How Are We Detecting Inconsistent Method Names? An Empirical Study from Code Review Perspective

KISUB KIM and XIN ZHOU, Singapore Management University, Singapore, Singapore
DONGSUN KIM, Korea University, Seoul, Republic of Korea
JULIA LAWALL, Centre Inria de Paris, Inria, Paris, France
KUI LIU, Huawei Software Engineering Application Technology Lab, Hangzhou, China
TEGAWENDÉ F. BISSYANDÉ , University of Luxembourg, Luxembourg, Luxembourg
JACQUES KLEIN, University of Luxembourg, Luxembourg, Luxembourg
JAEKWON LEE, Kangwon National University, Chuncheon, Republic of Korea
DAVID LO, Singapore Management University, Singapore, Singapore

Proper naming of methods can make program code easier to understand, and thus enhance software maintainability. Yet, developers may use inconsistent names due to poor communication or a lack of familiarity with conventions within the software development lifecycle. To address this issue, much research effort has been invested into building automatic tools that can check for method name inconsistency and recommend consistent names. However, existing datasets generally do not provide precise details about why a method name was deemed improper and required to be changed. Such information can give useful hints on how to improve the recommendation of adequate method names. Accordingly, we construct a sample method-naming benchmark, ReName4J, by matching name changes with code reviews. We then present an empirical study on how state-of-the-art techniques perform in detecting or recommending consistent and inconsistent method names based on ReName4J. The main purpose of the study is to reveal a different perspective based on reviewed names rather than proposing a complete benchmark. We find that the existing techniques underperform on our review-driven benchmark, both in inconsistent checking and the recommendation. We further identify potential biases in the evaluation of existing techniques, which future research should consider thoroughly.

This research/project is supported by the National Research Foundation, Singapore, under its Industry Alignment Fund – Pre-positioning (IAF-PP) Funding Initiative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore. This work was also supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP)–ICT Creative Consilience Program grant funded by the Korea government (MSIT) (IITP-2025-RS-2020-II201819). This work was also supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2021R1I1A3048013). This work was also partially supported by the National Natural Science Foundation of China (No. 62172214).

Authors' Contact Information: Kisub Kim, Singapore Management University, Singapore, Singapore; e-mail: falconlk00@gmail.com; Xin Zhou, Singapore Management University, Singapore, Singapore; e-mail: xinzhou.2020@phdcs.smu.edu.sg; Dongsun Kim (corresponding author), Korea University, Seoul, Republic of Korea; e-mail: darkrsw@korea.ac.kr; Julia Lawall, Centre Inria de Paris, Inria, Paris, France; e-mail: julia.lawall@inria.fr; Kui Liu, Huawei Software Engineering Application Technology Lab, Hangzhou, China; e-mail: brucekuiliu@gmail.com; Tegawendé F. Bissyandé, University of Luxembourg, Luxembourg, Luxembourg; e-mail: tegawende.bissyande@uni.lu; Jacques Klein, University of Luxembourg, Luxembourg, Luxembourg; e-mail: jacques.klein@uni.lu; Jaekwon Lee, Kangwon National University, ChunCheon, Republic of Korea; e-mail: jaekwon.lee@kangwon.ac.kr; David Lo, Singapore Management University, Singapore, Singapore; e-mail: davidlo@smu.edu.sg.

ACM 1557-7392/2025/7-ART178
https://doi.org/10.1145/3711901

CCS Concepts: • **Software and its engineering → Software version control**; **Empirical software validation**;

Additional Key Words and Phrases: Method Name Recommendation, Consistency Checking, Code Review, Empirical Study

ACM Reference format:
Kisub Kim, Xin Zhou, Dongsun Kim, Julia Lawall, Kui Liu, Tegawendé F. Bissyandé, Jacques Klein, Jaekwon Lee, and David Lo. 2025. How Are We Detecting Inconsistent Method Names? An Empirical Study from Code Review Perspective. *ACM Trans. Softw. Eng. Methodol.* 34, 6, Article 178 (July 2025), 27 pages.
https://doi.org/10.1145/3711901

"*There are only two hard things in Computer Science: Cache invalidation and naming things.*"—Philip Lewis Karlton [28]

1 Introduction

A method is the smallest unit of program behavior in a software project [24, 52]. Accordingly, choosing method names that are consistent with the conventions of the software project and the behavior of the method body is crucial for software maintenance [8, 18], comprehension [23, 30, 46, 49], and reuse [7, 12, 13, 36]. Nevertheless, software developers are not always successful in choosing appropriate method names. Figure 1 shows a few examples from open source projects. In Figure 1(a), the method name `register` suggests that the method stores some data in an internal container while the method actually creates a new object and returns it. Thus, the corresponding code review suggests `generateToken` instead. In Figure 1(b), the method name `transitToRestoreActive` suggests that the method changes a program state selectively, while actually, the method is executed all the time. The corresponding review recommends `enforceRestoreActive`.

To alleviate the problem of inconsistent method names, many researchers have proposed techniques [2, 33, 35, 38, 42] to automate two tasks: (1) **method name consistency checking (MCC)** and (2) **method name recommendation (MNR)**. A challenge, however, is how to evaluate such approaches in a meaningful way. For example, the dataset of Liu et al. [38] comes from revisions that only change method names. While creating a revision that only changes method names does suggest that the method names were unsuitable, such a revision does not provide clear evidence of whether the change reflects the conventions of a community or the opinion of a single developer. Datasets furthermore have been balanced, to meet the needs of learning methods and report results on the recognizability of specific classes. But in practice, method names tend to stabilize over time, and thus most methods in a well-maintained software project already have consistent names, so a technique scanning for naming problems has to be able to cope with imbalanced inputs. Finally, techniques often rely on similarity thresholds that have been optimized for specific datasets. These issues indicate that there may be a gap between the research results and reality.

Code review data can provide insight to address the above method naming challenges. Alsuhaibani et al. [6] found that method names are discussed during code review and that developers change method names in response to comments from reviewers. Such discussions raise an opportunity to sample method-renaming pairs, in which method name changes are supported by discussions among maintainers of the affected code base, and to use this database to assess MCC and MNR techniques. For our benchmark, ReName4J, we first collect the reviews from pull requests in GitHub [19]. We leverage naming-related terms (e.g., "naming," "name," "inconsistent," and "descriptive") to identify relevant reviews, and collect the corresponding patches, including the actual code files as well as the metadata (e.g., file hashes, paths, pull request status, and commit information). To

```
public String register(DirectoryBrowserSupport dbs, StaplerRequest req) {
    ...
    String value = authenticationName + ":" + date.getTime() + ":" + completeUrl;
    try {
        return encode(value);
    } catch (Exception ex) {
        LOGGER.log(Level.WARNING, "Failed to encode " + value, ex);
    }
    return null;
}
```

register should be renamed to be consistent with the current approach (was more meaningful when it was stateful). Proposal: **generateToken**.

(a) A method from Jenkins project and its corresponding review comment.[1]

```
public void transitToRestoreActive() {
    if (state != ChangelogReaderState.ACTIVE_RESTORING) {
      ...
      pauseChangelogsFromRestoreConsumer(standbyRestoringChangelogs());
    }
    state = ChangelogReaderState.ACTIVE_RESTORING;
}
```

Renamed this method to make it clear we aren't necessarily "transitioning", we actually call it all the time now any time we want to "be in restoreActive".

(b) A method from Kafka project and its corresponding review comment.[2]

Fig. 1. Code review examples related to method naming.

extract the method names, we parse all the collected patches and files, then we map the extracted names to the reviews. We include 400 sample pairs of method names (i.e., buggy and fixed), method bodies, and the corresponding reviews.

Based on ReName4J, this article investigates existing MCC and MNR techniques and asks the following **research questions (RQs)**:

- —RQ1: Can the existing MCC/MNR tools retrieve the appropriate names for the methods that are considered in code review scenarios?
- —RQ2: Do the existing MCC/MNR tools achieve an acceptable detection rate for consistent method names as well as for those of inconsistent method names?
- —RQ3: What happens when we test the MCC/MNR tools on an imbalanced test dataset reflecting the properties of real-world software projects instead of a balanced test dataset that is artificially selected?

Answering these questions helps fill the gaps between the research and practice by providing three insights. First, the effectiveness of existing techniques for measuring MCC/MNR relies on the validity of the potential ground truth. Second, it is important to consider how current tools fare in both identifying inconsistencies and verifying consistency. Third, the impact of the balance or imbalance of test sets on performance should also be considered. We believe that these RQs could capture the attention of researchers who are interested in software refactoring specifically, MNR and consistency checking. Moreover, researchers and practitioners can better comprehend various perspectives and evaluate the relevant tools.

To address these questions, we present an empirical study of how state-of-the-art techniques detect/recommend the names for test oracles from our benchmark. The study involves three recent techniques: Spot [38], Cognac [52], and GTNM [37], for MCC/MNR tasks. The study's results showcase that existing techniques are less effective at recommending suitable method names for the newly developed benchmark, which is based on code review data, compared to those used in previous studies. This indicates code review comments may provide insights for better name

[1]https://github.com/jenkinsci/jenkins/pull/4239/files/41919ee7829223062d5d5ca4d592fd8056e55017#r330954855
[2]https://github.com/apache/kafka/pull/8319/files/c7ac051e495a98b6c72a340c24f4b9bb1f25dcdd#r395348873

recommendations, such as exploiting context information available in other methods of the same classes or modules. For detecting inconsistent names, the target techniques perform similarly to random classification, often misclassifying consistent names as inconsistent due to the nuanced nature of human-reviewed names, which demands deeper code context understanding. Differences in method name complexity and length between our benchmark and existing datasets may also challenge models trained on simpler examples. Insights from ReName4J suggest that incorporating more method body information could improve classification performance. Finally, the imbalanced dataset, which reflects real-world conditions, may expose issues such as overly strict threshold values for classifying inconsistent names.

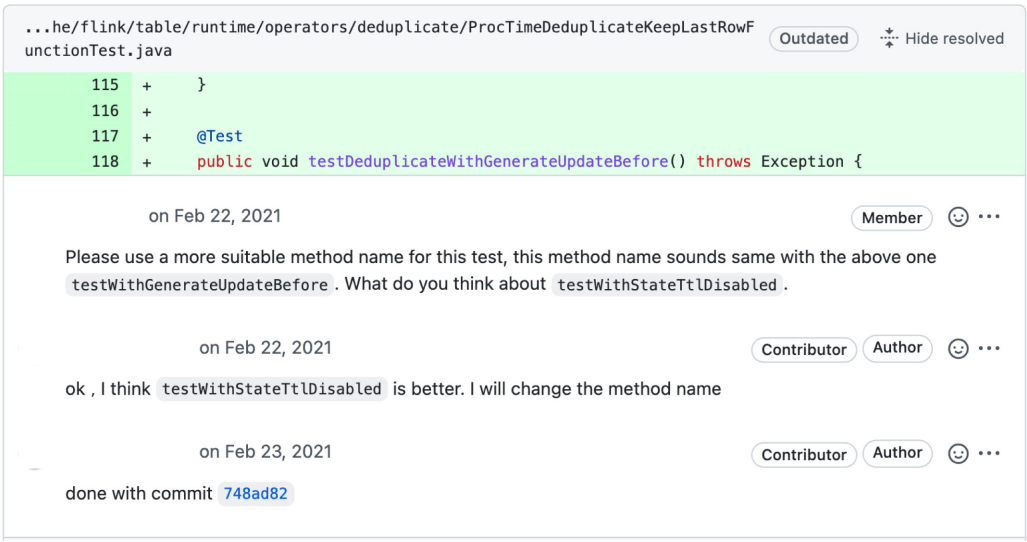
Our **key findings** are as follows:

- —The method names in existing datasets lack validation because the consistency of methods has not been thoroughly verified through reviews. In contrast, our benchmark includes method names that have been rigorously checked and accepted by project code reviewers. By incorporating these verified method names, our benchmark offers greater practical value and precision for evaluation.
- —The existing test dataset [38] includes many methods with simple names, typically consisting of a single verb such as "get", "take," or "listen." These intuitive method names are easy to recommend because method bodies often contain many such tokens. However, code review comments reveal that reviewers prefer more descriptive and comprehensive names. In fact, the average number of characters in method names from the existing dataset is 13.18, while in our benchmark, it is 24.90, approximately 88.92% longer.
- —The existing techniques show a performance decline across all tasks when applied to our human-reviewed method names, compared to the results reported in the original studies.
- —The evaluation design of existing recommendation-based techniques needs careful reconsideration, as it may be biased toward specific cases.

In summary, this article contributes the following:

- —The first in-depth investigation of MCC/MNR tools, reflecting a code-review perspective.
- —A small, but reliable benchmark, consisting of sample pairs of method names (i.e., buggy and fixed), method bodies, and review comments. This is also the first manually labeled naming dataset from a code review perspective. It may serve as a baseline dataset for future MCC/MNR research as the method names have been confirmed by project reviewers.
- —Delving into advanced evaluation methods for MCC/MNR techniques due to variations in performance measurement protocols within prior literature. Moreover, strategies to construct an improved benchmark that minimizes manual effort should be examined.
- —We conducted in-depth interviews with seven developers currently employed at a tech company, along with an external postdoctoral researcher and two Ph.D. students. The purpose of these interviews was to understand the following: (1) whether historically incorrect method names are indeed incorrect and assess the stability of the changes made to these names, (2) why existing techniques fall short in ensuring consistency and accuracy in method name checking and recommendation, and (3) the limitations of these existing techniques when applied in practical, real-world scenarios.

The experimental results show that the review-driven benchmark could provide insights for better naming by exploiting more context available in the reviews from the experts. The provided reviews in ReName4J also can give intuitions on how to improve the classification performance by utilizing more information in the body of the methods.



(a) A review from Apache/Flink project.[3]

(b) A review from Elastic/Elasticsearch project.[4]

Fig. 2. Real-world code reviews on method names.

2 Motivating Example

The naming of methods is a critical issue for software engineering and development as method names have a high impact on the comprehension of the source code [40, 49]. Consequently, developers actively discuss naming issues frequently during code reviews in commercial and open source software projects [6]. We first investigate the method names and review pairs in open source projects to determine whether the explicitly reviewed method names could be a better ground truth for MCC and MNR tasks.

Figure 2 illustrates two examples of naming-related reviews from Apache/Shardingsphere and Elastic/Elasticsearch, respectively. Figure 2(a) shows a case in which a code reviewer recommends

[3]https://github.com/apache/flink/pull/14863
[4]https://github.com/elastic/elasticsearch/pull/82639

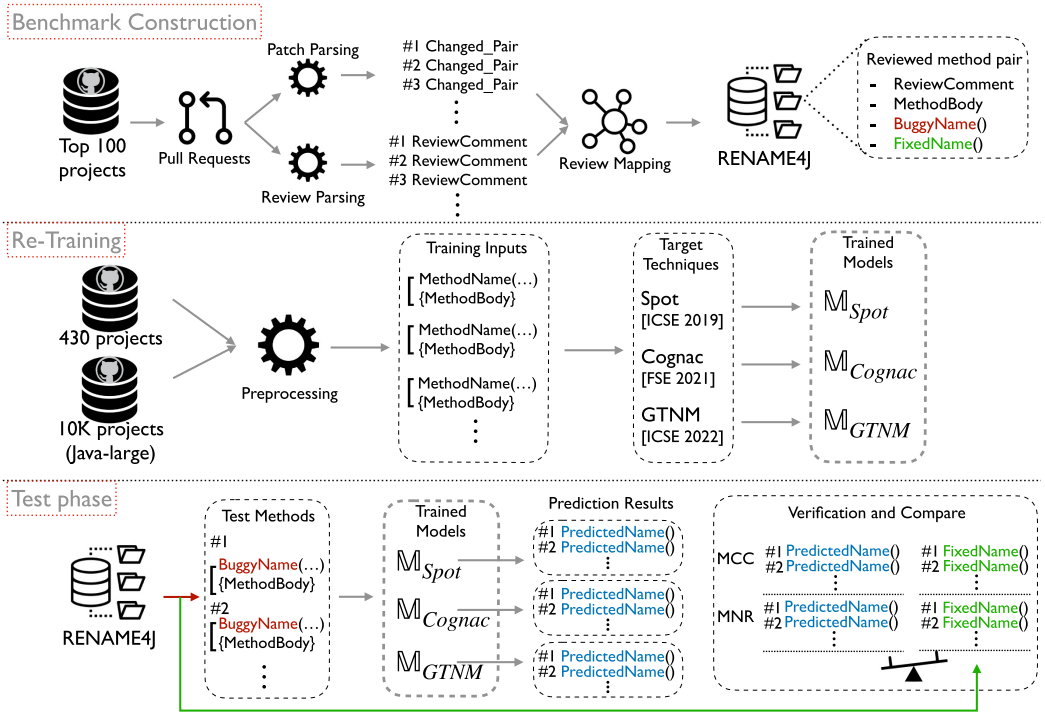


Fig. 3. Overall procedure for our study.

using a more suitable method name for a test while giving the justification that the current name sounds the same as that of another method. Figure 2(b) illustrates a renaming argument based on the specific naming conventions of the project. Specifically, the original name was `getRealmRef()`, which follows a common Java-Bean naming convention, but the reviewer suggested instead following the Builder naming convention, illustrated as `name()`, which implies that the name should be `realmRef()`.

Investigating various examples, we discovered that the code reviewers tend to comment on the method names in many ways such that the developers can directly apply a proposed renaming or be inspired to appropriately change the names. These changes are explicitly discussed, changed, and accepted. We consider that these cases are more solid than other renaming cases that are only changed and accepted without any discussion in software development.

3 Methodology

This section presents an overview of our empirical study. As shown in Figure 3, our study comprises three steps: (1) constructing a novel method naming benchmark (ReName4J), (2) retraining existing method-name recommendation techniques, and (3) testing the performance of the techniques with respect to the benchmark. Our benchmark, ReName4J, collects method renaming instances from code review data such that every instance is backed by discussions of real developers and reviewers. Based on the benchmark and obtained models, we conduct two tasks: (1) inconsistency classification of each pair of method name and body (MCC) and (2) MNR for a given method body (MNR).

3.1 Target Techniques

To answer the RQs described in Section 1, we select three target techniques: Spot [38], Cognac [52], and GTNM [37] as (1) they have been published recently at top software engineering venues and (2) their implementations are completely reproducible.

Spot: Spot [38] is designed with the intuition *"Methods implementing similar behavior in their body code are likely to be consistently named with similar names, and vice versa."* Notably, Spot adopts unsupervised learning [21] and lazy learning [1] to embed method names and bodies into numerical vectors. Given a method name and body, it compares two sets of names. According to the embedding vectors, the first set consists of other method names similar to the name of the given method and the second set consists of the names of methods whose bodies are identified as similar to the body of the given method. When the intersection of these two sets is larger than a threshold, the method name is recognized as consistent; otherwise, Spot classifies it as inconsistent (i.e., the MCC task). If the name turns out to be inconsistent, Spot recommends potential consistent names for the method based on the comparison results on method bodies (i.e., the MNR task).

Cognac: Cognac [52] is a program-structure-independent approach aiming to avoid the out-of-vocabulary problem. Unlike prior techniques, Cognac not only leverages the *local context* extracted from target methods themselves, including code tokens and the associated contexts, but it also considers the *global context*, which is the contextual information from other methods such as call relations (i.e., caller/callee information) with the target method. Technically, it follows the *seq2seq* paradigm, using the extracted token sequences to infer method names.

GTNM: GTNM [37] is a transformer-based neural model considering the information of the whole project (global context). It extracts context from three different levels, given the target method and the project, including (1) local context, (2) project-specific context, and (3) documentation context. Technically, it employs a transformer-based seq2seq framework [50] with two encoders and a single decoder to generate the method name. Particularly, we include GTNM in our study because its design closely resembles that of Cognac while it misses the evaluation of MCC. Since Cognac has already showcased that an MNR technique can effectively serve as an MCC task, it is crucial to undertake a comprehensive performance assessment in this context. This addition could shed light on the potential advantages associated with choosing the right approach for tasks that necessitate a specific tool.

Two other approaches, MNire [42] and DeepName [35] have been recently proposed and shown to outperform previous approaches. We had to discard MNire [42] due to the unavailability of the source code [52]. Fortunately, Cognac [52] has been compared against the numbers in the MNire paper with the same dataset, and Cognac outperforms MNire. We failed to execute DeepName [35] based on its replication package, as the instructions are not complete, and the author who is in charge of the package did not reply to our questions. Nevertheless, DeepName is similar to Cognac, in that both use caller/callee information, and both were released at almost the same time. Both Cognac and DeepName were evaluated against MNire, and they show approximately 10% and 7% improvement, respectively, in the F1-score with the same dataset for the MNR task.

3.2 ReName4J: Naming Review Benchmark

Our study first builds a benchmark based on code review data, which can be used to assess method-name recommendation techniques. To the best of our knowledge, there is no concrete benchmark for method (re)naming that incorporates code-review data. Existing techniques are evaluated on a dataset collected from source code revision history without an associated explanation for the naming changes. Distinguishing between true renaming tasks and coincidental code changes proves challenging, as developers often disperse a unified modification across multiple commits [53, 54]. In

contrast, ReName4J leverages code review data to collect actual method renaming tasks, supported by justifications of why the name should be revised. Notably, the dataset could be applied to both traditional and learning-based approaches.

ReName4J has four components for each method renaming task: (1) **ReviewComment**: the review comment explaining why the name should be revised, (2) **MethodBody**: the relevant method body (i.e., source code), (3) **BuggyName**: the method name before refactoring (buggy_method_name()), and (4) **FixedName**: the method name after refactoring (fixed_method_name()).

To collect the review and method name pair dataset, we leverage GitHub's REST API.[5] We first crawl the top 100 Java projects sorted by the stars, considering the project's popularity [20]. Among the top 100 Java projects, we discard the projects where the discussions are primarily written in languages other than English. This leaves 66 projects as our targets. We then crawled a total of 144, 759 pull requests for these projects. As we only need to consider pull requests with naming-related reviews, we extract those pull requests that contain keywords such as "naming," "name," "inconsistent," and "descriptive" in their review comments. Based on the process, we collected 11, 301 pull requests. A pull request tends to include multiple commits and a commit can be associated with multiple files. Our dataset thus covers 73, 985 commits and a total of 593,240 associated files.

We preprocess the collected code review data as follows. We break down the patch files to discover the methods where only the name changes while excluding those whose names have been changed with their body as changes in the body can naturally affect the name. This step resulted in 1,212 changed method names. Connecting reviews with their corresponding methods is difficult due to the structure of GitHub's database, as it does not allow for direct linking between a review comment and a specific method. Typically, review comments pertain to an entire pull request. Another obstacle to mapping reviews to methods is that reviewers often comment on external code links that cannot be accessed through GitHub's API but are visible on the web. Finally, we tackle the issue by establishing a connection between the patch files containing commit modifications and the metadata of pull requests. These metadata encompass crucial information such as commit hashes, parent hashes, and review comments. A total of three authors manually investigated all pairs of reviews and method names, including only those with a clear connection between the review and the method. For all the PRs, the three authors voted on each, reaching agreement on 945 out of 1,212 cases, resulting in an agreement rate of 77.97%. Additionally, all the 267 discrepancies cases were resolved through discussion and consensus to ensure accuracy. The examples in Section 2 also demonstrate how this connection can be manually labeled by matching the context of the reviews and methods.

The manual checking and filtering criteria are as follows:

- We include methods where the pull request was accepted by the project's code reviewer. The methods we extracted have no specific restrictions, and the code reviewer could be either marked as "Contributor" or "Member," as long as they were assigned to review the PR.
- We include methods with names that are directly suggested by the reviewer (e.g., testGetConnectionSession() → assertGetConnectionSession() | *Based on the review: Please rename "testGetConnectionSession" to "assertGetConnectionSession"*).[6]
- We include methods with names that are undoubtedly inspired by the review (e.g., completeRestoration() → completeRestorationIfPossible() | Based on the review: *nit: can we rename this to "maybeCompleteTaskTypeTransition" or "completeTaskTypeConversionIfNecessary," etc?*

[5] https://docs.github.com/en/rest

[6] https://github.com/apache/shardingsphere/pull/18618

Right now, it kind of sounds like we just randomly attempt to convert it to a new task type out of nowhere).[7]

—We include methods with names that are recommended based on the project's naming convention (e.g., realmRef() → getRealmRef() | Based on the review: *Nit: existing getter methods in this class following the Builder type naming convention, e.g., name() instead of getName(), while this method uses the JavaBean naming convention (getXxx). I am OK with either of them but would prefer consistency within a single class).*[8]

—We include methods with names that are reviewed and commented on for consistency (e.g., register() → getToken() | Based on the review: *"register" should be renamed to be consistent with current approach (was more meaningful when it was stateful)*).[9]

—We ignore simple typos, as they can be noise in the benchmark, as well as example methods.

The selected commits satisfy at least one of the defined criteria. As a result, we identify, as the ground truth, 400 method renaming tasks supported by the corresponding reviews. The selection of 400 tasks was not predetermined but rather occurred naturally during our process. We found that the directly suggested method names constituted 7.75%, undoubtedly inspired ones were 17.25%, those recommended based on naming conventions were 48%, and the ones for consistency were 37.50%. These categories sometimes overlapped because a single MNR could be influenced by multiple factors. For instance, a method name might be directly suggested while the reviewer mentions the convention or consistency. In addition, we explicitly eliminate all the class files associated with the corresponding methods from the training data to ensure that there is no data leakage [43]. The absolute number is smaller than the datasets used in the prior studies [38, 52, 37], but our benchmark collects only obvious renaming tasks confirmed by code reviewers and developers. This allows a more precise evaluation of method renaming techniques. Note that we utilized a standard javalang[10] library for any parsing-related tasks.

To understand the characteristic differences between the existing test oracles [38] and those from our benchmark, we further conduct a brief preliminary investigation. Based on our observation, the average number of characters for the existing test method names and those of our benchmark are 13.18 and 24.90 (approx. 88.92% longer), respectively, while those of split sub-tokens are 2.52 and 4.53 for each method name. This indicates a clear difference whereas our benchmark contains method names that are more descriptive, and it suggests that the code reviewers, in practice, prefer to describe as many parts of the methods as possible.

3.3 Experiment Design

To conduct a fair comparison, we employ the following procedure [52] for all three target techniques. Overall, we evaluate the three techniques described in Section 3.1 with two tasks: MNR addressing RQ1 and MCC addressing RQ2, after retraining them by using the datasets. Then, we feed the method name/body pairs in ReName4J to the (retrained) target techniques and measure the performance with respect to several metrics. In addition, our study evaluates the techniques on more realistic data (i.e., a highly unbalanced dataset including the name/body pairs in ReName4J) to see their effectiveness in a different scenario.

3.3.1 Retraining. To avoid bias from the dataset, we re-train the recommendation-based target techniques (denoted as $\mathbb{M}$, e.g., $\mathbb{M}_{Cognac}$ and $\mathbb{M}_{GTNM}$) with a single identical dataset. We leverage the *Java-large* dataset, released by Alon et al. [4], as the training and validation dataset for two,

[7] https://github.com/apache/kafka/pull/8988
[8] https://github.com/elastic/elasticsearch/pull/82639
[9] https://github.com/jenkinsci/jenkins/pull/4239
[10] https://github.com/c2nes/javalang

Table 1. Statistics of the Training and Validation Dataset

	Train	Validation	Total
Projects	9,772	450	10,222
Files	1,756,282	51,631	1,807,913
Methods (for $\mathbb{M}_{Cognac}$ and $\mathbb{M}_{GTNM}$)	13,992,028	466,800	14,458,828
Methods (for $\mathbb{M}_{Spot}$)	2,116,413	-	2,116,413

Cognac [52] and GTNM [37] of our target techniques. This dataset is the most popular [4, 5, 35, 37, 42, 52] and well-maintained for the MNR task. It consists of 10,222 top-ranked Java projects from GitHub, including 14,458,828 methods and 1,807,913 unique files. We randomly shuffled and split all the projects into 9,772 training and 450 validation projects. In addition, to avoid any data leakage, we carefully checked and eliminated all methods included in ReName4J (Section 3.2) from the training dataset so that there is no intersection between training and testing datasets. Table 1 shows the training and validation data statistics.

Exceptionally, we use a different training dataset for the checking-based approach (denoted as $\mathbb{M}_{Spot}$), due to its known scalability issues, which we confirmed with its authors [38]. The *Java-large* dataset, which is almost seven times bigger than the dataset originally used for training Spot, is impractical for Spot's training according to its authors. Consequently, we utilize the original training dataset used by Spot's authors to reproduce the model. This original dataset for Spot includes 2,116,413 methods extracted from 430 Java projects.

3.3.2 Method Name Recommendation (MNR) Task. The first task to measure the effectiveness of the target techniques is recommending method names for a given method implementation (i.e., method body). In this task, each target technique, $\mathbb{M}$, takes a method body (**MethodBody** in ReName4J) and produces a set of candidate method names that best describe the body, as described in the following equation:

$$\mathbb{M} : B^* \rightarrow N^*, \tag{1}$$

where B is an alphabet allowed for a method body and N is an alphabet allowed for a method name, respectively. B^* and N^* are sets of sequences over B and N, respectively. After feeding a method body, $b \in B^*$, available in ReName4J, we produce a recommended name by $\mathbb{M}(b) = n \in N^*$. We then compare n with our ground truth name, **FixedName**, in the benchmark. The results of this task for each technique are discussed in Section 4.1 as RQ1.

While Cognac [52] and GTNM [37] were designed for MNR, Spot [38] was originally built for consistency checking and later added recommendation functionality. To ensure a fair comparison, we replaced the target method names of our test methods with meaningless tokens when using Spot, which only predicts method names when they are classified as inconsistent.

3.3.3 MCC Task. Instead of recommending a name directly, it might be useful if a technique clarifies whether a given method name is inappropriate to describe the method body. This consistency check can help developers or code reviewers scan a project and figure out the overall naming practice. Equation (2) represents the MCC task, $\mathbb{D}_{\mathbb{M}}$, for a given method body $b \in B^*$, method name $n \in B^*$, and specific name recommendation technique $\mathbb{M}$.

$$\mathbb{D}_{\mathbb{M}} : B^* \times N^* \rightarrow \{C, IC\}, \tag{2}$$

where C and IC denotes *consistent* and *inconsistent* name. Basically, we expect that the verdict should be C when feeding a method body (**MethodBody**) and corresponding **FixedName** in

ReName4J. Otherwise, it should be *IC* if **BuggyName** is given. The results of this task are discussed in Section 4.2 as RQ2.

State-of-the-art techniques [35, 42] claimed that it is possible to conduct MCC based on MNR by employing a specific similarity checking metric, described in the following subsection with other metrics. Several prior MNR approaches [35, 37, 52] use this hypothesis to check the consistency of the method names obtained using their recommendation models.

3.3.4 Reflection of the Real World. In the real world, the likelihood of encountering inconsistent method names is significantly lower than those that are consistent. To accurately reflect this situation, we adopt another dataset. As our target methods in the benchmark are small parts of multiple class files, there exist remaining methods from these files (i.e., the majority of Java class files tend to consist of multiple methods). The quantity of these remaining methods is substantially greater (13,137) in comparison to our test oracles (400). These are stable methods that have not been modified, and thus, we consider the name of a method as **StableName** and a body as **StableBody**. Note that these are not explicitly identified as stable during the code review process, but this is a common assumption as checking every single method that is not reviewed is practically infeasible. Also, this is the same process with a prior study [38]. This implies that we can use them to simulate a realistic scenario, and we utilize them for our experimentation in addressing real-world reflection (i.e., RQ3), which aims to reflect reality. To conduct an experiment on such a dataset, we take the same protocol as the MCC task while the verdict should be *C* when feeding a **StableName**. Otherwise, it should be *IC* if a **BuggyName** is given. The results of this task are reported in Section 4.3 as RQ3.

3.4 Performance Metrics

3.4.1 Metrics for MNR. As the objective of the MNR task is to recommend the same or similar names to those that human developers wrote, we compare the ground truth names in the ReName4J with the names generated by the target techniques. Note that the unit of comparison is tokenized words (i.e., sub-tokens) of the names rather than the whole chunk of the names to capture partial matches. We employ *Precision*, *Recall*, *F1-score*, and **exact match accuracy (*EMAcc*)** to evaluate the results of the MNR task, as represented in Equations (4)–(7), respectively.

We use the followings notations for the equations: $o \in N^*$ and $p \in N^*$ are a ground truth name (i.e., test oracle) in ReName4J and a name recommended by a targeted technique based on the method body corresponding to o, respectively. $token()$ is a function defined from N^* to its power set as follows:

$$token : N^* \rightarrow \mathcal{P}(N^*), \tag{3}$$

where $token(name)$ produces a token set after tokenizing *name*. This study applies the camel case tokenization as it is the naming convention of Java.

—Precision_{MNR} (positive predictive value) is the metric that represents an estimation of how many tokens are correctly predicted within all the predicted tokens. It is expressed as follows:

$$\text{Precision}_{MNR}(o, p) = \frac{|token(p) \cap token(o)|}{|token(p)|}. \tag{4}$$

—Recall_{MNR} (sensitivity) is the metric that represents an estimation of how many tokens are correctly predicted within all the oracle tokens. It is expressed as follows:

$$\text{Recall}_{MNR}(o, p) = \frac{|token(p) \cap token(o)|}{|token(o)|}. \tag{5}$$

—F1-score$_{MNR}$ is the harmonic mean of the precision and recall. It weights the two ratios (precision and recall) in a balanced way:

$$\text{F1-score}_{MNR}(o,p) = \frac{2 \times Precision(o,p) \times Recall(o,p)}{Precision(o,p) + Recall(o,p)}. \quad (6)$$

Furthermore, we take EMAcc(o,p), i.e., EMAcc, to assess whether a technique can recommend a name exactly the same as the ground truth name in ReName4J. The function is defined as:

$$\text{EMAcc} : N^* \times N^* \rightarrow \{0,1\}, \quad (7)$$

where EMAcc$(o,p) = 1$ if o and p are identical. Otherwise, the value is 0.

3.4.2 Metrics for MCC. To measure the performance on the MCC task, we use metrics from prior studies [35, 38, 42, 52]: Precision, Recall, F-score, and Accuracy. MCC can have four possible outcomes: IC classified as IC (i.e., true positive = TP), IC classified as C (i.e., false negative = FN), C classified as C (i.e., true negative = TN), and C classified as IC (i.e., false positive = FP). Accordingly, the metrics are defined as follows: $\text{Precision}_{IC} = \frac{|TP|}{|TP|+|FP|}$, $\text{Recall}_{IC} = \frac{|TP|}{|TP|+|FN|}$, $\text{Precision}_{C} = \frac{|TN|}{|TN|+|FN|}$, $\text{Recall}_{C} = \frac{|TN|}{|TN|+|FP|}$, the F1-score is computed as $\frac{2 \times Precision \times Recall}{Precision+Recall}$ for each class. Moreover, the *Accuracy* on the whole dataset is defined as $\frac{|TP|+|TN|}{|TP|+|FP|+|TN|+|FN|}$.

Additionally, the recommendation-based approaches need a threshold value T, which is used for the similarity between the predicted name and buggy or fixed names. It is manually set for the MCC task in prior studies [35, 42, 52], and following them, we set it as 0.85 in this study.

4 Experimental Results

This section reports all the results of our target techniques and answers the RQs.

4.1 RQ1: Can the Target Techniques Recommend the Appropriate Names for a Given Method Body?

Setup: This RQ aims to evaluate the effectiveness of existing tools in suggesting suitable method names based on code review scenarios. The focus is on whether these tools can accurately recommend appropriate method names for methods under consideration in code reviews, i.e., when they are evaluated on ReName4J instead of their original datasets. Following the protocol described in Section 3.3.2, we feed each **MethodBody** in ReName4J to each technique and take the corresponding recommended name as a result. After getting the results, we compute the performance based on the metrics defined in Section 3.4.1.

Results and Analyses: The target techniques show lower performance on ReName4J than in their original studies. Table 2 shows the MNR results as x/y, where x is our result and y (in light grey) is the previously reported result. Note that Spot [38] did not report on the performance of the MNR task with the metrics described in Section 3.4, but rather mainly uses first token matching. Thus, we cannot directly compare it with the original performance of Spot. Still, Spot performs worst among the target techniques.

Thanks to ReName4J, where all renames are backed by code review (i.e., the **ReviewComment** component), we can figure out the reason why the techniques fail to recommend proper method names. Cognac incorrectly recommends *size()* for a method, and the following

Table 2. Results for the MNR

	Spot [38]	Cognac [52]	GTNM [37]
EMAcc	3.50/-	8.57/51.80	14.73/62.01
Precision$_{MNR}$	30.79/-	38.59/71.40	47.39/77.01
Recall$_{MNR}$	22.90/-	25.12/61.90	41.80/74.15
F1-score$_{MNR}$	26.27/-	30.49/66.30	44.42/75.60

The gray values (nn.nn) denote the reported results in their papers while "-" indicates that such results are not available.

review[11] corresponds to the case in which **BuggyName** and **FixedName** are *size()* and *approximateNumEntries()*, respectively:

> **Reviewer A**: It seems like this method doesn't return an exact "size" for the RocksDb implementation so we need to spec it differently if we want to include it. We should also consider the right name and whether we actually want to expose it.
> ...
> **Reviewer B**: "approximate" sounds good. Also to differentiate with number of bytes as *@anotherReviewer* pointed out on the ticket, how about use "approximateNumEntries"?

According to the review comment corresponding to the method name, the reviewers suggested "approximate..." since the function returns an approximated size instead of the exact "size" of a specific container.

ReName4J can help improve a MNR technique as it provides the rationale of a method renaming, written by real developers. For example, the above example suggests that researchers should focus more on semantic and global context information rather than local context. Here is another example:[12] Cognac and GTNM recommends *withPartition()* and *createTopic()*, respectively, for the following **MethodBody**:

```
{
        return new TopicPartition(
         record.topic(), record.partition() == null ?
          RecordMetadata.UNKNOWN_PARTITION : record.partition());
}
```

while the actual **BuggyName** and **FixedName** in ReName4J are *createTopicPartition()* and *extractTopicPartition()*. The corresponding **ReviewComment** is:

> **Reviewer**: This ("createTopicPartition") is not a very good name since "creating a partition" usually means calling createPartitions. How about "extractTopicPartition"?

This example motivates that we need to leverage global context such as other source code in the same project since there are similar method implementation names (e.g., *createPartitions()* and *createTopics()*) in other source files.

[11] https://github.com/apache/kafka/pull/1486#discussion_r66461876
[12] https://github.com/apache/kafka/pull/11689#discussion_r788921625

Table 3. Results for MCC

		Spot [38]	Cognac [52]	GTNM [37]
IC	**Precision**$_{IC}$	50.52/56.80	48.27/68.60	43.56/-
	Recall$_{IC}$	97.00/84.50	80.00/97.60	66.00/-
	F1-score$_{IC}$	64.44/67.90	60.21/80.60	52.49/-
C	**Precision**$_{C}$	62.50/72.00	41.61/96.00	29.90/-
	Recall$_{C}$	5.00/38.20	14.25/55.60	14.50/-
	F1-score$_{C}$	9.26/49.90	21.23/70.40	19.53/-
Accuracy		51.00/60.90	47.12/76.60	40.25/-

The gray values (nn.nn) denote the reported results under their own test oracle, while "-" indicates that such results are not available.

On ReName4J*, the techniques are less successful at recommending appropriate method names (MNR) than in previous evaluations. Code review comments may provide insights for better name recommendations, such as exploiting context information available in other methods of the same classes or modules.*

4.2 RQ2: Can the Target Technique Detect Inconsistent Method Names?

Setup: This RQ aims to assess the ability of the target techniques to identify inconsistent method names. The study evaluates the effectiveness of each technique in detecting inconsistencies in method names by comparing **BuggyName** and **FixedName** pairs with their corresponding **MethodBody**. The goal is to determine if the techniques can classify method names as inconsistent or consistent based on the provided data. This is particularly related to the MCC task described in Section 3.3.3. Note that we follow the hypothesis described in a prior study [38] that the methods that have not been commented on in pull requests to be renamed have consistent names. We feed a pair of a **BuggyName** and the corresponding **MethodBody** in ReName4J to each technique; we expect that these are classified as inconsistent names. In addition, we feed a pair of a **FixedName** and the corresponding **MethodBody**; we expect that they are classified as consistent names.

After constructing a confusion matrix based on the four possible outcomes, we compute values for the four metrics described in Section 3.4.2. The computed values are shown in Table 3. The results follow the form x/y used in Table 2, where x is the result on ReName4J, and y is the previously reported result. As there are no previous MCC results for GTNM, the table does not include the previous values in this case. Nevertheless, it is evident that GTNM can serve as a valuable tool for consistency checking. This is underlined by the fact that a similar methodology has been employed in the development of Cognac, which also aims at name recommendations.

Results and Analyses: As for the MNR task, most of the values drop from the original studies except for the case of Recall$_{IC}$ for Spot. The overall degradation is less than the MNR task, but some cases are still significant. For example, the Recall$_{C}$ and F1-score$_{C}$ values of Spot and Cognac drop from 38.20% and 55.60% to 5.00% and 14.25%, respectively. This indicates that most of the **FixedName** and **MethodBody** pairs are incorrectly classified as *inconsistent* ones. Although the drop of Precision$_{IC}$ and Precision$_{C}$ is not as significant as the drop in Recall$_{C}$ and F1-score$_{C}$, the values are just around $\approx$ 50, which indicates that the performance is not better than random classification, as the MCC task is fundamentally a binary classification.

The following example shows the reasons for some failures. Cognac and GTNM differently classify the following pair of **BuggyName** and **MethodBody** as *consistent* and *inconsistent*, respectively.

(1) **BuggyName**: *append()*
(2) **MethodBody**:

```
{
    return append(lastOffset < 0 ? baseOffset : lastOffset + 1,
     timestamp, key, value);
}
```

whereas the **FixedName** is *appendWithOffset()* corresponding to the **MethodBody**. Both Cognac and GTNM classify the pair of this **FixedName** and the **MethodBody** as *inconsistent*. The **ReviewComment**[13] is as follows:

> **Reviewer**: Hmm, can we think of another name for the methods that increment the offset automatically?
> ...
> **Committer**: Or maybe this can be the default and the other can be named "appendWithOffset"?
> ...
> **Reviewer**: That works for me.

The techniques fail to correctly classify both the **BuggyName** and the **FixedName** since they recommend *append()* and *addrecord()* as method names. This indicates that the techniques should better utilize the information available in **MethodBody** as it mentions "Offset" tokens several times. In addition, the reviewers talk about the method of incrementing the offset automatically.

The target techniques achieve a performance similar to random classification as they overclassify consistent names as inconsistent names. The models may struggle with over-classification due to the nuanced nature of human-reviewed names, which often require a deeper understanding of the code context. A potential significant factor here could also be the difference in complexity and length of method names between our benchmark and the existing datasets, which may challenge the models trained on simpler examples. The reviews in ReName4J *can give intuitions on how to improve the classification performance, such as utilizing more information in the method body.*

4.3 RQ3: What Happens When We Test the Effect of Balanced vs. Imbalanced Data?

Setup: The objective of this RQ is to assess the efficacy of each target technique when evaluating them on an imbalanced dataset (a huge number of consistent pairs and a small number of inconsistent ones), which reflects the MCC task in real practice, as outlined in Section 3.3.4. Similar to RQ2, we feed the pairs of method names and bodies. For inconsistent pairs, we assess all the **BuggyName** and **MethodBody** inconsistent pairs in ReName4J using each technique. For consistent pairs, we assess all the **StableName** and **StableBody** pairs from the dataset described in Section 3.3.4; we expect that they will all be classified as *consistent*. This is the "imbalanced dataset" (400 vs. 13,137) in this study.

Results and Analyses: On the imbalanced dataset, the target techniques mostly fail to classify consistent names. The results are shown in Table 4 and denoted as x/y, where x is the result of the imbalanced dataset described above, while y is the result for the balanced dataset (same with Table 3). $Precision_{IC}$ values are significantly decreased, with a decline of 90.40%, 94.39%, and 94.90% for Spot, Cognac, and GTNM, respectively. As only a few inconsistent pairs are classified as consistent, $Precision_{C}$ values rise, with an increase of 55.82%, 131.36%, and 212.41%, respectively.

[13] https://github.com/apache/kafka/pull/2282#discussion_r93328221

Table 4. Results for MCC with the Real-World Reflecting Dataset

		Spot [38]	Cognac [52]	GTNM [37]
IC	**Precision**$_{IC}$	4.85/50.52	2.71/48.27	2.22/43.56
	Recall$_{IC}$	97.00/97.00	80.00/80.00	66.00/66.00
	F1-score$_{IC}$	9.24/64.44	5.25/60.21	4.30/52.49
C	**Precision**$_{C}$	97.39/62.50	96.27/41.61	93.41/29.90
	Recall$_{C}$	5.56/5.00	15.25/14.25	14.24/14.50
	F1-score$_{C}$	10.52/9.26	26.32/21.23	24.72/19.53
	Accuracy	9.89/51.00	17.11/47.12	15.73/40.25

The gray values (nn.nn) denote the previous results from Section 4.2.

The results in Table 4 indicate that the techniques are highly biased to classify method names and body pairs as inconsistent ones. The following pair[14] shows an example of incorrect classification. Both Cognac and GTNM classify the following pair of **StableName** and **StableBody** as *inconsistent* with different recommendations.

(1) **StableName**: *testWithStateTtlDisabled()*
(2) **StableBody**:

```
{
      ProcTimeDeduplicateKeepLastRowFunction func =
      createFunctionWithoutStateTtl(true, true);
      ...
      testHarness.processElement(insertRecord("book", 1L, 12));
      ...
      // Keep LastRow in deduplicate may send UPDATE_BEFORE
      ...
}
```

Both Cognac and GTNM fail to correctly recommend **StableName**. They suggest *testKeepLastRowFunction()* and *testInsertWithUpdate()*, respectively. This result indicates that the techniques need to further focus on the invoked methods as the key was the method, *createFunctionWithoutStateTtl()*.

Overall, the techniques need to relax the threshold values when detecting inconsistent pairs. For the following **StableBody**,[15] Cognac and GTNM recommend *getAlterResource()* and *createSimpleResourceStatement()*, respectively, which are quite similar to the actual **StableName**, *createAlterResourceStatement()*:

```
{
    return new AlterResourceStatement(Collections.singleton
     (new DataSourceSegment(resourceName,
     "jdbc:mysql://127.0.0.1:3306/ds_0",
     null, null, null, "root", "", new Properties())));
}
```

[14] https://github.com/wangpeibin713/flink/blob/748ad82745d9d493922150fe007136e125a50209/flink-table/flink-table-runtime-blink/src/test/java/org/apache/flink/table/runtime/operators/deduplicate/ProcTimeDeduplicateKeepLastRowFunctionTest.java
[15] https://github.com/open-beagle/shardingsphere/blob/5be6a2c81c647cdffb1f7f17db9c69204be14b4e/shardingsphere-proxy/shardingsphere-proxy-backend/src/test/java/org/apache/shardingsphere/proxy/backend/text/distsql/rdl/resource/AlterResourceBackendHandlerTest.java#L132

Table 5. Background Details of Real-World Developers

Participant	Affiliation Type	Degree	Current Position	Years of Dev.	Java Exp.	Daily Java Usage	Time Spent (mins)
1	Academia	Ph.D.	Research Fellow	12	4.5	5	50
2	Academia	Bachelor	PhD Student	2	2	5	85
3	Academia	Master	PhD Student	3	1.5	4	120
4	Industry	Ph.D.	Senior Researcher	6.5	5	5	40
5	Industry	Master	Senior Researcher	4	2.5	5	44
6	Industry	Ph.D.	Senior Researcher	5.5	5	5	38
7	Industry	Ph.D.	Principal Researcher	7.5	5	5	60
8	Industry	Ph.D.	Senior Researcher	4	3	5	110
9	Industry	Master	Principal Engineer	10	7	5	30
10	Industry	Bachelor	Junior Engineer	7	5	5	78

As the recommended names and **StableName** share many common tokens, it is appropriate to classify the above case as consistent, but the techniques fail to correctly classify it because their threshold values are too tight.

> *The imbalanced dataset, reflecting the real practice, may reveal shortcomings such as too strong threshold values for inconsistent name classification.*

5 User Study

Setup: This user study aims to gather in-depth insights from software developers regarding their experiences and practices. To ensure the results are statistically significant, we calculated the required number of samples to achieve a 95% confidence level, with a margin of error of ±5%. This calculation determined that a minimum of 197 samples was necessary. Consequently, we conducted interviews with 200 samples to meet and slightly exceed this threshold.

We asked participants to provide information about their backgrounds as developers. The following list was the first questions: (1) Years as a professional software developer; (2) Years of experience with Java code; (3) How often they write Java programs in their current job [Likert scale, 1–5]; and (4) Academic background [BSc, MSc, and PhD]. Table 5 shows the developers' background details.

Consequently, with 200 instances across 10 developers, we randomly distributed 20 samples to each participant. For each method body provided, we presented four distinct questions to assess whether the method name was suitable and a sub-question follows if the answer is negative. The method names presented were categorized into different origins: Buggy, Fixed, Cognac, and GTNM. To ensure unbiased responses, the origins of these method names were concealed from the participants.

Given a **{MethodBody}**,
- Q1.1: Is the following name, **{BuggyName}**, consistent with its body?
- Q1.2: If No, why?
  - Completely inconsistent.
  - Partially inconsistent.
  - Coding Convention/Style issue.
  - Mistake/Typo.
  - Depends on the broader context.
- Q2.1: Is the following name, **{FixedName}**, consistent with its body?
- Q2.2: If No, why?
  - ...

Results and Analyses: On the background checking questions, the participants in the study had an average of 6 years of development experience. Regarding Java development, the average year was 4.5 and most of them scored 5 on the Likert scale except one with 4. Among them, five held Ph.D.

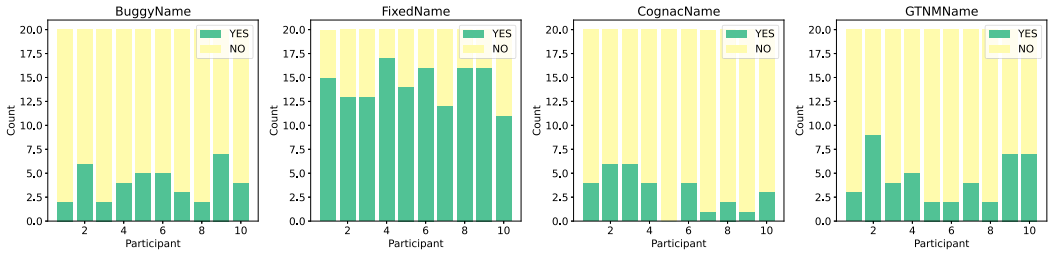


Fig. 4. Statistics of developer answers on method names for given method bodies.

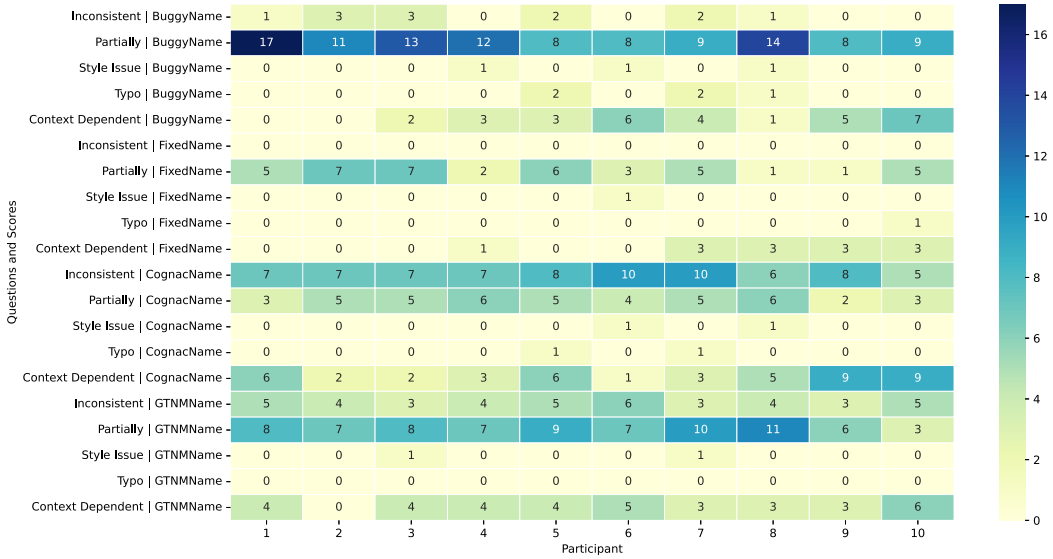


Fig. 5. Statistics of developer answers on the rationale of negative votes.

degree, three had Master's degree, and two had Bachelor's degree. Additionally, each interview lasted approximately 65.5 minutes, allowing us to gather all the responses.

Analysis of YES/NO Responses: Figure 4 presents a clear overview of the voting patterns for various method names. The results indicate a strong preference among participants for the **FixedName**, with the majority casting a "YES" vote, suggesting that developers widely recognize the **FixedName** as accurate and appropriate for the given method bodies. In contrast, the **BuggyName**, as well as the **SuggestedNames** by the target approaches (Cognac and GTNM), received predominantly negative votes, as evidenced by the significantly higher "NO" responses. This pattern reveals a general skepticism or disagreement among developers regarding the correctness or suitability of these names.

Analysis of Specific Question Scores: Figure 5 presents the detailed rationale scores for negative responses. When developers voted "NO" for **BuggyName**, the majority cited it as *Partially inconsistent*, indicating that while some aspects of the name may align with the method's functionality, there are notable inconsistencies. As expected, **FixedName** received generally low counts for negative responses, reflecting its overall acceptance. However, among those who did vote negatively, *Partially inconsistent* was also the most commonly selected rationale, suggesting that while generally well-received, some minor issues or ambiguities might still exist.

In contrast, **SuggestedNames** by Cognac were frequently labeled as *Completely inconsistent* by participants, highlighting significant misalignment with the method's functionality. Additionally, these names were often considered to *Depend on the broader context*, implying that developers found these names to be ambiguous or requiring additional context to be understood. For GTNM, the trend was somewhat more favorable, with more participants selecting *Partially inconsistent* rather than *Completely inconsistent*. This presents that while GTNM's suggestions were not fully aligned with expectations, they were perceived as closer to being appropriate compared to those from Cognac.

The results of this interview suggest that automated tools like Cognac and GTNM may require further refinement to better align with developer expectations. On the other hand, the strong preference for **FixedName** implies that manual corrections based on the code review process are currently more effective in achieving correct names that are widely accepted by the developer community.

Analysis of StableNames*:* Additionally, we randomly sampled 30 pairs of stable method names that were incorrectly detected as inconsistent (i.e., <**StableName**, **SuggestedName**>). To assess how developers perceive the suggested names for these methods, we conducted an experiment involving three participants from the industry. Each participant was provided with all 30 cases for evaluation. The 30 **SuggestedNames** were generated by our target approaches, with 15 produced by Cognac and 15 by GTNM, respectively.

Among the three developers, two marked the **StableNames** as consistent for all 30 sample cases. Only two cases from one developer were marked as partially inconsistent. During a follow-up phone discussion with this developer, they confirmed that the two exceptions were indeed only partially inconsistent, and they did not have strong objections to the **StableNames**, as they were very similar to the **SuggestedNames**.

> *Developers still prefer to see the method names recommended by reviewing the code as such names are recommended by the expert, knowing the details of project environments. This suggests that the recommended names in the code review process are often preferred and indicates that our benchmark can be effectively utilized to reflect real-world requirements. Overall, there are still research gaps that need to be addressed to improve existing MCC/MNR approaches.*

6 Discussion

We mainly discuss the experimental settings for evaluating the target techniques as well as the differences between the datasets. The following discussion could also deliver actionable insights for future research directions.

6.1 Threshold *T* used in the MCC task

When performing the MCC task, most of the techniques compute the similarity between the ground truth name and the predicted name. This similarity calculation bears a threshold value T, which is heuristically decided in the previous studies [35, 42, 52]. This threshold value then determines the performance. However, the choice of T depends on the technique. For example, MNire [42] takes various values for T while Cognac [52] uses 0.85 as a fixed value. Moreover, the threshold is chosen to maximize the performance of each technique.

Although the impact of the threshold T is investigated in a study [42], it was only to maximize the performance of each technique. This indicates that it needs to be tuned depending on the approach and the choice of T potentially causes bias in different scenarios. Furthermore, one of the prior studies [52] stated, "we never know a method name is consistent or not before the detection in

Table 6. The Results of MCC with the Fixed Value for Threshold T as 1

		Cognac [52]	GTNM [37]
IC	**Precision**$_{IC}$	47.98/48.27	43.49/43.56
	Recall$_{IC}$	80.00/80.00	66.75/66.00
	F1-score$_{IC}$	59.98/60.21	52.66/52.49
C	**Precision**$_{C}$	39.85/41.61	28.49/29.90
	Recall$_{C}$	13.25/14.25	13.25/14.50
	F1-score$_{C}$	19.89/21.23	18.09/19.53
Accuracy		46.62/47.12	40.00/40.25

The gray values (nn.nn) denote the previous results from Section 4.2 as we aim to discover the effects of the threshold value T.

practice" regarding the decision on the threshold. This may imply that their test oracles cannot guarantee the quality for evaluating MCC/MNR tasks. We investigate the performance of the recommendation-based techniques with T as 1 as we hypothesize **FixedName** in ReName4J may be the ground truth.

The results in Table 6 indicate that the target techniques are slightly influenced by the threshold value. Cognac's recommendations for the *IC* class do not change at all. However, there are more false positives for the *C* class, reducing the precision. More importantly, both subjects experience a decrease in performance for the *C* class. Intriguingly, there is a slight performance increase in GTNM's recommendation on Recall$_{IC}$ class (i.e., 66.00% to 66.75%). The increase in true positives has resulted in this phenomenon, which is unexpected as it is commonly believed that the higher the threshold value, the more difficulty the techniques will encounter. Inspired by this, we motivate the following discussion.

6.2 Measurement of True Positives for the Inconsistent Class

During the MCC task of recommendation-based techniques, i.e., Cognac and GTNM, we discovered that the initial evaluation design may be biased due to the following reasons. A binary classification is not suitable for these approaches, as they initially recommend and then check the inconsistency based on the predictions. In this way, a recommended method name cannot ensure if a method name is inconsistent or consistent. Previous studies, such as Cognac [52], define a true positive in their evaluation as a case where the similarity between the predicted tokens and the oracle tokens from the inconsistent name is less than the threshold value T (i.e., $Sim(o, p) < T$). This means that the lower the model's recommendation performance, the more true positives it has for the inconsistent class. However, this approach may not accurately reflect the goal of improving method name consistency, as the predicted names should ultimately match the ground truth names. To address this, we revised the evaluation setting and only consider a predicted name as a true positive if its similarity to the oracle tokens from the consistent name is equal to 1, as this represents a perfect match with the ground truth.

The findings presented in Table 7 indicate a marked decrease in performance compared to the initial results. Cognac decreased by 86%, 92%, and 89% for $Precision_{IC}$, $Recall_{IC}$, and $F1\text{-}score_{IC}$, respectively. GTNM performed slightly better, with decreases of 64%, 76%, and 70% on each metric. The results indicate that recommendation-based approaches demonstrate significant limitations on

Table 7. The Results of MCC with Corrected Measurement of the True Positives for IC Class

		Cognac [52]	GTNM [37]
IC	**Precision**$_{IC}$	6.79/48.27	15.56/43.56
	Recall$_{IC}$	6.25/80.00	15.75/66.00
	F1-score$_{IC}$	6.51/60.21	15.65/52.49

The gray values (nn.nn) denote the previous results from Section 4.2 as we aim to check the phenomenon after the correction.

inconsistency checking. Overall, there is a need for further improvement in such approaches as the results in Table 7 do not seem to be promising. Additionally, such an evaluation protocol requires careful examination.

6.3 Size of the Test Dataset

On the Method Name Recommendation. Most recommendation-based studies [35, 37, 42, 52] employ the test dataset from Java-large [5] for the MNR task. This dataset includes approximately 636K test methods from 61K files which is a very large number compared to our datasets. Although we understand that the bigger the dataset is, the more solid and robust the evaluation that can be conducted, such a dataset has been solely cloned from open source projects without any filtering criteria. This suggests that there is no evidence that such method names are ever changed, let alone whether the changes are reviewed by others.

On the Method Name Consistency Checking. Different from the dataset commonly used for the MNR task, most prior techniques [34, 35, 38, 42, 52] leverage the dataset from a checking-based study [38], which has been constructed using specific filtering criteria. The filtering criteria include methods whose names have been changed in a commit without any alteration to the body code. Additional conditions, such as disregarding method changes with typographical errors and those that do not involve changes to the first sub-token, are also applied as the techniques are sensitive to such cases. These filtering criteria ensure that the dataset only contains methods whose names have been explicitly changed by developers. The dataset comprises 2,805 methods from 430 projects for the test phase. However, it cannot guarantee that the method name changes are associated with the inconsistency of the method name and its body since there are cases where the developer only changed the method body first and then fixed the name after [53, 54].

On Our Benchmark. Our main goal is to conduct an empirical investigation of the existing tools with different perspectives (i.e., code review). While we continuously collect and craft more method and review pairs, we acknowledge that the current number of test oracles in this study is relatively small compared to the existing ones. As the ultimate goal of the MCC/MNR tasks is to be as precise and practical as possible, it is vital that the test oracles closely resemble the ground truth. To achieve this, we propose utilizing a code review process involving the code reviewers as a means to enhance the quality of the data. We demonstrate the significant differences in the lengths of names between the method names from our benchmark and those of existing datasets. This may imply that the benchmark can be leveraged to evaluate MCC/MNR techniques with different scenarios as we described in earlier sections.

In summary, the set of test oracles used for the MNR task is the largest, but its practical usefulness is questionable. The dataset used in the MCC task cannot guarantee whether the method name changes are associated with the inconsistency checking task or coincident code changes as they

are collected without considering the rationale behind them. Meanwhile, our benchmark dataset is composed of methods that have been reviewed and explicitly pointed out by code reviewers of open source projects. As we continually extend the benchmark, we expect it will allow for a more comprehensive evaluation of MCC/MNR techniques.

6.4 Key Distinctions

On the Incorporation of Code Review Data. ReName4J leverages code review data from pull requests on platforms like GitHub to collect actual method renaming tasks supported by justifications provided in review comments. This inclusion of code review context could provide valuable insights into why method names are changed, enhancing the dataset's relevance and practicality.

On the Ground Truth Validation. Unlike existing datasets, ReName4J rigorously validates method names through project code reviewers. This validation process ensures that the method names in the dataset have been thoroughly checked and accepted by experts, enhancing the dataset's reliability and accuracy.

On the Comprehensive Method Naming. The dataset includes method names that are more descriptive and comprehensive, reflecting the preferences of code reviewers for meaningful and informative method names. This contrasts with existing datasets that may contain simpler or less informative method names, highlighting the importance of realistic and context-rich data for MNR tasks.

On the Contextual Information. ReName4J provides not only method names but also associated review comments, method bodies, and metadata, offering a rich source of contextual information for evaluating MNR techniques. This additional context can help algorithms better understand the rationale behind method name changes and improve recommendation accuracy.

In existing datasets, method names may be simplistic and lack validation from code review processes. For instance, a dataset with method names like "get," "take," or "listen" may not capture the complexity and nuances of real-world method naming practices. In contrast, ReName4J includes method names like "retrieveCustomerInformation" or "processPaymentTransaction" which are more representative of actual naming conventions observed in code reviews.

6.5 Possible Solution and Future Direction

Besides utilizing the review datasets, several potential approaches can be explored to improve the detection of inconsistent method names. First, leveraging enhanced contextual analysis by utilizing surrounding code information, such as class names and variable names, can provide a deeper understanding of appropriate method names. Second, advanced machine learning techniques, including fine-tuning models on diverse datasets and employing ensemble methods, can significantly enhance detection accuracy. Third, curating better-quality training data, generating synthetic examples, and establishing a user feedback loop for continuous learning and improvement may be crucial for detecting inconsistent names as well. Finally, collaborating with the developer community could establish more practical applications. These solutions not only address current challenges but also pave the way for future advancements in method name consistency detection. We believe that our study can offer diverse perspectives, particularly in the realm of code-related human interactions. We also plan to extend our dataset by incorporating method names from additional programming languages to broaden the scope and applicability of our findings.

6.6 Threats to Validity

Threats to external validity may lie on the targeted three techniques [37, 38, 52] that are open source; thus, the results may not be representative of other techniques or closed-source techniques.

Furthermore, as we consider approaches targeting Java, our conclusions are only valid for this language. Exploring training and testing models on individual project data may have further value, and it highlights the complexity of naming conventions across different environments. However, our current study aims to provide a broad understanding of naming conventions by focusing on a diverse set of top Java projects. A threat may lie in the dataset as well. We utilize a common dataset for training all the recommendation-based techniques, and such a dataset can possibly be biased, which could lead the models in the wrong direction. To mitigate this threat, we incorporate a large-scale dataset [4] that consists of high-quality and well-maintained open source projects. The identifier names in these large-scale projects are known to be mostly consistent [4, 5]. Additionally, such approaches focus on learning common features from the majority of the methods. As a result, during the training phase, a natural mitigation of this concern could be attained.

Threats to internal validity may include human-reviewed method names in our benchmark. To address this threat, the authors cross-checked the method names and bodies. We found that there exist very long method names with huge bodies which could be ambiguous to decide whether the names are consistent or not. All the authors who cross-checked the method names agreed on the final version of the benchmark. In addition, although it is not guaranteed that all positive items have been identified, appropriate rationales for the name changes could support the quality of the naming dataset as well as recommendation tasks. Additionally, the method renaming cases that undergo discussion may indicate ambiguity in choosing the "best" name for a given method. This suggests that our results might represent a lower boundary of acceptable names. Many other valid names might not have been discussed or selected as the final choice, implying that the actual performance could be better than indicated by our benchmark. This potential lower bound should be considered when interpreting the results.

Threats to construct validity relate to the evaluation metrics we employed. We took the same metrics as target techniques [37, 38, 52], which are Accuracy, Precision, Recall, F1-score, over sub-tokens, and EMAcc. Although there may exist other potential metrics to measure the performance of MCC/MNR techniques, evaluating such techniques with the prior metrics over sub-tokens is the approach taken by all the related literature.

7 Related Work

Empirical Investigation on Identifier Naming. Based on the assumption, poor names make programs harder to understand and maintain, many researchers [11, 14, 36,49] have conducted empirical investigations on naming source code identifiers. The majority of these studies explore the impact of names on source code readability [18, 45], program comprehension [23, 44, 51], and software maintainability [12]. In addition, several studies [22, 29] investigate the unintended inconsistency of the identifiers. Recently, researchers [6, 27] have begun to focus on naming methods, as these are the smallest units of an aggregated functionality. Such studies investigate existing MNR techniques or survey professional developers to understand the impact of naming choices.

MCC/MNR Techniques, on the Applied Models. Approaches were built based on textual similarity and co-occurrence. Høst and Østvold [26] introduced a debugging approach for method names by inferring naming rules based on the return type, control flow, and method parameters. Lucia et al. [16] proposed an information retrieval based approach using LSI [17] to compute the textual similarity to improve program comprehension.

Recently, researchers have proposed learning-based techniques. Allamanis et al. [2, 3] employ deep learning techniques (i.e., basic and copy convolutional attention models) to recommend method names. Liu et al. [38] take the same embedding strategy with Paragraph Vector [31] and convolutional neural networks [32] considering additional code nodes at the **abstract syntax tree (AST)** level to capture code semantic information [41]. Code2seq [4] and Code2vec [5] are

well-known code embedding techniques that are evaluated with the MNR task. Code2seq [4] follows the standard encoder-decoder architecture [15] with a bi-directional LSTM [25] while Code2vec [5] leverages a newly designed path-attention network. Wang et al. [52] further added the pointer-generator network [47] and introduced Cognac aiming to avoid the out-of-vocabulary problem. With a similar concept, Nguyen et al. [42] and Li et al. [35] proposed MNire and DeepName, respectively, considering different contexts. These models leverage the RNN-based Seq2seq [10] model with attention mechanism [9, 39] aiming to capture contextual sentences. GTNM proposed by Liu et al. [37] is a transformer-based model that leverages the self-attention mechanism capturing rich semantic dependencies.

MCC/MNR Techniques, on the Leveraged Contexts. To check and recommend the method names, various related contexts are considered. Most approaches exploit the features from the local context, such as the method implementation (e.g., tokens of the return type, parameters, control flow graph, data flow graph, AST, and each identifier in the body) [2–5, 38, 48]. Recently, researchers started to explore a wider range for the context, such as the enclosing class name [42] and method invocation relations (i.e., caller and callee) [35, 52]. Liu et al. [37] further extended the context to project-specific information (i.e., in-file methods and cross-file contextual methods) and documentation of the method (i.e., Javadoc) [37]. A few studies leverage high-level artifacts such as software requirement documents [16] and semantic profiles of the method implementations [26].

8 Conclusion

We investigated existing MCC and recommendation approaches with a novel benchmark that contains clear rationales for changing method names. Investigating the recent empirical studies on developer activities for naming identifiers, we discovered that developers could finalize the method names with code review comments. Inspired by this phenomenon, we construct a benchmark for method name changes by collecting and mapping with the related review comments, which can be a further layer of checking the consistency. We hold the perspective that our benchmark could serve as a reference for MCC and recommendation techniques. This is because the enclosed test oracles are derived from recommendations or insights provided by code reviewers within the projects. To validate existing studies on the benchmark, we explored whether they show consistent performance compared to the previous dataset that does not rely on clear rationales. Although there was an exceptional case due to the scalability issue with an existing technique, Spot, we carefully selected our target techniques and designed the experimental setting for a fair comparison. The results of the existing approaches demonstrate a consistent decrease in our test oracles for most of the metrics they defined. Overall, this indicates that there still exists room for improvement, especially when they encounter different scenarios and evaluation protocols. Moreover, we revealed that there exist potentially biased features discussed in Sections 6.1 and 6.2 in the evaluation of existing techniques that research should consider to fill the gap between research and practice. We hope that these actionable insights could be applied to future research directions.

Data Availability

We publicly release a replication package that includes all the code and datasets to reproduce the experiments of our study at https://figshare.com/s/8cdb4e3208e01991e45c.

References

[1] David W. Aha. 2013. *Lazy Learning*. Springer Science & Business Media.
[2] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 38–49.

[3] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *Proceedings of the International Conference on Machine Learning*. PMLR, 2091–2100.
[4] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating sequences from structured representations of code. In *Proceedings of the International Conference on Learning Representations*. Retrieved from https://openreview.net/forum?id=H1gKYo09tX
[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. In *Proceedings of the ACM on Programming Languages 3 (POPL '19)*, 1–29.
[6] Reem Alsuhaibani, Christian Newman, Michael Decker, Michael Collard, and Jonathan Maletic. 2021. _On the naming of methods: A survey of professional developers. In *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 587–599. DOI: https://doi.org/10.1109/ICSE43902.2021.00061
[7] 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.
[8] 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.
[9] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. *Neural machine translation by jointly learning to align and translate*. arXiv:1409.0473. Retrieved from https://arxiv.org/abs/1409.0473
[10] Denny Britz, Anna Goldie, Minh-Thang Luong, and Quoc Le. 2017. Massive exploration of neural machine translation architectures. arXiv:1703.03906. Retrieved from https://arxiv.org/abs/1703.03906
[11] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2009. Relating identifier naming flaws and code quality: An empirical study. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering*. IEEE, 31–35.
[12] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2010. Exploring the influence of identifier names on code quality: An empirical study. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 156–165.
[13] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2011. Improving the tokenisation of identifier names. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer, 130–154.
[14] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2011. Mining Java class naming conventions. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 93–102.
[15] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv:1406.1078. Retrieved from https://arxiv.org/abs/1406.1078
[16] Andrea De Lucia, Massimiliano Di Penta, and Rocco Oliveto. 2010. Improving source code lexicon via traceability and information retrieval. *IEEE Transactions on Software Engineering* 37, 2 (2010), 205–227.
[17] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. 1990. Indexing by latent semantic analysis. *Journal of the American Society for Information Science* 41, 6 (1990), 391–407.
[18] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. 2018. The effect of poor source code lexicon and readability on developers' cognitive load. In *Proceedings of the 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 286–28610.
[19] GitHub. 2022. Retrieved from https://github.com/
[20] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 933–944.
[21] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. 2009. Unsupervised learning. In *The Elements of Statistical Learning*. Springer, 485–585.
[22] Yoshiki Higo and Shinji Kusumoto. 2012. *How often do unintended inconsistencies happen? Deriving modification patterns and detecting overlooked code fragments*. In *Proceedings of the 2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 222–231.
[23] Johannes Hofmeister, Janet Siegmund, and Daniel V. Holt. 2017. Shorter identifier names take longer to comprehend. In *Proceedings of the 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 217–227.
[24] Einar W. Høst and Bjarte M. Østvold. 2009. Debugging method names. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer, 294–317.
[25] Zhiheng Huang, Wei Xu, and Kai Yu. 2015. Bidirectional LSTM-CRF models for sequence tagging. arXiv:1508.01991. Retrieved from https://arxiv.org/abs/1508.01991
[26] Einar W. Høst and Bjarte M. Østvold. 2009. Debugging method names. In *Proceedings of the Conference on Object-Oriented Programming (ECOOP '09)*. Sophia Drossopoulou (Ed.), Springer, Berlin, 294–317.
[27] Lin Jiang, Hui Liu, and He Jiang. 2019. Machine learning based recommendation of method names: How far are we. In *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 602–614.

[28] Karlton. 2022. Retrieved from https://www.nndb.com/people/400/000031307/
[29] Suntae Kim and Dongsun Kim. 2016. Automatic identifier inconsistency detection using code dictionary. *Empirical Software Engineering* 21, 2 (2016), 565–604.
[30] 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 '06)*. IEEE, 3–12.
[31] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *Proceedings of the International Conference on Machine Learning*. PMLR, 1188–1196.
[32] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. 1999. Object recognition with gradient-based learning. In *Shape, Contour and Grouping in Computer Vision*. David A. Forsyth, Joseph L. Mundy, Vito Gesú, and Roberto Cipolla (Eds.), Springer, 319–345.
[33] Kejun Li, Taiming Wang, and Hui Liu. 2021. NameChecker: Detecting inconsistency between method names and method bodies. In *Proceedings of the 2021 28th Asia-Pacific Software Engineering Conference (APSEC)*, 22–31. DOI: https://doi.org/10.1109/APSEC53868.2021.00010
[34] Kejun Li, Taiming Wang, and Hui Liu. 2021. NameChecker: Detecting inconsistency between method names and method bodies. In *Proceedings of the 2021 28th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 22–31.
[35] Yi Li, Shaohua Wang, and Tien Nguyen. 2021. A context-based automated approach for method name consistency checking and suggestion. In *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 574–586.
[36] Ben Liblit, Andrew Begel, and Eve Sweetser. 2006. Cognitive perspectives on the role of naming in computer programs. In *Psychology of Programming Interest Group*. Citeseer, 11.
[37] F. Liu, G. Li, Z. Fu, S. Lu, Y. Hao, and Z. Jin. 2022. Learning to recommend method names with global context. In *Proceedings of the 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1294–1306. DOI: https://doi.org/10.1145/3510003.3510154
[38] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. 2019. Learning to spot and refactor inconsistent method names. In *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1–12.
[39] Minh-Thang Luong, Ilya Sutskever, Quoc V. Le, Oriol Vinyals, and Wojciech Zaremba. 2014. Addressing the rare word problem in neural machine translation. arXiv:1410.8206. Retrieved from https://arxiv.org/abs/1410.8206
[40] Jonathan I. Maletic and Andrian Marcus. 2001. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE '01)*. IEEE, 103–112.
[41] 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 30th AAAI Conference on Artificial Intelligence*, Vol. 30, 1287–1293.
[42] 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.
[43] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why should I trust you?" Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1135–1144.
[44] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vasquez, Denys Poshyvanyk, and Rocco Oliveto. 2019. Automatically assessing code understandability. *IEEE Transactions on Software Engineering* 47, 3 (2019), 595–613.
[45] 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), e1958.
[46] Andrea Schankin, Annika Berger, Daniel V. Holt, Johannes C. Hofmeister, Till Riedel, and Michael Beigl. 2018. Descriptive compound identifier names improve source code comprehension. In *Proceedings of the 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 31–3109.
[47] Abigail See, Peter J. Liu, and Christopher D. Manning. 2017. Get to the point: Summarization with pointer-generator networks. arXiv:1704.04368. Retrieved from https://arxiv.org/abs/1704.04368
[48] Takayuki Suzuki, Kazunori Sakamoto, Fuyuki Ishikawa, and Shinichi Honiden. 2014. An approach for evaluating and suggesting method names using n-Gram models. In *Proceedings of the 22nd International Conference on Program Comprehension*, 271–274.
[49] Armstrong A. Takang, Penny A. Grubb, and Robert D. Macredie. 1996. The effects of comments and identifier names on program comprehensibility: An experimental investigation. *Journal of Programming Languages* 4, 3 (1996), 143–167.
[50] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* 30 (2017), 6000–6010.
[51] Yaza Wainakh, Moiz Rauf, and Michael Pradel. 2021. Idbench: Evaluating semantic representations of identifier names in source code. In *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 562–573.

[52] 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 Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 741–753.
[53] Fengcai Wen, Csaba Nagy, Michele Lanza, and Gabriele Bavota. 2020. An empirical study of quick remedy commits. In *Proceedings of the 28th International Conference on Program Comprehension*, 60–71.
[54] Fengcai Wen, Csaba Nagy, Michele Lanza, and Gabriele Bavota. 2022. Quick remedy commits and their impact on mining software repositories. *Empirical Software Engineering* 27 (2022), 1–31.

Received 21 August 2023; revised 30 October 2024; accepted 3 December 2024