. Widely used ablation experiments should be able to answer this question. RQ4 investigates the generalizability of the proposed approach across different LLMs, such as *ChatGPT-4*, *ChatGPT-3.5*, *Llama-3*, and *Gemini-1.5*. RQ5 concerns the efficiency of the proposed approach on private projects, which may also reveal the impact of potential data contamination of the underlying LLMs.

3.2 Baselines

We considered six baselines that could generate method names from functional descriptions.

- Deep learning-based baselines: We considered *RNN-att-Copy*[23], CodeBERT [20] and UniXcoder [26] as baselines. To the best of our knowledge, *RNN-att-Copy* is the latest approach specifically designed for this purpose, and thus, it serves as our primary baseline. Since its original implementation is not publicly available, we reproduced the RNN-att-Copy for both datasets (Java and Python), and the implementation is publicly available at [14] . CodeBERT and UniXcoder are widely used to suggest code elements [10, 36, 45], and they can potentially suggest method names according to functional description [50] . Consequently, we also take these models as additional baselines, as [50] did and fine-tuned them using functional descriptions as input and method names as output.
- LLM-based baselines: We considered ChatGPT-4o [38], Llama-3 [18] and Gemini-1.5 [44] as baselines. Although our approach is the first LLM-based approach specially designed for this purpose, LLMs without customization also have the potential for this task [54]. Thus we take them as additional baselines. Notably, we selected the latest version of the LLMs available at the time of selection. By comparing the proposed approach against such LLMs, we may reveal to what extent the customization proposed in this paper can improve the underlying LLMs.

3.3 Datasets and Performance Metrics

An overview of the datasets used in the review is presented in TABLE 1. As shown in the table, we employed two datasets, *JavaData* and *PythonData*, comprising 43k items in total. The JavaData was created by Gao et al. [23]. However, there is still a lack of Python datasets for this purpose. To this end, we followed the methodology outlined by Gao et al. [23] to construct a Python dataset

Table 2. Improving State of the Art

Types	**Approaches**	*JavaData*			*PythonData*		
		EM	**SSI**	**ED**	**EM**	**SSI**	**ED**
Context Sensitive	ContextCraft	**75.50%**	**84.10%**	**3.10**	**74.67%**	**83.90%**	**3.13**
Deep Learning	RNN-att-Copy	49.51%	62.21%	4.58	48.30%	61.75%	4.60
	CodeBERT	59.63%	73.21%	4.01	58.19%	71.75%	4.06
	UniXcoder	60.39%	75.90%	3.92	58.93%	74.35%	3.96
Large Language Models	ChatGPT (few-shot)	66.03%	80.16%	3.42	64.55%	79.88%	3.48
	Gemini (few-shot)	64.43%	76.54%	4.05	63.50%	75.90%	4.12
	Llama (few-shot)	61.66%	75.13%	3.87	61.95%	74.80%	3.95

by systematically extracting method names and descriptions from various popular Python modules, covering domains such as web development, database management, game and mobile app development, hardware control (including Raspberry Pi), and GUI frameworks. Using the "inspect" module [21], we automatically extracted 60k method name and description pairs. We performed baseline preprocessing as Gao et al. [23] did, excluding built-in type methods and keeping only those with descriptions of up to 100 tokens [23]. Both datasets were sourced from publicly available official documentation of Java and Python, as these documents are consistent and follow standardized naming conventions.

Notably, Gao et al. [23] divided the dataset into a training set (accounting for 80%), a testing set (accounting for 10%), and a validation set (accounting for 10%), and we followed this setting. Our approach used the training set as the corpus where the best examples were retrieved. To avoid potential bias in evaluation, we assess the few-shot setting of LLMs using the same examples (excluding contextual details) as obtained with the proposed approach for a given input. All of the approaches were evaluated using the same testing set. We assessed performance using widely used *Exact Match* (EM) [23], *Edit Distance* (ED) [23], and *Semantic Similarity Index* (SSI) [11]. EM measures how often the suggested method names are identical to the ground truth. The ED and SSI measures the lexical and semantic similarity between the suggested method names and the ground truth. Noting that LLMs inherently possess randomness, we trial each prompt five times for both the few-shot setting and *ContextCraft* (and present the average performance) to ensure a robust evaluation.

3.4 RQ1: Improving the State of the Art

To answer RQ1, we independently applied each evaluated approach to the same dataset. The evaluation results are presented in Table 2. The first column presents the evaluated approaches, whereas columns 2-4 and 5-7 present the performance metrics for *JavaData* and *PythonData*, respectively.

Table 2 shows that our approach outperforms *RNN-att-Copy* in EM, SSI, and ED, it substantially improved the EM from 49.51% to 75.50% for *JavaData* and from 48.30% to 74.67% for *PythonData*, this led to a significant improvement of 52.49% = (75.50% - 49.51%)/49.51% and 54.60=(74.67%-48.30%)/48.30%. For SSI, the improvement is 35.19%=(84.10%-62.21%)/62.21% and 35.87%=(83.90%-61.75%)/61.75%. Additionally, it reduced the ED from 4.58 to 3.10 and 4.60 to 3.13 with a relative reduction of 32.31% = (4.58 - 3.10)/4.58 and 31.95%=(4.60-3.13)/4.60, respectively. *ContextCraft* and the SOTA baseline (*RNN-att-Copy*) generated 1781 and 1175 expected Java method names, respectively, with 907 method names overlapping where both succeeded. Combining both approaches

Table 3. ContextCraft Performance with different Embedding Models

Embedding Models	*JavaData*			*PythonData*		
	EM	**SSI**	**ED**	**EM**	**SSI**	**ED**
text-embedding-ada-003	**75.50%**	**84.10%**	**3.10**	**74.67%**	**83.90%**	**3.13**
All-mpnet-base-v2+	73.91%	82.89%	3.15	73.18%	83.20%	3.18
GraphCodeBERT	73.35%	82.23%	3.15	73.10%	83.00%	3.20
CodeBERT	71.75%	81.00%	3.17	72.08%	81.50%	3.22
RoBERTa	70.50%	78.50%	3.21	70.10%	78.10%	3.24

would increase the number of exact matches from 1175 (*RNN-att-Copy*) and 1781 (*ContextCraft*) to 2049, demonstrating that the two approaches are complementary. The same is true for the Python dataset, where combining the two approaches would increase the number of exact matches by 12% compared to *ContextCraft* alone.

Our approach also outperformed other deep learning and LLM-based baselines, with a minimum EM improvement of 9.47 percentage points (pp) and 15.11 pp, respectively. ContextCraft achieved an EM score that is relatively 14%=(75.50%-66.03%)/66.03% and 25%=(75.50%-60.39%)/60.39% higher than the LLM-based and deep learning baselines, respectively.

The off-the-shelf LLMs (i.e., ChatGPT, Llama and Gemini) outperformed the state-of-the-art approach specially designed for this task, i.e., *RNN-att-Copy*, although such models were less accurate than the proposed approach. For example, ChatGPT improved the chance of EM from 49.51% (*RNN-att-Copy*) to 66.03%, with a relative improvement of 33.37%=(66.03%-49.51%)/49.51%. It may suggest that LLMs have great potential in functional description-based method name generation, even without sophisticated customization.

To reveal why the proposed approach can outperform the baselines, we manually analyzed some examples where it worked better than baselines. A typical example is the method "*getItemListeners*" whose functional description is *"returns an array of all the item listeners registered on this choice"*. The state-of-the-art approach *RNN-att-Copy* suggested method name "*changeFont*" that is substantially different from the ground truth. ChatGPT with few-shot setting generated method name "*getListener*" that is close to the ground truth, but it still misses one important word "*item*" that modifies the keyword "*Listener*". Gemini suggested "*getListeners*" that is even closer to the ground truth by turning the singular form of "*Listener*" into its plural form "*Listeners*" although it still misses the word "*item*". Llama suggested "*getItems*": It retrieved the word "*item*" but missed "*Listeners*". In contrast, the proposed approach suggested "*getItemListeners*" that is identical to the ground truth. It retrieved "*get*" and "*Listeners*" because, from contextual details of retrieved examples, the PTP introduced in Section 2.3 identified "*get*" as a possible prefix in the method name (with a probability of 61%) and "*Listeners*" as a possible suffix (with a probability of 56%). It did not suggested "*getListeners*" as Gemini did because the LFM introduced in Section 2.5 told the LLM that "*getListeners*" generated by it on a retrieved example was incorrect and had an ED of 5 to the expected name ("*getFocusListeners*").

3.5 RQ2: Impact of Embedding Models on Performance

To address RQ2, we conducted the Example Retrieval process (detailed in Section 2.1) using various text embedding models. Distinct examples were retrieved from different embedding models for each input functional description. Using these examples, *ContextCraft* generates context-rich prompts that are fed into the LLM, as outlined in Section 2.7, and evaluates the overall performance.

Table 4. Effect of Probabilistic Token Positioning (PTP)

Setting	*JavaData*			*PythonData*		
	EM	**SSI**	**ED**	**EM**	**SSI**	**ED**
Enabling PTP	75.50%	84.10%	3.10	74.67%	83.90%	3.13
Disabling PTP	69.67%	79.69%	3.27	68.10%	78.55%	3.35
Relative Improvement	**8.37%**	**5.53%**	**-5.20%**	**9.65%**	**6.81%**	**-6.20%**

From Table 3, we observe that the *ContextCraft*, when using the text-embedding-ada-003 model (default sitting), outperformed the other models, achieving EM, SSI, and ED scores of 75.50%, 84.10%, and 3.10, respectively. This impressive performance is mainly due to OpenAI's next-generation embedding model, which generates embeddings with up to 3072 dimensions, enhancing the semantic richness and precision [39]. Additionally, we observe that other embedding models also contribute to improving the overall performance of ContextCraft, offering viable alternatives based on specific needs or constraints.

3.6 RQ3: Effect of Individual Components

To answer RQ3, we disabled each component individually and conducted the evaluation again. By comparing the results against those of the default setting (where all components were enabled), we may quantitatively reveal the effect of the given component. The proposed approach comprises three key components: PTP, PWI, and LFM. Consequently, we answer RQ3 with the following three independent experiments by validating the effect of the three key components, respectively.

3.6.1 Effect of Probabilistic Token Positioning. To investigate the effect of PTP proposed in Section 2.3, we disabled it and repeated the evaluation. The evaluation results are presented in Table 4, where the last row presents the relative improvement in EM and SSI and a reduction in ED (where a more significant ED indicates lower performance). PTP substantially improved the performance of the proposed approach. For *JavaData*, it increased the chance of EM and SSI by 8.37%=(75.50%-69.67%)/69.67%) and 5.53%=(84.10%-79.69%)/79.69%, respectively. It also reduced the ED by 5.20%=(3.27-3.10)/3.27. For *PythonData*, it increased the chance of EM and SSI by 9.65%=(74.67%-68.10%)/68.10%) and 6.81%=(83.90%-78.55%)/78.55%, respectively. It also reduced the ED by 6.20%=(3.35-3.13)/3.35.

The following example explains how PTP helps the proposed approach. The example comes from class *oracle.sql.REF* where the functional description of method "*setObject*" is as follows: "*Sets the value of the structured type that this object references to the given instance of an object.*" For the given functional description of the method, the default setting of the approach where PTP was enabled by the suggested method name "*setObject*" which is identical to the ground truth. However, disabling PTP made the approach suggest the name "*setStructuredType*" different from the ground truth. The significant difference is that the two suggested names employed different suffixes ("*StructureType*" vs "*Object*"). The default setting with PTP succeeded in generating the correct suffix ("*Object*") because from a retrieved example, PTP identified that "*Object*" had a high probability (0.64) of appearing in method names as a suffix. As a result, the proposed approach suggested copying "*Object*" when it recognized the word "*Object*" in the query (i.e., the given functional description). In contrast, without PTP, the approach failed to identify it and generated "*structuredType*" by copying it from the input.

Table 5. Effect of Pivot Word Identification (PWI)

Setting	*JavaData*			*PythonData*		
	EM	**SSI**	**ED**	**EM**	**SSI**	**ED**
Enabling PWI	**75.50%**	**84.10%**	**3.10**	**74.67%**	**83.90%**	**3.13**
Disabling PWI	72.53%	81.76%	3.20	71.15%	80.55%	3.28
Relative Improvement	**4.09%**	**2.86%**	**-3.13%**	**4.71%**	**4.16%**	**-4.57%**

Table 6. Effect of LLM-Based Feedback Mechanism (LFM)

Setting	*JavaData*			*PythonData*		
	EM	**SSI**	**ED**	**EM**	**SSI**	**ED**
Enabling LFM	75.50%	84.10%	3.10	74.67%	83.90%	3.13
Disabling LFM	73.94%	82.79%	3.15	73.18%	82.45%	3.17
Relative Improvement	**2.11%**	**1.58%**	**-1.59%**	**2.04%**	**1.67%**	**-1.26%**

3.6.2 Effect of Pivot Word Identification. We evaluated the effect of PWI by disabling it and repeating the evaluation. From Table 5, we observe that enabling PWI improved the performance of the proposed approach by increasing the chance of EM by 4.09% for *JavaData* and 4.71% for *PythonData*. It also enhanced SSI by 2.86% and 4.71%, respectively. It reduced the average ED between the generated names and the ground truth by 3.13% (*JavaData*) and 4.57%(*PythonData*).

An illustrating example comes from *javax.swing.GroupLayout* where the method "*findWidgetGap*" is described as: "*Determines the gap needed between two widgets for alignment*". Given this functional description, the default setting of the approach where PWI was enabled suggested method name "*findWidgetGap*" which is exactly the same as what the original developers coined. The alternative version of the approach where PWI was disabled suggested method name "*determineWidgetGap*" that is slightly different from the ground truth in that they employed different verbs ("*determine*" vs "*find*"). The default setting with PWI succeeded in generating the correct verb ("*find*") because, from a retrieved example, PWI identified a strong correlation between the pivot word "determine" in the functional description and the token "*find*" in method names, with a high similarity score of 0.63. As a result, the proposed approach suggested to generate "*find*" when it recognized the pivot word "*determines*" in the given functional description. In contrast, without PWI, the approach failed to map "*determines*" into "*find*", but generated "*determine*" by copying it from the input because it realized that this word was important.

3.6.3 Effect of LLM-Based Feedback Mechanism. To evaluate the effect of the LFM proposed in Section 2.5, we disabled it and repeated the evaluation. Table 6 presents the evaluation results. From this table, we observe that LFM could slightly improve the performance of the proposed approach, increasing the chance of EM by 2.11% and 2.04% for *JavaData* and *PythonData*, respectively. It also improved SSI by 1.58% and 1.67%. It also reduced the ED between suggested method names and the corresponding ground truth by 1.59% and 1.26%. Although the performance improvement caused by LFM is smaller than that caused by other components (e.g., PTP and PWI), LFM has a positive effect concerning all employed performance metrics, suggesting that it is beneficial and should not be disabled. We illustrate the effect of the LFM with a real-world example from *javax.swing.plaf.ProgressBarUI*. In this project, there is a method "*startAnimationThread*" whose function is descripted by the original authors as follows: "*starts the animation thread, creating and*

Table 7. Improving Various LLMs

LLMs	Setting	*JavaData*			*PythonData*		
		EM	**SSI**	**ED**	**EM**	**SSI**	**ED**
ChatGPT-4o	Few-Shot	66.03%	80.16%	3.42	64.55%	79.88%	3.48
	ContextCraft	75.50%	84.10%	3.10	74.67%	83.90%	3.13
	Improvement	**14.34%**	**4.92%**	**-9.36%**	**15.68%**	**5.03%**	**-10.06%**
ChatGPT-4	Few-Shot	65.50%	79.70%	3.40	63.90%	79.30%	3.45
	ContextCraft	74.90%	83.70%	3.18	73.80%	83.30%	3.35
	Improvement	**14.35%**	**5.02%**	**-6.47%**	**15.56%**	**5.04%**	-9.55%
ChatGPT-3.5	Few-Shot	55.11%	74.33%	3.85	54.50%	73.80%	3.90
	ContextCraft	64.34%	78.09%	3.51	63.55%	77.65%	3.56
	Improvement	**16.75%**	**5.52%**	**-8.83%**	**16.63%**	**5.22%**	**-8.72%**
Gemini-1.5	Few-Shot	64.43%	76.54%	4.05	63.50%	75.90%	4.12
	ContextCraft	73.37%	79.67%	3.67	72.75%	79.20%	3.70
	Improvement	**13.88%**	**4.08%**	**-9.38%**	**14.56%**	**4.35%**	**-10.19%**
Llama-3	Few-Shot	61.66%	75.13%	3.87	61.95%	74.80%	3.95
	ContextCraft	70.03%	80.65%	3.45	69.55%	80.20%	3.51
	Improvement	**13.57%**	**7.35%**	**-10.85%**	**12.27%**	**7.22%**	**-11.13%**

initializing it if necessary". The default setting of the proposed approach, where the LFM was enabled, suggested the expected method name correctly. In contrast, when LFM was disabled, the proposed approach suggested method name "*initiateAnimationThread*" that is slightly different from the ground truth in that "*start*" is replaced by "*initiate*". The default setting with the LFM succeeded in generating the correct prefix ("*start*") because from a retrieved example (as shown below), the LFM gave a suggestion to choose "*start*" over "*initiate*" in the predicted method names. The retrieved example is as follows:

> Function description: "*Initiates the download of a file by creating and managing a dedicated download thread*"; Method name: '*startDownloadThread*'; Probabilistic token positioning: *Suffix ('thread', Probability:57%)*; Pivot word identification: 'initiate': ('start', 57%), 'download': ('download', 65%), 'thread': ('thread', 61%); LLM-based feedback mechanism: LLM predicted method name 'initiateDownloadThread', which has an ED score of 6 compared to the actual '**start**DownloadThread'.

When the LFM was disabled, the feedback message was missing, and thus, the proposed approach simply copied the word "*initiate*" from the functional description.

3.7 RQ4: Working with Various LLMs

To answer RQ4, we replaced the underlying LLM (i.e., ChatGPT-4o) with other LLMs and re-evaluated the approach. The purpose of the evaluation is twofold. First, it reveals whether the proposed approach remains effective when applied to other LLMs. Second, it determines whether the improvements observed in other LLMs are comparable to those achieved with ChatGPT-4o.

We evaluated various LLMs using few-shot settings and then re-tested with ContextCraft-generated prompts. Table 7 shows results comparing traditional few-shot settings ('without') and

Table 8. Performance on Private Data

Evaluated Approaches	EM	SSI	ED
ContextCraft	**74.91%**	**83.85%**	**3.15**
RNN-att-Copy	44.51%	57.21%	4.58
CodeBERT	54.63%	68.21%	4.01
UniXcoder	55.39%	70.90%	3.92
ChatGPT (few-shot)	65.93%	79.46%	3.47
Gemini (few-shot)	63.88%	75.75%	4.08
Llama (few-shot)	61.45%	74.89%	3.91

ContextCraft-generated prompts ('with'). The"*Improvement*" measures to what extent the proposed approach improved (for EM and SSI) or reduced (for ED) the performance of the evaluated LLM. Notably, it presents the relative improvement (percentages) instead of absolute distance.

From Table 7, we make the following observations:

- The proposed approach remained accurate when the underlying LLM was replaced with other mainstream LLMs. The chance of EM varied from 64.34% (ChatGPT-3.5) to 73.37% (Gemini-1.5), substantially greater than that (49.51%) of the state-of-the-art approach *RNN-att-Copy*.
- For *JavaData*, the proposed approach substantially improved all of the evaluated LLMs. The relative improvement concerning the chance of EM on ChatGPT-4o, ChatGPT-4, ChatGPT-3.5, Gemini-1.5 and Llama-3 are 14.34%, 14.35%, 16.75%, 13.88% and 13.57%, respectively. The relative improvement in SSI is 4.92%, 5.02%, 5.52%, 4.08% and 7.35%, respectively. It also successfully reduced the ED by 9.36%, 6.47%, 8.83%, 9.38%, and 10.85%, respectively.
- For *PythonData*, the proposed approach also substantially improved all of the evaluated LLMs. The relative improvement concerning the chance of EM on ChatGPT-4o, ChatGPT-4, ChatGPT-3.5, Gemini-1.5 and Llama-3 is 15.68%, 15.56%, 16.63%, 14.56% and 12.27%, respectively. The relative improvement in SSI is 5.03%, 5.04%, 5.22%, 4.35% and 7.22%, respectively. It also successfully reduced the ED by 10.06%, 9.55%, 8.72%, 10.19%, and 11.13%, respectively.
- ChatGPT-4o is the best underlying LLM for the proposed approach. ChatGPT-4o demonstrated superior performance in the EM when compared to its counterparts. Specifically, ChatGPT-4o surpassed ChatGPT-4 and ChatGPT-3.5 by 0.80%= (75.50%-74.90%)/74.90% and 17.35%=(75.50%-64.34%)/64.34%, respectively. Compared against non-ChatGPT models (e.g., Gemini-1.5 and Llama-3), ChatGPT-4o equipped with the proposed approach also performed the best in any of the employed performance metrics.
- The proposed approach resulted in the greatest improvement on ChatGPT-3.5. Its improvement on ChatGPT-3.5, i.e., 16.75% (EM), is substantially greater than those on other models. One possible reason is that ChatGPT-3.5 alone (i.e., without the proposed approach) had low performance, e.g., the lowest EM, which left great space for improvement.

Besides the underlying LLMs, we also evaluated how the length of functional descriptions and method names would influence the proposed approach's performance. Our evaluation results suggest that the proposed approach worked well on various methods, regardless of the length of method names or functional descriptions. Due to space limitations, results are provided online [13]. Based on the preceding discussion, we conclude that the proposed approach has good generalization ability and works well with various LLMs on various method names' lengths.

3.8 RQ5: Performance on Private Project

It is likely that *JavaDataset* and *PythonData*, automatically collected from open-source applications, may have some overlap with the training data of the involved LLMs like ChatGPT. This is a well-known data contamination problem, which may result in overestimated performance metrics[40]. To validate the accuracy of the proposed approach on data that is guaranteed to be unknown to the underlying LLMs, we performed an evaluation using a private project dataset. We created the *PrivateData* (available at [12]) dataset by extracting 50 Java method names and their corresponding functional descriptions from the source code of a private project that is not hosted on any online repository. The functional descriptions were manually written in English by a software developer working on the project, with an average length of 10.79 words. The shortest description has 6 words, and the longest has 19 words. All method names were written in camelCase, and we tokenized them into their components for analysis. The tokenized method names had an average of 3.32 tokens, with a minimum of 2 and a maximum of 5. We applied each of the evaluated approaches to the resulting dataset independently, and evaluation results are presented in Table 8: (a table similar to Table 2). From table 8, we observe that *ContextCraft* outperformed the state-of-the-art baselines on *PrivateData* as well, achieving EM and SSI scores of 74.91% and 83.85%, respectively. It also reduced the ED score to 3.15.

3.9 Threats to Validity

The first threat to external validity is that only five LLMs were employed for evaluation because of our limited budget. Thus, conclusions drawn from them may not hold for other LLMs. To reduce the threat, we employed the most popular and widely recognized LLMs. The second threat to external validity is the limited size of the testing data. It is important to note that gathering such a dataset is challenging and time-intensive. To reduce the cost and potential bias, we reused one public dataset that other researchers used for the same task. We also published the replication package and datasets so other researchers could replicate them (even with additional datasets).

A threat to internal validity is that some test data may have been included in the LLMs' training data. Notably, these models do not explicitly disclose their training data sources, and certain models (such as Genimi) can even access online data during runtime, making it challenging to ensure that all testing data remain unfamiliar to the LLMs. To reduce the threat, we constructed a private dataset in Section 3.8, and the proposed approach remained accurate on this private dataset.

A threat to construct validity is that the method names in the dataset are not necessarily the best. All such names were coined by the original developers, and thus it is likely that most of the names are of high quality. However, there is no guarantee that they will always be the best since developers may have made mistakes and failed to figure out the best names. That is the reason why *renaming methods* are popular: They are often used to correct imperfect (even incorrect) method names. To reduce the threat, we only used methods from official Java and Python APIs, where all methods were extensively discussed, well-documented, and widely used.

4 Related Work

The most closely related work is the approach developed by Gao et al. [23]. They employed an attention-based encoder-decoder neural network with explicit copying and coverage mechanisms to translate functional descriptions into method names. They successfully addressed the vocabulary explosion challenge and mitigated the cold-start problem with a novel training mechanism. Wang et al. [42] observed that method names are often assigned before their implementations are available. They proposed a novel approach to suggest all potential methods (providing method names only) for a given class by using the class name as input instead of relying on method bodies. Their method combines deep learning models, i.e., recurrent neural networks with attention mechanisms and heuristic techniques, to generate meaningful method names during pre-implementation. Our approach proposed in this paper differs from these approaches [23, 42] in that ours is the first LLM-based approach to functional description-based method name generation. In contrast, their approach employs customized neural networks specially designed for this task.

Several approaches have been proposed to generate method names according to source code, especially method bodies. For example, MNire [37] treats method name suggestion as a text summarization problem by taking source code as texts. Notably, *MNire* generates candidate method names according to not only method bodies but also interfaces and enclosing class names. MNire was not designed to generate method names but to identify inconsistent method names. Consequently, it compared the generated method name against the existing one, and the existing name was inconsistent with its body if it was significantly different from the generated name. The novel approach proposed by Allamanis et al. [2] leverages a log-bilinear neural language model to suggest method names according to method bodies. A significant advantage of this model is that it captures long-range dependencies within the source code. Later, Allamanis et al. [3] cast method naming as a challenge in extreme source code summarization by taking method name as a concise representation of the entire method body. To this end, they proposed a novel attention-based neural network, utilizing convolution on the individual tokens within the method body. This approach allows the network to capture high-level patterns in the code. Alon et al. [4] developed innovative methods for method name suggestions by exploring the structural properties of code. They introduced a novel approach that utilizes paths within the Abstract Syntax Tree (AST) of the method, capturing the inherent structure by considering all paths between leaves. Building on this foundational work, they further refined their model into what is known as code2vec [6]. This model aggregates these paths into a single vector using an attention mechanism, which assigns weights to each path based on its importance to the method's functionality. Xu et al. [49] employed Hierarchical Attention Networks (HANs) to analyze the hierarchical structure of code snippets for method naming. Ge and Kuang [24] proposed a two-stage method name generation approach that utilizes both graph neural networks and dual attention mechanisms. The approach proposed by Wang et al. [48] generates method names by leveraging both local and global information to enhance consistency checking. Local context is derived from the method itself, while global context comes from related methods within the same project. Li et al. [31] introduced *DEEPNAME*, a context-aware deep learning-based approach that considers the surrounding code structures, like interactions and relationships between methods, to suggest names. Yang et al. [51] proposed a pattern-based approach that leverages recurring patterns in method names to recommend new names. Their approach identifies common naming patterns and uses them to guide the generation of new names. The approach proposed by Høst and Østvold [28] is not to suggest method names from scratch but to identify naming bugs and to fix the bugs by renaming. To fix buggy method names, their approach replaced inconsistent phrases with those associated with a profile of the given method. The association is constructed by static analysis of the semantics of methods and phrases. Our approach differs from these approaches in that our approach generates names according to functional descriptions in natural languages. In contrast, the existing approaches in this paragraph suggest method names according to source code, especially implementing the to-be-named methods. Another difference is that our approach leverages LLMs for the first time to generate method names, whereas none of the introduced related works have exploited such LLMs.

5 Conclusion and Future Work

It is highly desirable to automatically suggest high-quality method names according to functional descriptions in natural languages. However, existing approaches often fail to suggest the expected names, which may result in refusal or substantial manual modification. This paper proposed the first LLM-based approach to suggesting method names according to functional descriptions. The key technique contribution is an algorithm to generate context-rich prompts for LLMs. The prompt explicitly marks the tokens that are likely to appear in method names as well as their potential positions in the names, explicitly presents pivot words and their associated method name tokens, and explicitly specifies the evaluation result of the underlying LLM on selected examples. All such context-sensitive information helps the underlying LLM better understand the given task. We evaluated the proposed approach on a public dataset, and the results show it significantly improves the exact match rate from 49.51% to 75.50%. The results also indicate that the approach works well with various LLMs, including ChatGPT, Gemini, and Llama.

In future, we would like to extend the proposed approach to suggest other identifiers, e.g., class names. It is also interesting to investigate how the proposed approach could be adapted for the reversed task, i.e., generating functional descriptions for a given method. Although it is challenging and time-consuming to build a more comprehensive dataset for the task in the future, such a dataset is highly desirable and could be exploited for both model training and testing.

6 Data Availability

The datasets, implementation code, and replication package are publicly available [12].

Acknowledgments

This work was partially supported by the National Natural Science Foundation of China (62232003, 62172037), China Postdoctoral Science Foundation (No. 2023M740078), and China National Postdoctoral Program for Innovative Talents (BX20240008).

References

[1] Gilson Aidan, C. Safranek, Huang Thomas, V. Socrates, Chi Ling, R. Taylor, and Chartash David. 2023. How Does ChatGPT Perform on the United States Medical Licensing Examination? The Implications of Large Language Models for Medical Education and Knowledge Assessment. *JMIR Medical Education* 9 (2023), e45312. doi:10.2196/45312

[2] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting Accurate Method and Class Names. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)* (Bergamo, Italy). ACM, 38–49. doi:10.1145/2786805.2786844

[3] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *Proceedings of the 33rd International Conference on Machine Learning (ICML)*. PMLR, New York, NY, USA, 2091–2100. http://proceedings.mlr.press/v48/allamanis16.html

[4] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A General Path-Based Representation for Predicting Program Properties. *ACM SIGPLAN Notices* 53, 4 (2018), 404–419. doi:10.1145/3192366.3192412

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *International Conference on Learning Representations (ICLR)* (New Orleans, LA). OpenReview, 1701–1711.

[6] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning Distributed Representations of Code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[7] Madaan Aman, Tandon Niket, Gupta Prakhar, Hallinan Skyler, Gao Luyu, Wiegreffe Sarah, Alon Uri, Dziri Nouha, Prabhumoye Shrimai, Yang Yiming, et al. 2023. SELF-REFINE: Iterative Refinement with Self-Feedback. In *Proceedings of the 37th International Conference on Neural Information Processing Systems (NeurIPS)*. Neural Information Processing Systems Foundation, New Orleans, Louisiana, USA, 46534–46594.

[8] Fares Antaki, Samir Touma, Daniel Milad, Jonathan El-Khoury, and Renaud Duval. 2023. Evaluating the Performance of ChatGPT in Ophthalmology: An Analysis of its Successes and Shortcomings. *Ophthalmology Science* 2 (2023). Online first.

[9] Caprile and Tonella. 2000. Restructuring Program Identifier Names. In *Proceedings of the 2000 International Conference on Software Maintenance* (Vancouver, Canada). IEEE, Piscataway, NJ, USA, 97–107.

[10] Xiangping Chen, Xing Hu, Yuan Huang, He Jiang, Weixing Ji, Yanjie Jiang, Yanyan Jiang, Bo Liu, Hui Liu, Xiaochen Li, et al. 2025. Deep Learning-Based Software Engineering: Progress, Challenges, and Opportunities. *Science China Information Sciences* 68, 1 (2025), 1–88. doi:10.1007/s11432-023-4127-5

[11] Moreno Colombo. 2024. *Semantic Similarity Measures.* Springer Nature Switzerland, Cham, 49–69. doi:10.1007/978-3-031-42819-7_4

[12] ContextCraft. 2024. ContextCraft. https://github.com/contextcraft/contextcraft.

[13] ContextCraft. 2024. Impact of Length. https://github.com/contextcraft/contextcraft/blob/main/Extra_Material/RQ4.pdf.

[14] ContextCraft. 2024. RNN-Copy-Attention Source Code. https://github.com/contextcraft/contextcraft/tree/main/Source_Code/RNN_Copy_Attn. Accessed: 2024-09-12.

[15] Vilar David, Freitag Markus, Colin Cherry, Luo Jiaming, Ratnakar Viresh, and Foster George. 2023. Prompting PaLM for Translation: Assessing Strategies and Performance. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers).* Association for Computational Linguistics, Toronto, Canada, 15406–15427. doi:10.18653/v1/2023.acl-long.859

[16] Florian Deissenboeck and Markus Pizka. 2006. Concise and Consistent Naming. *Software Quality Journal* 14 (2006), 261–282.

[17] Jesse Dodge, Gabriel Ilharco, Roy Schwartz, Ali Farhadi, Hannaneh Hajishirzi, and Noah A. Smith. 2020. Fine-Tuning Pretrained Language Models: Weight Initializations, Data Orders, and Early Stopping. *arXiv preprint arXiv:2002.06305* (2020), 1536–1547.

[18] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The Llama 3 Herd of Models. *arXiv preprint arXiv:2407.21783* 33, 3 (2024), 1–35.

[19] Dror G. Feitelson, Ayelet Mizrahi, Nofar Noy, Aviad Ben Shabat, Or Eliyahu, and Roy Sheffer. 2020. How Developers Choose Names. *IEEE Transactions on Software Engineering* 48, 1 (2020), 37–52.

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

[21] Python Software Foundation. 2023. *inspect: Get Useful Information from Live Objects.* Python. Version 3.10, Available at: https://docs.python.org/3/library/inspect.html.

[22] Haifeng Gao, Bin Dong, Yongwei Zhang, Tianxiong Xiao, Shanshan Jiang, and Yuan Dong. 2021. An Efficient Method of Supervised Contrastive Learning for Natural Language Understanding. In *Proceedings of the 2021 7th International Conference on Computer and Communications (ICCC).* IEEE, Online, 1698–1704. doi:10.1109/ICCC54389.2021.9674736

[23] Sa Gao, Chunyang Chen, Zhenchang Xing, Yukun Ma, Wen Song, and Shang-Wei Lin. 2019. A Neural Model for Method Name Generation from Functional Description. In *Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (Vancouver, Canada). IEEE, Piscataway, NJ, USA, 414–421.

[24] Fan Ge and Li Kuang. 2021. Keywords Guided Method Name Generation. In *Proceedings of the 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC).* IEEE, Online, 196–206. doi:10.1109/ICPC52881.2021.00027

[25] Remo Gresta, Vinicius Durelli, and Elder Cirilo. 2023. Naming Practices in Object-oriented Programming: An Empirical Study. *Journal of Software Engineering Research and Development* 11, 1 (2023), 5:1–5:27. doi:10.5753/jserd.2023.2582

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

[27] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *9th International Conference on Learning Representations (ICLR 2021).* IEEE, Online, 48–65. https://openreview.net/forum?id=jLoC4ez43PZ

[28] Einar W. Høst and Bjarte M. Østvold. 2009. Debugging Method Names. In *European Conference on Object-Oriented Programming* (Vancouver, Canada). Springer, Berlin, Heidelberg, Germany, 294–317.

[29] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *Proceedings of the 10th International Conference on Learning Representations.* Association for Computational Learning, Virtual Event, 59–70. https://openreview.net/forum?id=nZeVKeeFYf9

[30] Katharina Jeblick, Balthasar Schachtner, Jakob Dexl, Andreas Mittermeier, Anna Theresa Stüber, Johanna Topalis, Tobias Weber, Philipp Wesp, Bastian Oliver Sabel, Jens Ricke, and Michael Ingrisch. 2024. ChatGPT makes medicine

easy to swallow: An exploratory case study on simplified radiology reports. *European Radiology* 34, 5 (2024), 2817–2825. doi:10.1007/s00330-023-10213-1

[31] Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. 2022. Symlm: Predicting Function Names in Stripped Binaries via Context-Sensitive Execution-Aware Code Embeddings. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, Los Angeles, CA, USA, 1631–1645. doi:10.1145/3548606.3560616

[32] Myeongsoo Kim, Saurabh Sinha, and Alessandro Orso. 2023. Adaptive REST API Testing with Reinforcement Learning. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Luxembourg, 446–458. doi:10.1109/ASE56229.2023.00218

[33] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2006. What's in a Name? A Study of Identifiers. In *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC)* (Athens, Greece). IEEE Computer Society, Los Alamitos, CA, USA, 3–12. doi:10.1109/ICPC.2006.51

[34] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMOSA: Escaping Coverage Plateaus in Test Generation With Pre-trained Large Language Models. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Melbourne, Australia, 919–931. doi:10.1109/ICSE48619.2023.00085

[35] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *arXiv preprint arXiv:1907.11692* 30 (2019), arXiv:1907.11692. https://arxiv.org/abs/1907.11692

[36] Ehsan Mashhadi and Hadi Hemmati. 2021. Applying CodeBERT for Automated Program Repair of Java Simple Bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, Madrid, Spain, 505–509. doi:10.1109/MSR52588.2021.00072

[37] Son Nguyen, Hung Phan, Trinh Le, and Tien N. Nguyen. 2020. Suggesting Natural Method Names to Check Name Consistencies. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. Association for Computing Machinery, New York, NY, USA, 1372–1384. doi:10.1145/3377811.3380926

[38] OpenAI. 2023. Generative Pre-trained Transformer 4 (GPT-4). *OpenAI* 4, 1 (2023), 1–2. https://cdn.openai.com/papers/gpt-4.pdf

[39] OpenAI. 2024. New Embedding Models and API Updates. https://openai.com/index/new-embedding-models-and-api-updates/.

[40] Oscar Sainz, Jon Ander Campos, Iker García-Ferrero, Julen Etxaniz, Oier Lopez de Lacalle, and Eneko Agirre. 2023. NLP Evaluation in Trouble: On the Need to Measure LLM Data Contamination for Each Benchmark. *arXiv preprint* 30, 5 (2023), arXiv:2310.18018.

[41] S. Samsi, Dan Zhao, Joseph McDonald, Baolin Li, Adam Michaleas, Michael Jones, William Bergeron, J. Kepner, Devesh Tiwari, and V. Gadepally. 2023. From Words to Watts: Benchmarking the Energy Costs of Large Language Model Inference. In *Proceedings of the 2023 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–9. doi:10.1109/HPEC58863.2023.10363447

[42] Wang Shangwen, Wen Ming, Lin Bo, Liu Yepang, Tegawendé, F. Bissyandé, and Mao Xiaoguang. 2023. Pre-implementation Method Name Prediction for Object-oriented Programming. *ACM Transactions on Software Engineering and Methodology* 32 (2023), 1–35. doi:10.1145/3597203

[43] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. 2020. MPNet: Masked and Permuted Pre-training for Language Understanding. *Advances in Neural Information Processing Systems* 33 (2020), 16857–16867.

[44] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, et al. 2023. Gemini: A Family of Highly Capable Multimodal Models. arXiv:2312.

[45] Tim van Dam, Maliheh Izadi, and Arie van Deursen. 2023. Enriching Source Code with Contextual Data for Code Completion Models: An Empirical Study. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, Virtual Conference, 170–182.

[46] Izhar Wallach and Abraham Heifets. 2017. Most Ligand-Based Benchmarks Measure Overfitting Rather Than Accuracy. *Journal of Chemical Information and Modeling* 58, 5 (2017), 916–932. doi:10.1021/acs.jcim.7b00403

[47] Talia Waltzer, Riley L. Cox, and Gail D. Heyman. 2023. Testing the Ability of Teachers and Students to Differentiate between Essays Generated by ChatGPT and High School Students. *Human Behavior and Emerging Technologies* 3, 4 (2023), 21–39. doi:10.1155/2023/1923981

[48] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-Instruct: Aligning Language Models With Self-Generated Instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Toronto, Canada, 13484–13508. doi:10.18653/v1/2023.acl-long.754

[49] Sihan Xu, Sen Zhang, Weijing Wang, Xinya Cao, Chenkai Guo, and Jing Xu. 2019. Method Name Suggestion with Hierarchical Attention Networks. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. ACM, New York, NY, USA, 10–21. doi:10.1145/3294032.3294079

[50] Guang Yang, Yu Zhou, Wenhua Yang, Tao Yue, Xiang Chen, and Taolue Chen. 2024. How Important Are Good Method Names in Neural Code Generation? A Model Robustness Perspective. *ACM Trans. Softw. Eng. Methodol.* 33, 3 (2024), 1–35.

[51] Yanping Yang, Ling Xu, Meng Yan, Zhou Xu, and Zhongyang Deng. 2022. A Naming Pattern Based Approach for Method Name Recommendation. In *Proceedings of the 2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Charlotte, NC, USA, 344–354. doi:10.1109/ISSRE55969.2022.00041

[52] Yue Yu, Simiao Zuo, Haoming Jiang, Wendi Ren, Tuo Zhao, and Chao Zhang. 2021. Fine-Tuning Pre-trained Language Model with Weak Supervision: A Contrastive-Regularized Self-Training Approach. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Online, 1063–1077. doi:10.18653/v1/2021.naacl-main.85

[53] Zichun Yu, Tianyu Gao, Zhengyan Zhang, Yankai Lin, Zhiyuan Liu, Maosong Sun, and Jie Zhou. 2022. Automatic label sequence generation for prompting sequence-to-sequence models. *Proceedings of the 29th International Conference on Computational Linguistics* (2022), 4965–4975. https://aclanthology.org/2022.coling-1.440/

[54] Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2023. Unifying the Perspectives of NLP and Software Engineering: A Survey on Language Models for Code. *arXiv preprint* 23 (2023), arXiv:2311.07989. https://arxiv.org/abs/2311.07989

[55] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2023. Large Language Models are Human-Level Prompt Engineers. In *Proceedings of the Eleventh International Conference on Learning Representations (ICLR)* (Kigali, Rwanda). IEEE, 35–60. https://openreview.net/forum?id=YdqwNaCLCx

Received 2024-09-13; accepted 2025-01-14