# Lightweight Global and Local Contexts Guided Method Name Recommendation with Prior Knowledge

Shangwen Wang
wangshangwen13@nudt.edu.cn
College of Computer Science, National University of Defense Technology
Changsha, China

Ming Wen*
mwenaa@hust.edu.cn
School of Cyber Science and Engineering, Huazhong University of Science and Technology
Wuhan, China

Bo Lin
linbo19@nudt.edu.cn
College of Computer Science, National University of Defense Technology
Changsha, China

Xiaoguang Mao
xgmao@nudt.edu.cn
College of Computer Science, National University of Defense Technology
Changsha, China

## ABSTRACT

The quality of method names is critical for the readability and maintainability of source code. However, it is often challenging to construct concise method names. To alleviate this problem, a number of approaches have been proposed to automatically recommend high-quality names for methods. Despite being effective, existing approaches meet their bottlenecks mainly in two aspects: (1) the leveraged information is restricted to the target method itself; and (2) lack of distinctions towards the contributions of tokens extracted from different program contexts. Through a large-scale empirical analysis on +12M methods from +14K real-world projects, we found that (1) the tokens composing a method's name can be frequently observed in its callers/callees; and (2) tokens extracted from different specific contexts have diverse probabilities to compose the target method's name. Motivated by our findings, we propose, in this paper, a context-guided method name recommender, which mainly embodies two key ideas: (1) apart from the *local context*, which is extracted from the target method itself, we also consider the *global context*, which is extracted from other methods in the project that have call relations with the target method, to include more useful information; and (2) we utilize our empirical results as the *prior knowledge* to guide the generation of method names and also to restrict the number of tokens extracted from the global contexts. We implemented the idea as `Cognac` and performed extensive experiments to assess its effectiveness. Results reveal that `Cognac` can (1) perform better than existing approaches on the *method name recommendation* task (e.g., it achieves an F-score of 63.2%, 60.8%, 66.3%, and 68.5%, respectively, on four widely-used datasets, which all outperform existing techniques); and (2) achieve higher performance than existing techniques on the *method name consistency checking* task (e.g., its overall *accuracy* reaches 76.6%, outperforming the state-of-the-art MNire by 11.2%). Further results reveal that the caller/callee information and the prior knowledge all contribute significantly to the overall performance of `Cognac`.

## CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; **Maintaining software**; *Software evolution*.

## KEYWORDS

Method name recommendation, Deep learning, Code embedding.

**ACM Reference Format:**
Shangwen Wang, Ming Wen, Bo Lin, and Xiaoguang Mao. 2021. Lightweight Global and Local Contexts Guided Method Name Recommendation with Prior Knowledge. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3468264.3468567

## 1 INTRODUCTION

The quality of identifier names plays critical roles in the readability and maintainability of source code [21, 22, 27, 32, 37, 56, 66]. Due to the huge amount of information contained towards the semantic of diverse program elements (e.g., variables and classes), developers often rely heavily on identifiers for program comprehension [23–26, 45, 54, 55, 60]. Method names, as a special type of identifiers, are especially important since they are the smallest named units of aggregated behaviour and also the cornerstone of abstraction in most conventional programming languages [38]. Nevertheless, in practice, developers often find it hard to name identifiers [46], and they often write inconsistent names in programs due to various reasons such as insufficient communication among development teams and lack of understanding of project development histories [16, 36, 41]. Actually, constructing high quality method names is considered as a challenging task, especially for inexperienced developers [38, 40].

It will cause many side effects if a method name does not match its associated method body (i.e., an inconsistent method name). Specifically, it can affect the readability and maintenance of the

*Co-first and corresponding author.


*ESEC/FSE '21, August 23–28, 2021, Athens, Greece*

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468567

code [4, 14, 37], and hence induce potential software defects or API misuses [20, 21, 65, 69]. For instance, Abebe *et al.* [4] found that inconsistent method names can negatively influence software maintenance activities. Besides, Butler *et al.* [20] also observed that inappropriate names can significantly increase the number of code quality issues detected by static checkers such as FindBugs [2]. To alleviate this problem, various approaches have been proposed recently to automatically recommend high-quality names given the implementation of a method [5, 10, 51]. For instance, Code2vec [12] represents source code using the paths that connect two leaf nodes in the Abstract Syntax Tree (AST), and then recommends to reuse the name of those methods who share similar syntax structures with the target one (i.e., the method whose name is going to be inferred). Existing studies [5, 10, 44, 46, 48, 51] deem that method names and identifiers are composed of ***tokens***, which are split from the name based on the camel case and underscore naming conventions. For instance, identifier `"methodName"` is composed of tokens `"method"` and `"name"`. MNire then treats method name recommendation as an abstract summarization task based on the seq2seq paradigm, and generates the tokens to compose the method names using those extracted from the implementation of the methods [51].

Despite their effectiveness, the major limitation concerning the performance of existing techniques is that they only consider the information locally to recommend names. Specifically, they only consider the implementation of a method to infer its method name [10, 12, 40]. However, a recent study shows that a large proportion of the method name tokens cannot be observed from the interfaces and implementations of the methods [51]. In this study, we find that such method name tokens can be often observed from the callees of the target method. Besides, recent studies have also shown that the information of program dependencies such as the caller/callee relations can effectively serve for diverse software engineering tasks [29, 43, 68, 70]. Therefore, it motivates us to investigate whether the context information of method call relations can be utilized to better infer appropriate method names. Incorporating more information, however, will inevitably increase the number of tokens feeding to a recommendation model. Consequently, it will bring new challenges since the long sequence input might induce more potential noises and may also reduce the generality of the learned model as revealed by recent studies [11, 59]. We also observe that those tokens constituting method names tend to occur more frequently in certain contexts (e.g., *parameters*, *return types* and *other types of statements*), which indicates that the contributions of tokens under diverse program contexts to compose an appropriate method name are different. Therefore, we are motivated to take into consideration the context information of different tokens.

In pursuit of designing a more effective approach to recommend appropriate method names, we first performed a large-scale empirical study on +14K top-starred GitHub repositories with +12M methods to validate our observations and motivations. We found that the methods that have call relations with the target one can provide abundant information to help infer method names. In detail, the tokens of a caller's method name can be found in its callee (either the interface or the implementation) for 40.5% of the total cases. We also found that the tokens extracted from different contexts of a method have diverse probabilities to compose the name of a method. For instance, tokens from the *ReturnStatement* generally possess higher probabilities (e.g., nearly 20.0%) to compose the target method name than those from other types of statements. Such empirical results confirmed our observations and intuitions.

Supported by our empirical findings, we propose a Context-guided method name recommender, **Cognac**, which in general follows the *seq2seq* paradigm to infer method names utilizing program entity names. In such a paradigm, the extracted program entity tokens are rephrased into a short sequence of tokens which forms the recommended method name. The reason why Cognac adopts the seq2seq paradigm is that previous studies have shown the superiorities of code tokens on name prediction [39, 51]. In particular, Nguyen *et al.* have revealed that purely relying on the representation of code tokens yields better results than that of using the AST or Program Dependence Graph (PDG) structures for method name recommendation [51]. Although Cognac follows the seq2seq paradigm as adopted by the state-of-the-art [51], it embodies two major novel ideas. First, apart from the ***local context***, which is extracted from the target method itself, including program entity tokens and the associated contexts, it also extracts tokens and their contextual information from other methods that possess call relations with the target method. Such information is denoted as the ***global context***, which can include tokens from a global perspective to help better infer the name of the target method. Second, Cognac utilizes the empirical results as the ***prior knowledge*** to better focus on the critical tokens. Recall that our empirical study has revealed that the probabilities of tokens under diverse *specific contexts* to compose method names are different, and we denote such probabilities as the *prior knowledge* in this study. The *prior knowledge* is utilized to serve for two main purposes: to guide the method name generation as well as to reduce the size of the input sequences. On one hand, different from the state-of-the-art MNire [51], which completely relies on the attention mechanism to decide which tokens to focus on when generating the output token, we integrate the prior knowledge with the learned attention weight (i.e., the probabilities of each token from the attention mechanism) to focus on those tokens with higher probabilities. On the other hand, we leverage the prior knowledge to limit the number of tokens that are extracted from the callers/callees, and thus our utilized global context is ***lightweight***. Specifically, we only accept the top ten tokens (such a number is empirically determined through a pre-study experiment) from the implementation of each callee prioritized by the prior knowledge. We exclude the implementation of the caller methods from the input in Cognac to avoid data leakage since the caller's implementation will definitely contain the target method name.

To evaluate the effectiveness of our approach for recommending high-quality method names, we trained and tested Cognac on totally four different datasets, which are the *Java-small*, *Java-med*, and *Java-large* from Alon *et al.* [10] and the one constructed by Nguyen *et al.* [51], containing 11, 1K, 9.5K, and more than 10K Java projects from GitHub respectively. We then compared it against totally 10 baseline approaches. Results show that Cognac outperforms all the state-of-the-art approaches by at least 5.0%, 9.2%, 8.2%, and 7.7% on the four datasets respectively w.r.t *F-score*. Moreover, we also applied Cognac to detect inconsistent method names via checking the lexical similarity between the original method name and the recommended one by Cognac, following the way as adopted by

Nguyen *et al.* [51]. Specifically, we utilized the dataset collected by Liu *et al.* [46] which includes 2,805 inconsistent method name cases mined from 430 Java projects. Results reveal that Cognac outperforms the state-of-the-art MNire significantly (the overall *accuracy* exceeds that of MNire by 11.2%). Furthermore, an ablation study shows that all the design decisions (i.e., information from the caller/callee methods as well as the guidance from the prior knowledge) contribute to the performance of Cognac on both tasks, among which the information from the callee methods is the most significant one.

In summary, our study makes the following contributions:

- **Empirical results:** Our study deepens the understanding towards the naturalness of method names w.r.t their correlations with the caller/callee methods and their tendencies to be observed among different contexts.
- **Method name recommendation with Cognac:** We implement a method name recommender that explores not only the *local context* but also the *global context* in a lightweight strategy and then generates the method name guided by our *prior knowledge*.
- **Performance assessment:** We perform extensive experiments to assess the performance of Cognac. Results reveal that Cognac achieves overall significantly better performance than the existing approaches on both the *method name recommendation* and *method name consistency checking* tasks.

# 2 BACKGROUND AND RELATED WORK

## 2.1 Definitions

Methods are declared and used under certain contexts. To ease our representation, we define several concepts here which will be used in the following contents of this study.

**Implementation context:** Given a method, all the program entities in the method body are referred to as its *implementation context* [51]. It includes all names and structures that are used to implement the method.

**Interface context:** Given a method, the types of the input parameters and the return type of this method are referred to as its *interface context* [51]. Technically, it describes the method's input and output.

**Enclosing context:** Given a method, the name of the class in which the method is defined is referred to as the *enclosing context* [51]. Such context provides the information of the general task/purpose of the class where the method is implemented.

**Call relation:** Given two methods $a$ and $b$, if $b$ is triggered in the *implementation context* of $a$, then the call relation $a \rightarrow b$ is established where $a$ is the caller while $b$ is the callee [68].

## 2.2 Method Name Recommendation

Given the critical role of method names in the readability of source code [17, 31], various techniques have been proposed to address the method name recommendation (MNR) task, that is to automatically generate high-quality method names. Existing techniques can be broadly categorized into program structure dependent and independent. We next introduce each of the state-of-the-art in detail.

### 2.2.1 *Program Structure Dependent.*

Parsing programs from the AST aspect can obtain the syntax structure information, and hence is leveraged by various approaches in program analysis [28, 62, 64]. Mou *et al.* [50] proposed a tree-based convolutional neural network (TBCNN) for programming language processing, in which a convolution kernel is designed over programs' ASTs to capture the structure information. Recently, Bui *et al.* [19] fused capsule networks with TBCNN to achieve higher learning accuracies based on tree structure. Utilizing AST paths that link any two leaf nodes in ASTs is an advanced program representation technique [11]. Code2vec [12] and Code2seq [10] represent a method body into a distributed vector by aggregating the bag of AST paths with the attention mechanism. They then recommend to reuse names of the methods who share similar AST structures with the target method.

Besides utilizing the structure information from the AST, researchers also propose to capture the *data-flow* and *control-flow* information and represent programs as PDG (i.e., *Program Dependency Graph*) to jointly model syntactic and semantic information [7], which is named as Gated Graph Neural Network (GGNN). To mitigate the long-distance relationship problem within the sequence encoder, Fernandes *et al.* [30] developed a framework to extend existing sequence encoders with a graph neural network (sequence GNN). Wang *et al.* [63] developed a novel graph neural architecture (GINN), which, unlike the standard GNN, focuses exclusively on intervals for mining the feature representation of a program and operates on a hierarchy of intervals for scaling the learning to large graphs. GREAT [35] is another model that combines long-distance information with the structure information.

### 2.2.2 *Program Structure Independent.*

Without the guidance from program structures, researchers can also rely on the sequence of method tokens to finish the MNR task. Allamanis *et al.* [5] introduced a log-bilinear neural probabilistic language model for source code which can embed each token into a high dimensional continuous space and select the name that is most similar in this embedding space to those of the function body. They later considered MNR as an extreme summarization task where the method name is regarded as the summary of the method body, and then introduced an attentional neural network that employs convolution on the input code tokens [8]. MNire [51] follows the seq2seq paradigm to generate the tokens of method names using the sequence composed by tokens from the *implementation context*, *interface context*, and *enclosing context* of the target method. HeMa [40] is a heuristic-based MNR approach that is specially designed for getter/setter functions and delegations. We note that a study recently accepted [42] also utilizes call relations to guide the method name generation. However, it significantly differs from our approach with respect to the technical design: Cognac is supported by the results of systematic empirical studies. Particularly, thanks to the prior knowledge, we can represent all input tokens well with a single encoder. However, without the distinction provided by our prior knowledge, the existing study [42] needs to use totally four encoders to represent different contexts. Such a design leads to a much more complex model than ours (we have calculated that in the encoder part, our model needs to learn 0.8M parameters while such a number of [42] is 12.6M). Consequently, [42] needs more data and time resources to train the model. This is potentially the reason that [42] only evaluates on one of the four MNR benchmarks utilized by us in this study. On the contrary, our model is more generalizable, especially when there is limited training data, which is critical in language models [18].

```java
public static List getMenuList() {
    return loadConfig();
}
```

**Listing 1: The `getMenuList` method in the Addressbook project.**

## 2.3 Method Name Consistency Checking

Given that inappropriate method names may make programs hard to understand [14, 15, 67] or even lead to program defects [3, 4, 13, 20, 53], researchers also try to solve the method name consistency checking (MCC) problem, which is to automatically check whether the method name is consistent with its implementation.

Høst and Østvold [38] exploited the Java language naming convention for extracting rules of method names, which are further used to identify *naming bugs*. Kim *et al.* [41] built a code dictionary from the existing API documents and then detected inconsistent names based on this dictionary. Allamanis *et al.* [6] proposed to learn the domain-specific naming convention from local contexts to enhance the stylistic consistency including identifier naming and formatting. With the idea that similar code should be named with similar names, Liu *et al.* [46] separately encoded method names and method implementations. Then given a method named *m*, they considered two sets which are (1) the set of method names that are close to *m* in the name vector space, and (2) the set of method names whose implementations are close to that of *m* in the code vector space. If the similarity of the two sets is lower than a threshold, *m* is considered as inconsistent. MNire [51] can also be applied to the MCC task by checking the similarity between the recommended name and the original name of the method.

## 2.4 Code Summarization

Apart from generating high quality names for methods, another perspective to enhance the comprehensibility of programs is to automatically generate natural language descriptions for code [33, 58]. Such techniques have been shown to be feasible for solving program comprehension problems in practice. For instance, Panichella *et al.* [52] leveraged the coverage information to summarize test cases, and the generated test summaries helped developers find more bugs. A number of source code summarization works emphasize that limiting the consideration scope to the target method itself is insufficient for generating good summaries. Specifically, McBurney and McMillan [49] improved the effectiveness of code summarization techniques by including the information about how the target methods are invoked. Haque *et al.* [34] considered the sibling methods within the same file with the target method and used an attention mechanism to find words and concepts to utilize in summaries. These works also motivate us to investigate if we can perform the MNR task from a global perspective.

# 3 MOTIVATING EXAMPLES

In this section, we discuss our observations that motivate Cognac on method name recommendations.

**Observation 1.** *Tokens composing the target method's name can be frequently observed from its caller and callee methods.* For instance, in the method `getMenuList` (as shown in Listing 1) of the Addressbook project,[1] there is only one statement calling another

```java
public static List loadConfig() {
    List list = new ArrayList();
    List elementList = DomUtil.getRootElement()
    for (Object obj : elementList) {
        MenuItem menu = new MenuItem();
        menu.setName();
        list.add(menu);
    }
    Collections.sort(list);
    return list;
}
```

**Listing 2: The `loadConfig` method in the Addressbook project.**

```java
public List refreshTicks(Graphics2D g2,
                         AxisState state,
                         Rectangle2D dataArea,
                         RectangleEdge edge) {
    List ticks = null;
    if (RectangleEdge.isTopOrBottom(edge)) {
        ticks = refreshTicksHorizontal(g2, dataArea, edge);
    }
    else if (RectangleEdge.isLeftOrRight(edge)) {
        ticks = refreshTicksVertical(g2, dataArea, edge);
    }
    return ticks;
}
```

**Listing 3: The `refreshTicks` method in the JFreeChart project.**

method named `loadConfig` (as shown in Listing 2) within the method implementation. Unfortunately, for the caller method (i.e., `getMenuList`), the tokens of the method name cannot be found in its implementation, and insufficient information can be extracted from the *implementation context* to help us infer the appropriate name. The only useful information that we can find from itself for guiding method name recommendation is its *interface context*, that is, the return type (i.e., List) contains certain tokens of the method name. On the contrary, abundant useful information can be extracted from its callee (i.e., the `loadConfig` method). Specifically, all three tokens composing the method name (i.e., `get`, `menu`, and `list`[2]) appear in the *implementation context* of the callee method `loadConfig`. Such results reveal that the information from the methods that possess call relations with the target method (e.g., callee methods in this example but in general caller methods can also be included) might provide extra information for us to suggest more appropriate method names for the target method. However, the majority of existing techniques [10, 12, 40] limit the research scope to the target method itself. The only one that considers information beyond the target method is MNire [51], which also takes the class name into consideration. They thus missed the opportunities to leverage more useful information from a global perspective.

**Observation 2.** *Tokens composing the target method's name tend to occur more frequently in specific types of contexts.* For instance, considering the method in Listing 3, which is from the JFreeChart project,[3] its function is to refresh the ticks given a rectangle. This instance confirms the previous observation from Nguyen *et al.* [51] (which also motivates this study) that names of program entities in the *implementation context* usually carry certain meaning that is related to the intention of the target method. Specifically, in this method, the two tokens of the method name (i.e., `refresh` and `ticks`) can both be found in the variables' names or method

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

[2] please note that the analysis of method name tokens is case-insensitive in this paper

[3] https://github.com/jfree/jfreechart

invocations in the method body (e.g., `ticks` and `refreshTicksHorizontal`). Nevertheless, we note that the probabilities of tokens under diverse statement types to compose the method name are different. In this example, lines 6 and 9 are two `IfStatements` while none of the 14 tokens in these two statements contain the tokens of the method name. On the contrary, although the `ReturnStatement` in line 12 contains only one token, it exactly matches the tokens of the method name. Such results indicate that for a specific program entity, the probability of its name to compose the method name may differ significantly according to its context (i.e., the type of the statement where it locates). Therefore, if we use the entity names to predict the tokens that compose the method name, incorporating the context information of each program entity can help us better focus on those critical tokens that have higher probabilities to compose the method name.

## 4 EMPIRICAL STUDY

### 4.1 Experiment Setup

Inspired by our observations, we further performed an empirical study to investigate whether such observations are pervasive among large-scale open source projects. Specifically, we aim to answer the following research questions:

**RQ1:** Can the tokens composing the name of a target method be frequently observed in its caller/callee methods?

**RQ2:** Do the tokens composing the name of a target method tend to occur more frequently in specific contexts than the others?

The answers to these questions provide empirical foundations on (1) whether the information obtained from those caller/callee methods can help us better predict the method names; and (2) whether the information of different program contexts, such as different statement types, can be utilized to better predict the method names. Such foundations are of great importance to our approach designs.

**Data collection and processing.** Following a previous study [51], we chose to use the dataset of 14,317 well-maintained and long-history Java projects on GitHub, which is collected by Allamanis and Sutton [9]. This is a dataset of high-quality since all duplicated projects have already been removed and all selected projects have been forked by GitHub users by at least once. Unlike the previous study [51], we only focused on the source code to reduce potential bias in this study. That means any code from the test files will be excluded in our investigation. As a result, we totally parsed 12,979,389 methods in our experiment. For each investigated method, we collected the method's name and all the names of the entities w.r.t the method's *implementation context* and *interface context*. Finally, all these names were split into tokens based on the camel case and underscore naming conventions, and the obtained tokens were transformed to their lowercase form, following the practices of previous studies [5, 51]. To extract the global contexts, in our study, we established call relations via analyzing the names within each *MethodInvocation* AST node in the project. Note that we excluded constructors from this empirical analysis as well as the evaluation of our approach, following previous studies [12, 40]. The behind intuition is that it is unlikely that developers do not know how to name constructors.

**Table 1: Critical frequencies of tokens from caller/callee.**

| | Number | Frequency |
|---|---|---|
| # Unique caller | 3,279,170 | - |
| # Unique callee | 2,800,498 | - |
| **# Call relations** | 7,034,508 | - |
| # Caller whose tokens in callee | 2,847,864 | 40.5% |
| # Callee whose tokens in caller | 1,712,216 | 24.3% |
| **# Caller whose tokens in callee** | 2,847,864 | - |
| # Caller whose tokens in callee's interface | 1,789,945 | 62.9% |
| # Caller whose tokens in callee's implementation | 2,460,554 | 86.4% |
| # Caller whose tokens in callee's interface uniquely | 387,310 | 13.6% |
| # Caller whose tokens in callee's implementation uniquely | 1,057,919 | 37.1% |
| **# Methods whose tokens cannot be found from itself** | 674,616 | - |
| # Methods whose tokens not in itself but in its caller | 6,000 | 0.9% |
| # Methods whose tokens not in itself but in its callee | 56,808 | 8.4% |

### 4.2 Frequencies of Tokens from Caller/Callee

Critical results from our investigation are illustrated in Table 1. Totally, we found 7,034,508 call relations with 3,279,170 unique callers and 2,800,498 unique callees (since a method can be involved in multiple call relations). Such figures indicate that (1) on average a method is involved in the call relation for more than once, which indicates the pervasiveness of call relations in real-world programs and (2) on average a caller method invokes more than two callees (7,034,508/3,279,170).

From the perspective of a caller, we found that for all the call relations, the tokens composing the caller's method name, if any, occur in the callee for 40.5% of the cases (2,847,864/7,034,508). Such results indicate that there is a significant portion (i.e., around 40%) of callers whose method name tokens can be found in the corresponding callees. We also investigated in which part of the callee (i.e., the *implementation context* or *interface context*) can we observe such tokens. We found that for all the 2,847,864 call relations where the tokens of the caller's name occur in the callees, the tokens occur in the *interface context* of the callees for 1,789,945 cases (62.9%) while in the *implementation context* of the callees for 2,460,554 cases (86.4%). More in-depth analysis reveals that the method name tokens occur in the *interface context* of the callee uniquely (which means tokens occur only in the *interface context* of the callee but not in its *implementation context*) for 387,310 cases while the number of the *implementation context* is 1,057,919. Such results reveal that (1) the *interface context* of the callee method can provide abundant information for inferring the caller's name; and (2) the *implementation context* of the callee method can provide more predictive information for its caller's name than its *interface context*.

From the perspective of a callee, since we know that the method name of the callee can definitely be found in the *implementation context* of its callers (i.e., through method invocations which form the caller/callee relation), we thus only focused on the *interface context* of its callers. We found that for the 7,034,508 call relations, the tokens composing the callee's method name can be found in the *interface context* of the callers for 1,712,216 (24.3%) of the cases. Such results also indicate that the *interface context* of the caller can provide abundant predictive information for its callee's name.

We also investigated the unique contribution from caller/callee methods. Totally we found 674,616 methods where none of the name tokens can be found locally (from the method's *implementation context* and *interface context*). Among them, 6,000 (0.9%) methods can find at least one method name token in their callers' *interface context* and 56,808 (8.4%) methods can find at least one token in their callees. Such results indicate that call relations can uniquely contribute to predicting appropriate method names even if the method name tokens cannot be found locally.

**[Finding-1]** *The method name tokens of considerable proportions of callers/callees (40.5% and 24.3% respectively) can be found in their corresponding callees/callers, which indicate that call relations can contribute significantly to predicting method names. Besides, for methods whose name tokens cannot be found locally, we can find the tokens in their caller/callee methods for a non-negligible proportion of cases (e.g., tokens can be found in callees for 8.4% of them).*

## 4.3 Frequencies of Tokens under Different Contexts

We investigated whether the tokens composing the name of the target method tend to occur more frequently in specific contexts. In our study, we analyzed the context from two granularities, which are the *coarse-grained context* and *fine-grained context*. *Coarse-grained context* denotes the six different sources where the tokens of the target method name can be potentially observed, including the target method's *implementation context*, *interface context*, and *enclosing context*, the *implementation context* of its callees, the *interface context* of its callees, and the *interface context* of its callers. Note that we included the *enclosing context* of the target method in this analysis as well as in our approach since a previous study [51] shows that tokens from this context can help infer the name of the target method. We omitted the *implementation context* of the target method's callers since they already contain the name of the target method. *Fine-grained context* denotes, in this study, the specific type of the statement where each token is extracted. For the *interface context*, we further split it into two sub-categories based on where the tokens are extracted, which are the *ReturnType* and *ParameterType*. Consequently, the detailed context can be represented as a pair of elements, including the source type and the statement type (e.g., ⟨ *Target method implementation context* , *ReturnStatement* ⟩, ⟨ *Callee interface context* , *ReturnType* ⟩). We recorded for each target method (1) the number of tokens under each context and (2) the number of tokens that compose the target method name under each context. The final statistics are summed over the whole dataset, and the probability of a certain type of context is calculated as the number of tokens that compose the target method divided by the total number of tokens under such a context. We utilize the proportion such calculated to approximate the probability. Note that beyond the statement type, there are also other granularities of context information (e.g., the expression type [47]). We chose to use the statement type in this study since the previous study [51] has demonstrated that incorporating too fine-grained program information may reduce the overall effectiveness in the task of method name recommendation.

The results are displayed in Table 2. Be noted that, there are 22 statement types in the Eclipse document [1], while we only list in this table those statements where we observed any method name tokens over the dataset. We noted that the probability of tokens under different contexts to compose method names differs significantly. The maximum value is obtained from the *ReturnStatement* from the *Target method implementation context* with a probability of around one fifth while the minimum probability is from the *TryStatement* from the *Callee implementation context* whose value is only 0.0041. We note that both the *coarse-grained context* and *fine-grained context* contribute to such differences. For instance, taking tokens from the *ReturnType* contexts for consideration, the probability of those tokens extracted from the *Target method interface context* is significantly higher than those from the *Caller interface context* and *Callee interface context* (0.13 vs. less than 0.1). From another perspective, for tokens from the *Target method implementation context*, those from the *ReturnStatement* are much more likely to compose the method name than those from *TryStatement* (a probability of 0.19 vs. 0.01). Such results confirm our intuition in Section 3 that the tokens from diverse contexts differ with each other w.r.t the possibility to compose the name of the target method.

**Table 2: Occurrence probability of tokens from different contexts.**

| Course-grained context | Fined-grained context | # Total | # In method name | Probability |
|---|---|---|---|---|
| Enclosing context | ClassName | 33,128,737 | 5,359,581 | 0.1618 |
| Target method interface context | ReturnType | 13,019,316 | 1,781,975 | 0.1369 |
| | ParameterType | 17,802,134 | 2,135,037 | 0.1199 |
| Target method implementation context | ExpressionStatement | 243,783,458 | 28,579,120 | 0.1172 |
| | VariableDeclarationStatement | 117,214,703 | 11,480,184 | 0.0979 |
| | AssertStatement | 640,604 | 49,664 | 0.0775 |
| | WhileStatement | 1,928,721 | 72,239 | 0.0375 |
| | IfStatement | 54,839,999 | 3,694,167 | 0.0674 |
| | TryStatement | 6,314,330 | 80,367 | 0.0127 |
| | ThrowStatement | 11,390,948 | 620,498 | 0.0545 |
| | SwitchStatement | 787,446 | 61,869 | 0.0786 |
| | SwitchCase | 4,408,811 | 147,524 | 0.0335 |
| | ReturnStatement | 46,543,537 | 8,945,790 | **0.1922** |
| | DoStatement | 197,582 | 6,740 | 0.0341 |
| | ForStatement | 10,456,460 | 647,361 | 0.0619 |
| | FieldDeclaration | 172,232 | 7,677 | 0.0446 |
| | SynchronizedStatement | 326,078 | 23,251 | 0.0713 |
| Caller interface context | ReturnType | 8,343,192 | 412,514 | 0.0494 |
| | ParameterType | 13,629,995 | 557,310 | 0.0409 |
| Callee interface context | ReturnType | 5,703,564 | 442,658 | 0.0776 |
| | ParameterType | 9,599,658 | 462,099 | 0.0481 |
| Callee implementation context | ExpressionStatement | 107,011,128 | 5,698,020 | 0.0532 |
| | VariableDeclarationStatement | 64,401,446 | 3,201,451 | 0.0497 |
| | AssertStatement | 306,603 | 9,371 | 0.0306 |
| | WhileStatement | 1,270,529 | 25,191 | 0.0198 |
| | IfStatement | 39,903,421 | 1,329,074 | 0.0333 |
| | TryStatement | 3,705,067 | 15,162 | **0.0041** |
| | ThrowStatement | 7,717,208 | 229,736 | 0.0298 |
| | SwitchStatement | 378,062 | 17,353 | 0.0459 |
| | SwitchCase | 2,525,321 | 69,378 | 0.0275 |
| | ReturnStatement | 32,419,149 | 2,048,736 | 0.0632 |
| | DoStatement | 131,611 | 2,706 | 0.0206 |
| | ForStatement | 6,741,176 | 247,414 | 0.0367 |
| | FieldDeclaration | 74,361 | 1,318 | 0.0177 |
| | SynchronizedStatement | 212,004 | 7,160 | 0.0338 |

**[Finding-2]** *The probability of a token to compose the target method name differs significantly according to its contexts. The maximum probability is nearly two orders of magnitude higher than the minimum one.*

# 5 METHODOLOGY

In this work, we propose Cognac, a deep learning based approach to recommend high-quality names for a given method, guided by the global and local context information with prior knowledge. As a *program structure independent* approach, which does not require the AST or PDG of programs, the workflow of Cognac is straightforward. Specifically, given a method, Cognac first extracts the targeted tokens from its local contexts as well as its global contexts. When extracting those tokens, Cognac also records the specific contexts (e.g., the type of statements) where such tokens are collected. Cognac then integrates those tokens as a sequence and sends it into a pointer-generator network with the attention mechanism guided by the prior knowledge learned from our empirical study. Finally, Cognac outputs another sequence of tokens which forms the recommended method names. The following introduces Cognac in detail.

## 5.1 Key Ideas

In general, our approach adopts the *abstractive summarization* strategy to generate the tokens of method names from the tokens of both global and local contexts, following the state-of-the-art MNire [51]. Such a paradigm is to rephrase extracted program entity tokens into a short sequence of tokens, which forms the method name to be recommended. At each timestep, a learned attention weight is used to decide which input tokens to focus on when generating the next output token. Our approach, despite falling into such a workflow, embodies the following two key ideas.

First, in addition to considering the program entity tokens and the associated contexts extracted from the target method (which are denoted as the ***local context***), we propose to include tokens and their contextual information from other methods that possess call relations with the target one as the ***global context***. Such a design can utilize more useful information from other relevant methods in the project that might contribute to inferring the name of the target method. Second, we utilize the empirical results as the ***prior knowledge*** to help us better focus on the critical tokens. Recall that the probabilities of tokens under diverse contexts to compose method names are different, which have been revealed by our large-scale empirical analysis. Such probabilities are hence utilized as the prior knowledge, which serves for two main purposes. On the one hand, it is integrated with the learned attention weight to jointly decide which input tokens to focus on under each timestep in the network. We postulate that such prior knowledge could guide the model to focus more on those critical tokens and thus improve the effectiveness of the learned model (confirmed in Section 6.4). On the other hand, we leverage the prior knowledge to limit the number of tokens that are extracted from the callers/callees, and thus our utilized global context is ***lightweight***. The behind intuition is that one method can possess call relations with multiple methods (cf. Section 4.2), therefore, the input token sequence would be too long if taking all tokens from the implementations of caller/callee methods into consideration. Such long sequence inputs may introduce potential noises and reduce the generality of the learned model according to previous studies [11, 59]. We have gained the observation that the caller/callee methods' *interface context* can already provide sufficient information to infer the name of target method (cf. Section 4.2). We therefore decide to consider the *interface context* of the caller/callee methods as well as the top ten tokens in the *implementation context* of each callee method with the highest probabilities to compose method names (we omit the *implementation context* of the caller methods to avoid data leakage as aforementioned). The number is set to ten empirically: we performed a pre-study experiment using 5, 10, and 20 tokens from the *implementation context* of each callee method separately and found that selecting ten tokens achieves the optimum. We also tried to keep all the tokens in each callee but observed inferior results compared to that of using ten tokens (see Section 7.2). Note that in general, tokens from the *implementation context* of the target method possess higher probabilities to compose the method name than those from the *implementation context* of its callee methods (cf. Table 2). We therefore take all tokens from the *implementation context* of the target method into consideration.

## 5.2 Source Extraction

Given a method, the first step of Cognac is to extract token sequence that will be used to infer the method name. We respectively extract the entity names from the *enclosing context*, the *interface context* of the callers, the *interface context* of the callees, the *implementation context* of the callees, the *interface context* of the target method, and its *implementation context* (resulting in totally six sources), after which these names are broken into tokens based on the camel cases and underscore naming conventions. Note that to restrict the length of the input sequence, we limit the number of tokens extracted from the *implementation context* of each callee method to be ten. Such tokens are ranked by their probabilities to compose the method name according to their detailed contexts (cf. Table 2) and for tied tokens, they are further ranked by their orders in the token sequence of the callee method.

For each token, we also assign it with an indicator according to the detailed context where it is extracted, which could result in totally 35 different indicators shown in Table 2 (e.g., ⟨ *Enclosing context , ClassName* ⟩, ⟨ *Callee implementation context , ReturnStatement* ⟩). Such indicators will be utilized to provide the *prior knowledge* in the attention mechanism in our model.

## 5.3 Pointer-Generator Network

A qualified method name generation model should possess two key features: first, it should be able to generate out-of-vocabulary (OOV) tokens in its output considering the uniqueness of specific methods; second, it should be able to generate tokens that does not appear in the input sequence since a non-negligible amount of method name tokens cannot be found from our considered contexts [51]. Therefore, we adopt a novel pointer-generator network [57] in the design of Cognac since it satisfies the two requirements. Figure 1 illustrates the overview of the model architecture. Due to page limit, we only briefly introduce this model in the paper, and more details could be referred to the existing work [57].

*Context vector calculation.* As shown in the bottom left part in Figure 1, the inputs of Cognac are a token sequence where tokens are extracted from both the *global context* and *local context* along with the contextual indicator (i.e., the probability of the token under such a context as revealed in the empirical study) for each token. The encoder then embeds the tokens into a vector $x = (x_1, x_2, \ldots, x_m)$ and then encodes them into a hidden representation $h = (h_1, h_2, \ldots, h_m)$ through a single-layer bidirectional LSTM. At the same time, the value of the context indicator of each input token, which is listed in Table 2 according to the detailed context of each input token, is recorded as $v_c = (v_{c_1}, v_{c_2}, \ldots, v_{c_m})$. At each timestep $t$, the attention distribution over the whole input sequence is calculated via summing up the learned distribution and the prior knowledge recorded in $v_c$:

$$a^t = softmax(e^t) + softmax(v_c) \tag{1}$$

where $e^t$ is learned using the encoder hidden state and decoder hidden state at this step while $v_c$ represents the prior knowledge which is the probability of each input token to compose the method

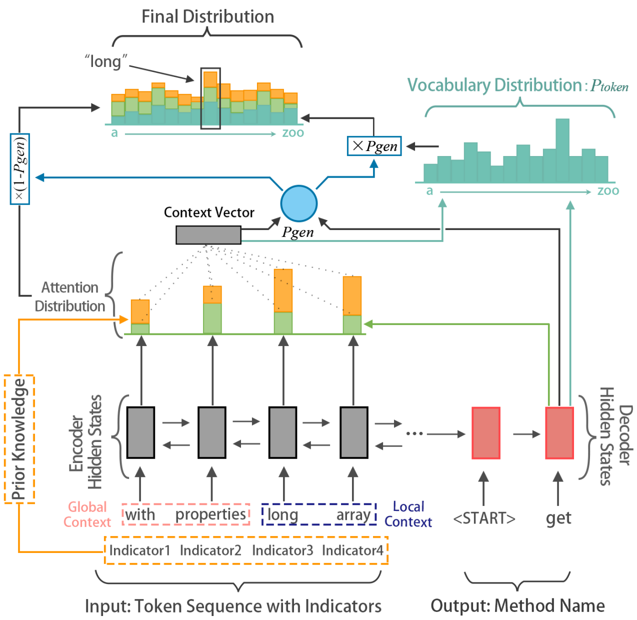


**Figure 1: Architecture of Cognac.**

name. Then the attention distribution is used to produce the *context vector* $h_t^*$ which can be regarded as the representation of what has been read from the input at this step: $h_t^* = \sum_i a_i^t h_i$.

*Output generation.* The obtained *context vector* serves for two main purposes. First, it is jointly learned with the encoder hidden state and decoder hidden state to produce the generation probability $p_{gen} \in [0, 1]$ at this step, which denotes the probability of generating tokens from the *fixed vocabulary*, which is the set of tokens that can be observed in the training dataset. On the contrary, 1 - $p_{gen}$ denotes the probability of copying a token directly from the input sequence, which is to select a token from the input as the output of the current timestep. Second, it is concatenated with the decoder hidden state to learn the probability distribution over all tokens in the *fixed vocabulary* ($P_{token}$). Finally, the probability of outputting the token $w$ at this step is calculated as:

$$P(w) = p_{gen}P_{token}(w) + (1 - p_{gen}) \sum_{i:w_i=w} a_i^t \tag{2}$$

where the first part denotes the probability of generating $w$ from the *fixed vocabulary* while the second part denotes the probability of copying $w$ from the input.

*Loss calculation.* During training, the overall loss for the whole sequence is calculated as the average loss at each step, which is the negative log likelihood of the oracle word $w_t^o$ for that step:

$$loss = \frac{1}{T}\sum_{t=0}^{T}(-logP(w_t^o)) \tag{3}$$

# 6 EVALUATION

## 6.1 Research Questions

We seek to answer the following research questions to assess the effectiveness of Cognac:

**RQ3:** How does Cognac perform on the method name recommendation task compared with the state-of-the-art?
**RQ4:** How does Cognac perform on the method name consistency checking task compared with the state-of-the-art?
**RQ5:** To what extent do diverse design decisions affect the performance of Cognac on the above two tasks?

## 6.2 The MNR Task (RQ3)

*6.2.1 Dataset.* To evaluate the effectiveness of Cognac on the method name recommendation task, we in total used four different datasets. We first reused three widely-adopted datasets in the community constructed by Alon *et al.* [10], which are named as *Java-small*, *Java-med*, and *Java-large*, containing 11, 1K, and 9.5K Java projects from GitHub respectively. To evaluate the effectiveness of MNire, Nguyen *et al.* built another dataset containing more than 10K Java projects [51]. Due to the unavailability of the source code of MNire, we can only compare with its reported performance. Therefore, in our study, we chose to reuse their dataset for fair comparison against the state-of-the-art MNire. Note that the MNire's dataset does not contain fixed training and testing data. We thus randomly split all the projects in this dataset into 9,772 training and 450 testing projects, following Nguyen *et al.* [51].

It should be noted that in all these datasets, the training and test examples are shuffled by projects, to avoid the performance enhancement caused by file-based shuffling [7, 10, 40].

*6.2.2 Metrics.* Following previous studies, we focused on *Precision*, *Recall*, and *F-score* for measuring the performance of Cognac [12, 51]. In particular, for a specific method whose oracle name is $o$ while the recommended name is $r$, its precision, recall, and F-score are calculated as: $precision = \frac{|token(r) \cap token(o)|}{|token(r)|}$, $recall = \frac{|token(r) \cap token(o)|}{|token(o)|}$, $F-score = \frac{2 \times precision \times recall}{precision+recall}$, respectively, where $token(x)$ returns the tokens in the name $x$ split by the camel case and underscore naming conventions. Then the performances on the whole dataset are computed as the average values of all the methods in the dataset.

*6.2.3 Results.* The results of Cognac on the four datasets are listed in Table 3 where we also present the results of ten state-of-the-art approaches. We performed a thorough literature review to include as many state-of-the-art approaches as possible for performance comparison. We do not include Liger [61] since it is applied to C# and Python languages and the source code is unavailable. Note that we only list the results of other approaches on the datasets where they have also been evaluated.

We found that the values achieved by Cognac w.r.t all the three metrics are higher than those from the state-of-the-art on all the four different datasets. Specifically, Cognac outperforms the state-of-the-art w.r.t *F-score* by at least 5.0% (63.2% vs. 60.2% from Sequence GINN), 9.2% (60.8% vs. 55.7% from TreeCaps), 8.2% (66.3% vs. 61.3% from TreeCaps), and 7.7% (68.5% vs. 63.6% from MNire) on the four datasets respectively. We noted that some existing approaches can achieve similar performance w.r.t a specific metric compared with Cognac (e.g., the precision of Code2seq and TreeCaps are close to that of Cognac on the *Java-med* dataset). Nevertheless, Cognac can achieve both high precision and high recall, which leads to an overall significant better performance (i.e., *F-score*) than existing

**Table 3: Effectiveness of Cognac on the MNR task (in %).**

| Dataset | Approach | Pre. | Rec. | F-score |
|---|---|---|---|---|
| Java-small | Sequence GINN [63] | 64.8 | 56.2 | 60.2 |
| | Sequence GNN [30] | - | - | 51.3 |
| | GGNN [7] | 40.3 | 35.3 | 36.9 |
| | Code2vec [12] | 23.4 | 22.0 | 21.4 |
| | Code2seq [10] | 50.4 | 35.4 | 42.6 |
| | TreeCaps [19] | 52.6 | 41.4 | 46.8 |
| | GREAT [35] | 47.3 | 40.0 | 43.6 |
| | TBCNN [50] | 40.9 | 31.8 | 35.5 |
| | Cognac | **67.1** | **59.7** | **63.2** |
| Java-med | HeMa [40] | 39.9 | 23.5 | 29.6 |
| | GGNN [7] | 50.1 | 41.3 | 45.3 |
| | Code2vec [12] | 36.4 | 27.9 | 31.9 |
| | Code2seq [10] | 62.6 | 46.8 | 53.7 |
| | TreeCaps [19] | 64.4 | 48.9 | 55.7 |
| | GREAT [35] | 57.2 | 44.1 | 51.4 |
| | TBCNN [50] | 45.2 | 41.4 | 43.2 |
| | Cognac | **64.8** | **57.3** | **60.8** |
| Java-large | GGNN [7] | 50.2 | 44.3 | 46.2 |
| | Code2vec [12] | 44.2 | 38.3 | 41.6 |
| | Code2seq [10] | 63.3 | 54.0 | 59.0 |
| | TreeCaps [19] | 66.9 | 56.3 | 61.3 |
| | GREAT [35] | 61.4 | 55.9 | 58.3 |
| | TBCNN [50] | 58.2 | 40.9 | 49.4 |
| | Cognac | **71.4** | **61.9** | **66.3** |
| MNire's | MNire [51] | 66.4 | 61.1 | 63.6 |
| | Cognac | **70.2** | **66.8** | **68.5** |

Data of other approaches are extracted from the recent studies [19, 30, 40, 51, 63]. "-" denotes no relevant information.

approaches. A concrete example here is that when trained on the MNire's dataset, Cognac recommends `listMenu` for the method as shown in Listing 1, achieving a 100% precision and a recall around 70%. Such a name is semantically similar to the developer-provided one, which indicates the practical usefulness of Cognac. Such a name, however, cannot be generated if the information from the callee method is ignored, indicating the significance of our concerned call relations.

A notable phenomenon is that the performances of Cognac on those datasets with more projects (i.e., *Java-large* and the MNire's dataset) are better than those from the datasets with fewer projects (i.e., *Java-small* and *Java-med*). Such results indicate that the sufficiency and diversity of the training data can help enhance the generality of the learned model.

Cognac *outperforms the state-of-the-art approaches by at least 5.0%, 9.2%, 8.2%, and 7.7% on the four datasets respectively w.r.t F-score. Moreover, its performances w.r.t different metrics all exceed those from the existing state-of-the-art on all the datasets.*

## 6.3 The MCC Task (RQ4)

*6.3.1 Dataset.* To evaluate the effectiveness of Cognac on the method name consistency checking task, we used the dataset collected by Liu *et al.* [46],which is also used to evaluate the state-of-the-art MNire [51]. This dataset is collected from 430 well-maintained Java open-source projects from four communities, namely Apache, Spring, Hibernate, and Google. For the training data, they select totally 2,116,413 methods, excluding main methods and constructors. For the testing data, they select totally 2,805 methods whose names are inconsistent by parsing the commit history of each project which satisfy the following two requirements: (1) the method name should be changed in a commit without any modification on the body code, which ensures the change is to fix the method name; and (2) the method name and body code should become stable after the change, which ensures the fixed version of the name is not revealed to be buggy later on.

After training Cognac on the training data, we randomly split the testing data into two classes (note that the testing data splitting is also random in previous studies [46, 51]). For the *inconsistent class* (*IC*), we used the buggy versions of the method names and labeled them as inconsistent. For the *consistent class* (*C*), we used the fixed versions of the method names and labeled them as consistent.

*6.3.2 Metrics.* To apply Cognac on the MCC task, we adopted the same strategy as MNire, which computes the similarity $Sim(r, o)$ between the recommended name $r$ and the original name $o$ (Note that for the *inconsistent class* (*IC*), the original name $o$ is the buggy method name, while for the *consistent class* (*C*), it is the fixed method name). Specifically, such a similarity is defined as the portion of the tokens that are shared between $r$ and $o$: $Sim(r, o) = \frac{|token(r) \cap token(o)|}{(|token(r)|+|token(o)|)/2}$. The consistency of this method is then determined using an empirically-decided threshold $T$. In particular, if $Sim(r, o) \leq T$, the method is considered as inconsistent, otherwise it is classified as consistent.

To measure the performance on the MCC task, we used the same metric as previous studies [46, 51], including precision, recall, and *F-score* for both the *IC* and *C* classes as well as the total *accuracy*. The above metrics are computed based on the following numbers. **True Positive (TP):** an inconsistent name in *IC* is identified as inconsistent; **False Positive (FP):** a consistent name in *C* is identified as inconsistent; **True Negative (TN):** a consistent name in *C* is identified as consistent; **False Negative (FN):** an inconsistent name in *IC* is identified as consistent. Therefore, for the *IC* class, $Precision = \frac{|TP|}{|TP|+|FP|}$, and $Recall = \frac{|TP|}{|TP|+|FN|}$. For the *C* class, $Precision = \frac{|TN|}{|TN|+|FN|}$, and $Recall = \frac{|TN|}{|TN|+|FP|}$. For both the *IC* and *C* classes, the *F-score* is calculated as $\frac{2 \times Precision \times Recall}{Precision + Recall}$. The *accuracy* on the whole dataset is defined as $\frac{|TP|+|TN|}{|TP|+|FP|+|TN|+|FN|}$. Note that whether Cognac identifies a specific method name as consistent or not depends on the similarity threshold $T$. In the previous study [51], the authors vary the similarity threshold $T$ in the range of $(0.85, 1)$, and separately report the maximum values of *F-score* on the *IC* and *C* classes and the maximum *accuracy*. However, we never know a method name is consistent or not before the detection in practice. Therefore, we decide to set the $T$ as a fixed value. Specifically, in our study, to determine the threshold, we chose the value where the overall *accuracy* reaches the maximum, which is is 0.85 in this study.

*6.3.3 Results.* The results of Cognac and the existing state-of-the-art are listed in Table 4. We noted that Cognac achieves the highest overall *accuracy*, which outperforms MNire by 11.2% (76.6% vs. 68.9%). For the *IC* class, Cognac's precision, recall and *F-score* are 9.4%, 4.3%, and 7.3% higher than those of MNire respectively. Such results reveal that compared with MNire, Cognac can detect more

Table 4: Effectiveness of Cognac on the MCC task (in %).

| | | Liu *et al.* [46] | MNire [51] | Cognac |
|---|---|---|---|---|
| IC | Precision | 56.8 | 62.7 | **68.6** |
| | Recall | 84.5 | 93.6 | **97.6** |
| | F-score | 67.9 | 75.1 | **80.6** |
| C | Precision | 51.4 | 56.0 | **96.0** |
| | Recall | 72.2 | **84.2** | 55.6 |
| | F-score | 60.0 | 67.3 | **70.4** |
| Accuracy | | 60.9 | 68.9 | **76.6** |

Table 5: Performance of variants of Cognac on the MNR task (in %).

| Model \ Dataset | Java-small | | Java-med | | Java-large | | MNire's | |
|---|---|---|---|---|---|---|---|---|
| | F | ↓ | F | ↓ | F | ↓ | F | ↓ |
| No caller information | 60.1 | 4.8 | 57.5 | 5.4 | 62.9 | 5.2 | 65.0 | 5.1 |
| No callee information | 57.7 | **8.6** | 54.7 | **10.0** | 59.9 | **9.6** | 62.1 | **9.3** |
| No prior knowledge | 59.3 | 6.2 | 56.2 | 7.6 | 61.5 | 7.3 | 63.8 | 6.9 |
| Cognac (original model) | 63.2 | | 60.8 | | 66.3 | | 68.5 | |

↓ denotes performance degradation.

inconsistent method names and the method names that are labelled as inconsistent are more likely to be the real inconsistent ones.

For the *C* class, we observed that the precision of Cognac is much higher than that of MNire (96.0% vs. 56.0%) while the recall of Cognac is much lower than that of MNire (55.6% vs. 84.2%). Such phenomenon could be caused by the fact that MNire adopts a varying threshold $T$. Specifically, for MNire, the threshold used for the *C* class is lower than that for the *IC* class, the consequence of which is that more names are labelled as consistent (we recall that a method name is labelled as consistent if the similarity exceeds the threshold, hence, the lower the threshold is, the more names will be labelled as consistent). Consequently, its recall w.r.t the *C* class is high. On the contrary, we set a fixed value for $T$, which may prevent many method names from being labelled as consistent. Nevertheless, Cognac still achieves the highest *F-score* on this class, which exceeds that of MNire by 4.6% (70.4% vs. 67.3%).

*With a fixed threshold,* Cognac *still outperforms the state-of-the-art approaches on the MCC task significantly. Specifically, its overall accuracy exceeds that of MNire by 11.2%, and it outperforms MNire by 7.3% w.r.t F-score for detecting inconsistent method names.*

## 6.4 Ablation Study (RQ5)

*6.4.1 Experiment Setting.* We in this RQ investigated the influences from three factors on the performance of Cognac, which are the tokens from the caller/callee methods respectively and the prior knowledge. Note that in the ablation study, the contribution of the prior knowledge refers to its guidance on method name generation (see Equation 1). In the first two experiments, we omitted tokens from the caller methods and callee methods respectively in the input token sequence. In the last one, we omitted the prior knowledge, which means we only used the learned matrix $e^t$ to decide the attention distribution in Equation 1. We performed such experiments on both the MNR task and MCC task.

*6.4.2 Results.* Results of the ablation study on the MNR task are demonstrated in Table 5. Generally speaking, all our model decisions make contributions to the final performance, more or less. For instance, if we do not use the prior knowledge to guide the attention weight putting on each input token, the overall performance w.r.t *F-score* will be decreased by 6.2% ~ 7.6% on the four datasets.

Table 6: Performance of variants of Cognac on the MCC task (in %).

| Model | IC F | IC ↓ | C F | C ↓ | Accuracy | ↓ |
|---|---|---|---|---|---|---|
| No caller information | 79.2 | 1.7 | 65.7 | 6.7 | 74.1 | 3.3 |
| No callee information | 77.4 | **4.0** | 64.2 | **8.8** | 72.4 | **5.5** |
| No prior knowledge | 79.3 | 1.6 | 65.5 | 7.0 | 74.1 | 3.3 |
| Cognac (original model) | 80.6 | | 70.4 | | 76.6 | |

↓ denotes performance degradation.

We noted that the information from the callee methods contributes the most to the overall performance of Cognac among the three factors, without which the *F-score* will degrade the most on all the four datasets. Specifically, if the tokens from the callee methods are not included, the *F-score* of Cognac will be decreased by 10% on the *Java-med* dataset, which is the largest degradation we witnessed in this ablation study. On the other hand, the contribution from the caller methods is relatively small, without which the degradation is only 4.8% ~ 5.4% on the four datasets. Such results could be caused by the fact that we only include the *interface context* of the caller methods (recall that we have excluded the tokens of the *implementation context* from the callers to avoid data leakage). However, the *implementation context* of the callee methods are included in our approach since there is no data leakage. We also noted that the contribution from our prior knowledge is non-negligible, without which the performances of Cognac could not exceed those achieved by the existing approaches. For instance, Cognac achieves an *F-score* of 59.3% without the prior knowledge on the *Java-small* dataset while the value of *Sequence GINN* is 60.2%. This confirms our intuition that incorporating the context information with prior knowledge can help our model better capture the critical information and thus improve its effectiveness.

Similar trends can be observed from the results of the ablation study on the MCC task, which are shown in Table 6. For the MCC task, the callee information is still the major part that contributes to the overall performance of Cognac without which the *accuracy* and the *F-scores* on the *IC* and *C* classes will be decreased by 5.5%, 4.0% and 8.8% respectively. The prior knowledge still plays a significant role. For instance, without the guidance from the prior knowledge, the *F-score* of Cognac on the *C* class will reach only 65.5% (a reduction of 7.0%), lower than that of MNire (67.3%).

*All the design decisions in* Cognac *contribute to its outstanding performance, among which the information from the callee methods is the most rewarding one. Specifically, if omitting the tokens from the callee methods,* Cognac *will suffer from decreases of 8.6%, 10.0%, 9.6%, and 9.3% w.r.t F-score on the four datasets on the MNR task as well as a decrease of 5.5% w.r.t accuracy on the MCC task.*

# 7 DISCUSSION

## 7.1 Performance Enhancement from the Pointer-Generator Model

Note that the seq2seq model in the existing approach MNire is simple: it is only capable of generating tokens from the *fixed vocabulary* while is unable to copy a token from the input. On the contrary, our Cognac adopts a pointer-generator model which is capable for both generating tokens from the *fixed vocabulary* and copying from the input tokens. Nonetheless, the superiority of Cognac is majorly attributed by the caller/callee information and the utilized prior knowledge. Specifically, we demonstrate this via the following experiment. We implemented a simple seq2seq model which still incorporates the prior knowledge (i.e., the way to calculate the attention weight is identical to the original Cognac). The difference between this model and the original Cognac is that in Equation 2 the $p_{gen}$ always equals to 1, which means that it is incapable of copying tokens from the input. We then trained and tested this model on the MNire's dataset. The experimental results show that this model achieves an overall performance of 67.8% w.r.t *F-score*, which is much higher than that from MNire (63.6%) but only slightly lower than that from the original Cognac (68.5%). This is reasonable considering that the pointer-generator model is proposed to mainly deal with the OOV tokens while the number of OOV tokens could be rather limited if the training dataset is large enough (in our study, it contains methods from 9,772 projects). Such results indicate that Cognac outperforms the existing approaches mainly due to the integrated caller/callee information and the prior knowledge. The adopted pointer-generator model helps it reach the optimum.

## 7.2 Rationality of the Lightweight Strategy

In our approach, we utilize the *global context* in a lightweight manner that is to limit the number of tokens extracted from the *implementation context* of each callee method to be 10. The behind intuition is that we have demonstrated through our empirical analysis that on average a caller calls more than two callees, and thus the input token sequences for these methods could be rather long if we consider all their implementations. Training on such long input sequences could reduce the generality of the learned model as revealed by the previous studies [11, 59]. To demonstrate the rationality of this decision, we performed another experiment where we used all tokens from the *implementation context* of the callee methods in Cognac and then assessed its performances w.r.t the MNR task. Results show that Cognac achieves 59.8%, 57.8%, 61.1%, and 64.2% respectively on the four different datasets for the MNR task w.r.t *F-score*, thus witnessing a degradation of 5.4%, 4.9%, 7.8%, 6.3%, respectively. This could be explained as too much noisy data in the input reduces the generality of Cognac. Such results reveal that the performances of Cognac will be significantly compromised if the information is utilized inappropriately, therefore, our lightweight strategy to utilize the *global context* is reasonable.

## 7.3 Threats to Validity

A threat to validity is that we only focus on the Java programming language (PL). Hence, all findings and evaluation results are restricted to this domain. Being that said, the principle of Cognac it not limited to one specific PL. It would be interesting to investigate the performance of Cognac on other PLs such as C# and compare against other existing approaches like Liger [61]. However, it requires another large-scale empirical analysis to build the prior knowledge, and thus we leave it as future work.

Another threat is that it is impossible to ensure that all of the methods in our empirical dataset have consistent names. Consequently, the constructed prior knowledge might be biased. To address this threat, we choose to use a dataset composed of high-quality and well-maintained open source projects [9]. Furthermore, literature approaches always assume that most of the names from top-ranked, high-quality projects are good [10, 12, 42, 51]. Such an assumption can be backed up by the fact that during preparing the dataset for the MCC task, only 2,805 among totally 2,116,413 methods (i.e., 0.13%) are found to be inconsistent. Moreover, such noises are actually acceptable for learning-based techniques since they are supposed to learn common features from the majority instead of the minority. Therefore, even if names in our datasets are not always of high-quality, their impacts are limited.

Besides, we choose to use the statistical metrics (e.g., precision and recall) as adopted by previous studies [10, 12, 51] to evaluate and compare the performance of Cognac. Unfortunately, whether a recommended name is really helpful for developers in practice remains unknown and is left as our future work.

## 7.4 Application Scenario

We briefly discuss the application scenario of Cognac. We recall that the inputs of our approach are the class name, the interface context of callers, the interface/implementation context of callees, and the interface/implementation context of the target method. That means we actually do not need to know how the target method should be invoked since we excluded the implementation of callers. Therefore, we do not need to arbitrarily name a method and then run our approach to verify the name after implementation. On the contrary, we can perform *just-in-time* name recommendation after obtaining the implicit calling relations to acquire the global contexts, which is also required by [42]. Therefore, the application scenarios of Cognac are *method-name-recommendation* after obtaining the calling relations and *method-name-inconsistency-checking* after a whole project has been implemented.

# 8 CONCLUSION

We introduce Cognac, a deep learning based approach to recommend high-quality method names. The key observations in this paper obtained through a large-scale empirical analysis are: (1) call relations can be utilized for better inferring method names; and (2) tokens under diverse specific contexts generally possess different probabilities to compose the method name. Therefore, we implemented Cognac, which takes into consideration the caller/-callee methods of the target one to incorporate more information and utilizes the empirical results as prior knowledge to better focus on critical information. Evaluation results show that Cognac can achieve significantly better results than the state-of-the-art on both the tasks of *method name recommendation* and *method name consistency checking*.

**Artifacts:** All data in this study are publicly available at:

**https://github.com/ShangwenWang/Cognac**.

# ACKNOWLEDGMENTS

The authors greatly thank the anonymous reviewers for their constructive comments. This work is supported by the National Natural Science Foundation of China No.62002125, No.61872445, and No.61672529.

# REFERENCES


[1] 2021. Eclipse Statement. https://help.eclipse.org/2020-12/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FStatement.html.

[2] 2021. Find Bugs in Java programs. http://findbugs.sourceforge.net/.

[3] Surafel Lemma Abebe, Venera Arnaoudova, Paolo Tonella, Giuliano Antoniol, and Yann-Gael Gueheneuc. 2012. Can lexicon bad smells improve fault prediction?. In *2012 19th Working Conference on Reverse Engineering*. IEEE, 235–244.

[4] Surafel Lemma Abebe, Sonia Haiduc, Paolo Tonella, and Andrian Marcus. 2011. The effect of lexicon bad smells on concept location in source code. In *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*. Ieee, 125–134.

[5] 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*. ACM, 38–49. https://doi.org/10.1145/2786805.2786849

[6] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 281–293. https://doi.org/10.1145/2635868.2635883

[7] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *Proceedings of the 6th International Conference on Learning Representations*. OpenReview.net.

[8] 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*. JMLR.org, 2091–2100.

[9] Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. 207–216. https://doi.org/10.1109/MSR.2013.6624029

[10] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *Proceedings of the 7th International Conference on Learning Representations*. OpenReview.net.

[11] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A general path-based representation for predicting program properties. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 404–419. https://doi.org/10.1145/3192366.3192412

[12] 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), 40:1–40:29. https://doi.org/10.1145/3290353

[13] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N Nguyen, and Mira Mezini. 2018. A systematic evaluation of static api-misuse detectors. *IEEE Transactions on Software Engineering* 45, 12 (2018), 1170–1188.

[14] Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. 2016. Linguistic antipatterns: What they are and how developers perceive them. *Empirical Software Engineering* 21, 1 (2016), 104–158.

[15] Venera Arnaoudova, Laleh M Eshkevari, Massimiliano Di Penta, Rocco Oliveto, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. 2014. Repent: Analyzing the nature of identifier renamings. *IEEE Transactions on Software Engineering* 40, 5 (2014), 502–532.

[16] Venera Arnaoudova, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. 2013. A New Family of Software Anti-patterns: Linguistic Anti-patterns. *2013 17th European Conference on Software Maintenance and Reengineering*, 187–196.

[17] Gabriele Bavota, Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. 2013. Methodbook: Recommending move method refactorings via relational topic models. *IEEE Transactions on Software Engineering* 40, 7 (2013), 671–694.

[18] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).

[19] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021. TreeCaps: Tree-Based Capsule Networks for Source Code Processing. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence*.

[20] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2009. Relating identifier naming flaws and code quality: An empirical study. In *2009 16th Working Conference on Reverse Engineering*. IEEE, 31–35.

[21] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2010. Exploring the influence of identifier names on code quality: An empirical study. In *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 156–165.

[22] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2011. Improving the tokenisation of identifier names. In *Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP)*. Springer, 130–154.

[23] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2011. Mining java class naming conventions. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. 93–102. https://doi.org/10.1109/ICSM.2011.6080776

[24] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2013. INVocD: Identifier name vocabulary dataset. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 405–408.

[25] Florian Deissenboeck and Markus Pizka. 2006. Concise and consistent naming. *Software Quality Journal* 14, 3 (2006), 261–282.

[26] Aryaz Eghbali and Michael Pradel. 2020. No Strings Attached: An Empirical Study of String-related Software Bugs. *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 956–967.

[27] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. 2018. The effect of poor source code lexicon and readability on developers' cognitive load. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*.

[28] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ACM, 313–324. https://doi.org/10.1145/2642937.2642982

[29] Chunrong Fang, Zixi Liu, Yangyang Shi, Jingfang Huang, and Qingkai Shi. 2020. Functional code clone detection with syntax and semantics fusion learning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[30] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Structured Neural Summarization. In *Proceedings of the 7th International Conference on Learning Representations*. OpenReview.net.

[31] Malcom Gethers, Trevor Savage, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. 2011. CodeTopics: which topic am I coding now?. In *Proceedings of the 33rd International Conference on Software Engineering*. 1034–1036.

[32] Latifa Guerrouj, Zeinab Kermansaravi, Venera Arnaoudova, Benjamin CM Fung, Foutse Khomh, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. 2017. Investigating the relation between lexical smells and change-and fault-proneness: an empirical study. *Software Quality Journal* 25, 3 (2017), 641–670.

[33] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the use of automated text summarization techniques for summarizing source code. In *2010 17th Working Conference on Reverse Engineering*. IEEE, 35–44.

[34] Sakib Haque, Alexander LeClair, Lingfei Wu, and Collin McMillan. 2020. Improved automatic summarization of subroutines via attention to file context. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 300–310.

[35] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2020. Global Relational Models of Source Code. In *Proceedings of the 8th International Conference on Learning Representations (ICLR)*. OpenReview.net.

[36] Yoshiki Higo and Shinji Kusumoto. 2012. How often do unintended inconsistencies happen? Deriving modification patterns and detecting overlooked code fragments. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 222–231.

[37] Johannes C. Hofmeister, J. Siegmund, and Daniel V. Holt. 2017. Shorter identifier names take longer to comprehend. *Empirical Software Engineering* 24 (2017), 417–443.

[38] Einar W. Høst and Bjarte M. Østvold. 2009. Debugging Method Names. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP)*. 294–317.

[39] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep Code Comment Generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*.

[40] Lin Jiang, Hui Liu, and He Jiang. 2019. Machine Learning Based Recommendation of Method Names: How Far are We. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 602–614. https://doi.org/10.1109/ASE.2019.00062

[41] Suntae Kim and Dongsun Kim. 2016. Automatic identifier inconsistency detection using code dictionary. *Empirical Software Engineering* 21, 2 (2016), 565–604.

[42] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. A Context-based Automated Approach for Method Name Consistency Checking and Suggestion. In *Proceedings of the 43rd International Conference on Software Engineering*.

[43] Yi Li, Shaohua Wang, Tien N Nguyen, and Son Van Nguyen. 2019. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 162:1–162:30. https://doi.org/10.1145/3360588

[44] Bo Lin, Shangwen Wang, Kui Liu, Xiaoguang Mao, and Tegawendé F. Bissyandé. 2021. Automated Comment Update: How Far are We?. In *The 29th IEEE/ACM International Conference on Program Comprehension (ICPC)*. 36–46.

[45] Hui Liu, Qiurong Liu, Cristian-Alexandru Staicu, Michael Pradel, and Yue Luo. 2016. Nomen est Omen: Exploring and Exploiting Similarities between Argument and Parameter Names. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 1063–1073.

[46] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Tae-young Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. 2019. Learning to spot and refactor inconsistent method names. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE, 1–12. https://doi.org/10.1109/ICSE.2019.00019

[47] Kui Liu, Dongsun Kim, Anil Koyuncu, Li Li, Tegawendé F Bissyandé, and Yves Le Traon. 2018. A closer look at real-world patches. In *Proceedings of the 34th International Conference on Software Maintenance and Evolution*. IEEE, 275–286. https://doi.org/10.1109/ICSME.2018.00037

[48] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the Usage of Text-To-Text Transfer Transformer to Support Code-Related Tasks. In *Proceedings of the 43rd International Conference on Software Engineering*.

[49] Paul W McBurney and Collin McMillan. 2014. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension*. 279–290.

[50] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 30.

[51] 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*. 1372–1384.

[52] Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C Gall. 2016. The impact of test case summaries on bug fixing performance: An empirical investigation. In *Proceedings of the 38th International Conference on Software Engineering*. 547–558.

[53] Michael Pradel and Koushik Sen. 2018. DeepBugs: A Learning Approach to Name-Based Bug Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 147 (2018), 25 pages. https://doi.org/10.1145/3276517

[54] Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. 2018. A comprehensive model for code readability. *Journal of Software: Evolution and Process* 30, 6 (2018).

[55] Simone Scalabrino, Christopher Vendome, and Denys Poshyvanyk. 2019. Automatically assessing code understandability. *IEEE Transactions on Software Engineering* (2019).

[56] Andrea Schankin, A. Berger, Daniel V. Holt, Johannes C. Hofmeister, T. Riedel, and M. Beigl. 2018. Descriptive Compound Identifier Names Improve Source Code Comprehension. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*.

[57] Abigail See, Peter J. Liu, and Christopher D. Manning. 2017. Get To The Point: Summarization with Pointer-Generator Networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*. 1073–1083.

[58] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 43–52.

[59] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*. 6000–6010.

[60] Yaza Wainakh, Moiz Rauf, and Michael Pradel. 2021. IdBench: Evaluating Semantic Representations of Identifier Names in Source Code. In *Proceedings of the 43rd International Conference on Software Engineering*.

[61] Ke Wang and Zhendong Su. 2020. Blended, Precise Semantic Program Embeddings. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, 121–134. https://doi.org/10.1145/3385412.3385999

[62] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2020. Automated Patch Correctness Assessment: How Far are We?. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 968–980. https://doi.org/10.1145/3324884.3416590

[63] Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang. 2020. Learning Semantic Program Embeddings with Graph Interval Neural Network. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 137 (2020), 27 pages. https://doi.org/10.1145/3428205

[64] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 1–11. https://doi.org/10.1145/3180155.3180233

[65] Ming Wen, Yepang Liu, Rongxin Wu, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. 2019. Exposing Library API Misuses Via Mutation Analysis. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 866–877. https://doi.org/10.1109/ICSE.2019.00093

[66] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating bugs from software changes. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 262–273.

[67] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 87–98.

[68] Yinxing Xue, Mingliang Ma, Yun Lin, Yulei Sui, Jiaming Ye, and Tianyong Peng. 2020. Cross-Contract Static Analysis for Detecting Practical Reentrancy Vulnerabilities in Smart Contracts. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1029–1040.

[69] Zhaoxu Zhang, Hengcheng Zhu, Ming Wen, Yida Tao, Yepang Liu, and Yingfei Xiong. 2020. How Do Python Framework APIs Evolve? An Exploratory Study. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 81–92. https://doi.org/10.1109/SANER48275.2020.9054800

[70] Gang Zhao and Jeff Huang. 2018. DeepSim: deep learning code functional similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.