Enhancing Identifier Naming Through Multi-Mask Fine-tuning of Language Models of Code

Sanidhya Vijayvargiya*, Mootez Saad‡, Tushar Sharma‡
*BITS Pilani**, *Dalhousie University*‡
Hyderabad, India*, Halifax, Canada†‡
f20202056@hyderabad.bits-pilani.ac.in*, mootez@dal.ca†, tushar@dal.ca‡

***Abstract*—Code readability strongly influences code comprehension and, to some degree, code quality. Unreadable code makes software maintenance more challenging and is prone to more bugs. To improve the readability, using good identifier names is crucial. Existing studies on automatic identifier renaming have not considered aspects such as the code context. Additionally, prior research has done little to address the typical challenges inherent in the identifier renaming task. In this paper, we propose a new approach for renaming identifiers in source code by fine-tuning a transformer model. Through the use of perplexity as an evaluation metric, our results demonstrate a significant decrease in the perplexity values for the fine-tuned approach compared to the baseline, reducing them from 363 to 36. To further validate our method, we conduct a developers' survey to gauge the suitability of the generated identifiers, comparing original identifiers with identifiers generated with our approach as well as two state-of-the-art large language models, GPT-4 Turbo and Gemini Pro. Our approach generates better identifier names than the original names and exhibits competitive performance with state-of-the-art commercial large language models. The proposed method carries significant implications for software developers, tool vendors, and researchers. Software developers may use our proposed approach to generate better variable names, increasing the clarity and readability of the software. Researchers in the field may use and build upon the proposed approach for variable renaming.**

I. Introduction

High code readability significantly improves code comprehension, reducing effort and time to complete software development tasks [1]. It also reduces the risk of bugs due to improved understanding of the code. Code readability depends significantly upon the quality of identifier names [2]. According to Jiang *et al.* [3], identifiers account for up to 70% of a typical software project in terms of number of characters. Well-crafted identifiers clearly communicate their purpose, which is necessary for cognitively demanding tasks such as software development, documentation, and code review [4].

Despite the importance of good identifiers, many programmers choose bad identifiers such as using single characters or abbreviations as reported by Beniamini *et al.* [5]. They analyzed five open-source Java systems totaling 1.4 million lines of code to assess identifier naming practices. They found that 36% of all Java identifiers used abbreviations omitting vowels and consonants. Additionally, 10% of Java identifiers were single characters. Analyses of projects in other languages revealed similar trends. In Python projects, 28% of identifiers used abbreviations, and 6% of identifiers were single characters [5]. Therefore, suggesting appropriate identifiers to developers are essential to keep a software system highly readable and maintainable.

Though there have been some efforts, especially towards exploring the characteristics that make identifiers suitable [6], [7], proposing a better alternative than the original name is challenging. The primary challenge in choosing identifiers lies in *capturing the relevant context*, a critical factor in the naming process. A suitable identifier (or, variable name) in a specific context can be entirely out of place if used in another context. This characteristic has constrained potential solutions to the identifier renaming problem.

Another limitation in current solutions is the absence of a well-established and large dataset of *good* variable names. Lack of such datasets limit machine learning-based solutions for suggesting a new identifier. However, creating such a dataset is challenging due to the innate problem of what defines a suitable or good variable name [8]. Even for humans, this problem can be subjective due to personal preferences and experience. However, there is a consensus that a suitable variable name must convey the function of the variable while adhering to the naming conventions of the programming language in question. These conventions outline the rules that dictate how a character string can be classified as a variable, along with the practices unique to a given language, such as camel-case in Java. While certain studies, such as those by Wainakh *et al.* [9], have concentrated on assessing the semantic relatedness between two identifier names and their corresponding embeddings, they fall short in determining the *suitability* of a variable name within a specific context.

Deep learning models, especially Transformers [10], can help capture these nuances in context through their learned representations of source code [11]. However, Transformer-based architectures introduce challenges related to tokenization. Transformers operate on tokens, which are the smallest meaningful units of text. An identifier name can consist of one or more subtokens, which are formed by splitting the identifier based on a specific tokenization technique. For example, the identifier `getLength` can be treated as one subtoken or split into two subtokens, such as `get` and `Length`, depending on the tokenizer used. A limitation of current research using Transformers for identifier naming [12] is that when generating a new identifier name, the model is constrained to generate the same number of subtokens as the original identifier (*i.e.,* the reference identifier used in training), even if the original name was suboptimal. To illustrate this, consider an array of objects of type *Book* with an iterator variable named *i*. Previous works would generate a new variable called *iter*, which has the same number of subtokens as *i*. However, generating a more descriptive name like `bookIndex` would require two subtokens instead of one. Given that identifier renaming is primarily applied to low-quality code, this assumption regarding the number of tokens can harm performance. Low-quality code typically tends to have shorter variable names, even single-character names, resulting in fewer subtokens in each variable name.

In this paper, we aim to address all of the problems mentioned with the variable renaming task. We start by curating a dataset of high-quality variable names using a set of heuristics based on the current literature. By selecting high-quality repositories and by applying heuristics to remove poor-quality identifiers lead to better identifier renaming where abbreviations and single-letter names are replaced with semantically significant names. Leveraging this dataset, we propose a novel fine-tuning approach that adapts a language model of code to the identifier renaming task using a masked language modeling technique. In addition to the fine-tuning approach, we propose a different inference method that generates identifier names with varying lengths by considering different numbers of subtokens and selects the most suitable one based on performance metrics. The study makes the following contributions:

- We propose an approach for identifier renaming by fine-tuning a language model using an identifier-aware fine-tuning method. This approach encompasses several important considerations, such as data curation and addressing lengthy sequences of code.
- The study proposes a mechanism to select the optimal number of subtokens to suggest a new variable name, overcoming the challenge of predicating the number of subtokens correctly for a given identifier.
- The study also offers a well-curated dataset comprising of $236,745$ high-quality identifiers, along with their code context. Such a dataset will facilitate further exploration and extension of the research in this domain.

Replication package: Code and data used for training and evaluation of our model can be found online [13]. Furthermore, we have deployed our fine-tuned model online[1]. Rather than grappling with local installation and execution, potential users (*e.g.,* software developers, or researchers) can simply query the API endpoint with input code to obtain model outputs. This facilitates seamless usage of our approach in a wide array of studies, powering future research directions, and ensures easy and transparent replication of the study.

II. Background

This section provides a brief overview about transformers, masked language models and common metrics used with them.

[1]https://huggingface.co/spaces/anon903/ReIdentify

A. Transformers in Software Engineering Applications

Research into applying natural language processing (NLP) techniques to code [14] has led to models trained to generate source code. Large language models for code attempt to capture the syntactic and semantic properties of code are pre-trained on a huge amount of data on the task of code completion, but fine-tuning is crucial to ensuring the model can understand the downstream task [15], [16]. The contribution of transformers to the domain of software engineering is valuable and they can help provide a significant improvement over the state-of-the-art [17]. By pre-training on software engineering domain-specific data, problems like software sentiment analysis and software defect prediction, among many others, have achieved improved results [18][19]. Better pre-training objectives which are more adept at capturing specific characteristics of code, such as flow of values between variables, is an area of extensive research which has already produced reliable results [16], [20].

B. Masked Language Modeling (MLM)

Masked language modeling (MLM) was originally used to pre-train language models for NLP tasks [21]. This process involves replacing specific words with a special mask token, allowing the model to learn how to predict these masked tokens. Such masked language models can be very useful in the identifier renaming task where the model can be used to predict a masked identifier given the rest of the code.

C. Pseudo Log-Likelihood and Pseudo Perplexity

Perplexity measures the likelihood of the model to predict the ground truth coherently [22]. A lower perplexity value indicates that for a given masked token, the model is more likely to output the actual correct token. It is a metric used for the evaluation of language models in NLP. The perplexity *PP*(W) of a test set $W = w_1, w_2, ..., w_M$ containing M words is defined as:

$$PP(W) = \sqrt[M]{\frac{1}{P(W)}} \tag{1}$$

Where $P(W)$ is the probability of the entire test set W according to the model:

$$P(W) = \prod_{i=1}^{M} P(w_i) \tag{2}$$

While perplexity provides a measure of overall sentence probability for autoregressive language models (*i.e.,* models that generate text token-by-token sequentially), an analogous metric called *pseudo log-likelihood* (PLL) can be defined for masked language models such as CodeBERT. Pseudo log-likelihood aims to estimate the probability of each token by masking it and predicting it based on the rest of the sentence as context. The pseudo log-likelihood of a sentence is then computed as the sum of the log probabilities of predicting each token [23]. Therefore, pseudo log-likelihood provides a way to get comparable probability estimates and evaluation metrics for bidirectional masked language models, paralleling the role perplexity plays for left-to-right autoregressive models. The formal definition of pseudo-likelihood is given as:

$$\text{PLL}(S) = \sum_{t=1}^{N} \log P_{\text{MLM}}(s_t \mid S_{\setminus t}) \tag{3}$$

where P_{MLM} is the masked language model used for scoring, S represents a sequence of length N composed of a set of tokens s and $S_{\setminus t}$ is the context that represents all tokens that precede and succeed the token s at index t. In other words, $S_{\setminus t} = (s_1, \ldots, s_{t-1}, s_{t+1}, \ldots, s_N)$.

However, Kauf and Ivanova [24] found that this approach leads to inflated scores for out-of-vocabulary words, which are split into multiple subword tokens by the tokenizer. Specifically, when scoring each subword token, the original PLL method allows the model to exploit the availability of future word pieces, making the subtokens deceptively easy to predict. To address this issue, they proposed an adjusted metric called `PLL-word-l2r`. In this method, when scoring a subword token from a particular word, they mask not only that target token but also all subsequent subword tokens belonging to the same word. For example, to compute the PLL for the word "*souvenir*" which is tokenized into [`so, ##uven ,##ir`], they would score "`so`" using "`[MASK] [MASK] [MASK]`" as context instead of just "[`##uven, ##ir`]". The model is therefore unable to rely on future word pieces when scoring each subtoken. They compute the overall PLL for the word by summing the scores of each subtoken masked in this left-to-right manner. This metric is defined below:

$$\text{PLL}_{\text{l2r}}(S) = \sum_{w=1}^{|S|} \sum_{t=1}^{|w|} \log P_{\text{MLM}}(s_{w_t} \mid S_{\setminus s_{w_{t'} \geq t}}) \tag{4}$$

Pseudo-perplexity (PPPL) provides an estimate of how *surprising* a sentence is to a bidirectional masked language model. It is computed based on the pseudo-log-likelihood (PLL) score. As previously discussed, the `PLL-word-l2r` metric estimates the PLL of a sentence by masking and scoring tokens in a left-to-right within-word manner. Once the PLL score is computed using this method, a pseudo-perplexity score is derived as follows:

$$\text{PPPL}(S) = \exp\left(-\text{PLL}(S)\right) \tag{5}$$

That is, the PPPL is calculated by taking the exponential of negative PLL. A lower pseudo-perplexity score indicates the sentence is more probable and less surprising to the model.

III. Methods

In this section, we elaborate on the proposed method for identifier renaming. We first elaborate on the extraction process of identifier names. Then, we describe our method for fine-tuning a language model of code on the task of identifier renaming including the strategy used during inference. An overview of the process is presented in Figure 1.

A. Dataset creation process

In steps 1 and 2 in Figure 1, we first create a corpus consisting of $28,349$ Java classes collected from a list of 50 open-source projects on GitHub. We used a curated list by IssueHunt, a Japanese bug bounty platform for open-source projects. This list details repositories primarily from well-known organizations and covers various domains. Additionally, we verified the quality of the repositories in the list by ensuring they had a minimum of $5,000$ stars, detailed README files, a significant number of contributors and forks, and varied domains. The list of the selected projects can be found in our replication package (data/SelectedRepositories.csv) Given their popularity and activity, these repositories are believed to be well-maintained, adhering to software development best practices and design principles. Hence, the identifier names used in such code bases would naturally exhibit high quality in terms of readability and expressiveness.

Using these classes, we create a dataset $\mathcal{D}$ of $236,745$ samples, where each sample is a pair (C, v). C is a Java class and v is a variable from the set of variables V_C defined in C. We then choose $1,000$ samples randomly as our training dataset for the fine-tuning approach, and another 100 samples, sourced from different repositories than the training dataset, for the test dataset.

$$\mathcal{D} = (C, v) \mid C \in \mathcal{C}, v \in V_C$$

The rationale of using the full Java class, instead of solely relying on methods as done in other works [12] is to allow the incorporation of a larger context window that incorporates additional information. With the same aim, we also keep comments at different levels, *i.e.,* line comments, block comments, and class-level comments. To construct V_C, we follow filtering steps to ensure that low-quality names are not included into the dataset.

Length-based filtering: Very short identifiers do not contribute to code readability [6]. Therefore, we excluded identifiers smaller than or equal to four characters. We specifically found four as the threshold, as for five character words, more than sixty percent of the variables were nouns present in the NLTK English dictionary [25]. The presence in the English dictionary can be linked to fewer abbreviations. Further, nouns and noun phrases are encouraged in variable names as variables frequently refer to entities [26], and a higher ratio of nouns could signify the names were more descriptive of their true functions. This requirement filters out common instances like single-letter variable names, which are often used in the context of iterators. This helps prevent the model from developing a bias towards shorter names during fine-tuning.

Uniqueness-based filtering: Variable names that end with digits treating trailing numbers as a distinguisher are considered bad variable names [27]. This rule ensures that each variable has a distinct name rather than relying on appending numbers to similar variables. Unique names enable developers to differentiate variables based on their specific purpose

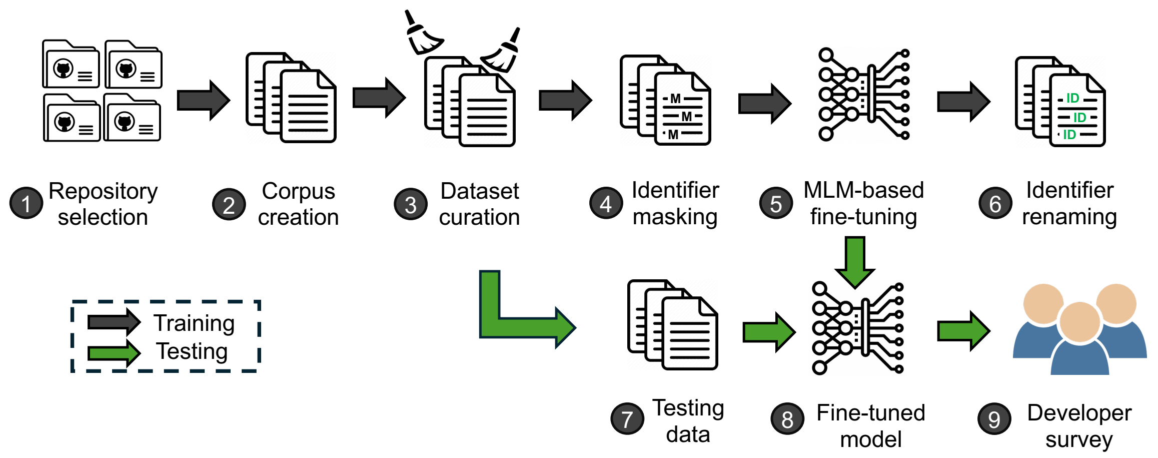


Fig. 1: Overview of the identifier renaming process. The first steps include open-source projects mining from GitHub and dataset curation. Then we fine-tune a language model of code using a masked language modeling-based method to properly capture context and generate higher-quality variable names. To further validate our approach, we conduct a developer study to see the extent this method aligns with practitioners' preferences.

instead of using numbers to distinguish otherwise identical names.

Frequency-based filtering: We identify a list of the most commonly used identifiers, such as "count", "result", "output", and "index". Such identifiers are quite abstract and ambiguous due to their unclear meaning and applicability in countless places in code. For example, "index" can refer to an index for authors or an index for books in a library management system.

The pruning strategies do not imply that variable names exhibiting those particular traits are always low readability. We want to bias the model away from certain naming behaviors, like using numbers as distinguishers; thus, we do not include those variable names in the training data. Figure 2 illustrates the percentage of identifier names that are kept and those that are discarded using the aforementioned criteria.

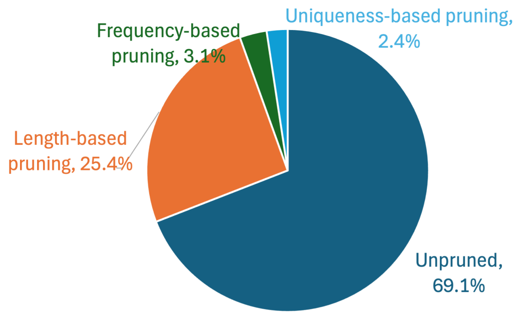


Fig. 2: Distribution of identifiers pruned using various pruning strategies

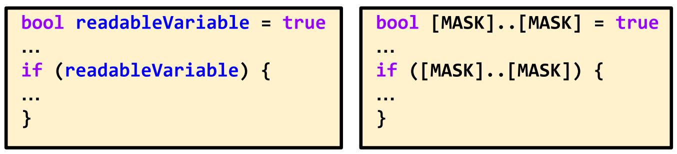
TABLE I: Dataset specifications

Train	Test	Repos	Avg. LoC	Avg. tokens
1,000	100	50	616.32	11,112.2

Table I summarizes the dataset used in this study. We use different sets of repositories to create training and testing samples, following best practices for creating training and testing datasets.

B. *Fine-tuning using Masked Language Modeling*

As a pre-processing step, for each sample in the dataset, we start by tokenizing the class C and masking all of the occurrences of the variable v in that sample. Moreover, given that we are constrained by the language model's maximum context length L, we divide the tokenized class into blocks of size less or equal to L. In the tokenization process, an identifier can be split into multiple tokens given that these models use frequency-based tokenization schemes such as Byte-Pair Encoding. For example, a `readableVariable`, would be split into `readable` and `Variable` as two separate tokens. Hence, when we mentioned that we masked the occurrences of each variable, we meant that we masked each token of that variable in each of its occurrences as illustrated in Figure 3.



```
bool readableVariable = true
...
if (readableVariable) {
...
}
```

```
bool [MASK]..[MASK] = true
...
if ([MASK]..[MASK]) {
...
}
```

Fig. 3: Masking tokens related to a specific variable at each occurrence of that variable.

Another important remark is that we only keep the blocks that contain at least one occurrence of the variable that encompasses all of its tokens. We do this to guarantee that an variable occurrence is not *split* in-between blocks.

Algorithm 1 describes the steps followed to fine-tune GRAPHCODEBERT on the variable renaming task. We run a forward pass on each block and collect the logits of all masked tokens as shown in L7. In L8-9, we gather all logits given by

Algorithm 1 Fine-tuning of a Language Model for Identifier Renaming

```
Require:
1: $B$: Set of blocks of a class $C$ containing a masked variable $v$.
2: $T_{GT}$: List of the ground truth values of the tokens of the variable $v$.
3: $M$: Pre-trained language model
4: procedure FINETUNELM($B, V_{GT}, M$)
5:     loss $\leftarrow 0$
6:     for $block \in B$ do
7:         $logits \leftarrow M(block)$        ▷ Get the logits of all masked tokens
8:         for $t \in T_{GT}$ do
9:             $t_{\text{logits}} \leftarrow \text{GATHER}(t, logits)$
10:            $t_{\text{logits}} \leftarrow \text{MEAN}(t_{\text{logits}})$
11:            $t_{\text{prob}} \leftarrow \text{SOFTMAX}(t_{\text{logits}})$
12:            $t_{\text{pred}} \leftarrow \text{ARGMAX}(t_{\text{logits}})$
13:            loss $\leftarrow$ loss + CROSSENTROPYLOSS($t_{\text{pred}}, t$)
14:        end for
15:    end for
16:    loss $\leftarrow \frac{\text{loss}}{\text{len}(B)}$
17:    $\nabla_M \leftarrow$ BACKPROP(loss)
18:    $M \leftarrow$ UPDATEPARAMS($M, \nabla_M$)
19:    return $M$
20: end procedure
```

the forward pass that corresponds to the ground token t of a variable to be predicted. For example, if we have a variable named [set, value][2], we gather the logits of the masked instances of set. In L10-12, we aggregate these logits by taking their mean, and we transform them into a probability distribution to obtain the predicted value of this token. Finally, we accumulate the prediction loss. At the end, the model's parameters are updated with the respect to the average loss per block.

C. Running inference

During inference, typically, we expect to receive a code snippet, mask its identifiers and then predict (or rename) their values. A major issue in following this approach is rooted in the nature of the tokenizers used in language models. These tokenizers are trained on a specific corpus and do not adhere to rule-based tokenization that considers the naming conventions of identifiers. Instead, the tokenization process is guided by the learned properties of the training corpus. Hence, the tokenizer may not always split identifiers according to their naming conventions, such as camel case or snake case. For instance, the identifier `getValue` may not be tokenized as `get` and `Value`, which would align with the camel case convention. Instead, it could be tokenized as `getVal` and `ue`, depending on the frequency and co-occurrence of subword units in the training corpus. This is an issue since the number of tokens can lead the model to fail at generating proper names.

To avoid this issue, we introduce *confidence* based inference. We simply set a hyperparameter T that represents the number of tokens that should be generated to form the new variable and return the result with the highest confidence (or lowest perplexity). We do this at inference only because if we were to tokenize the identifiers according to their naming conventions, it is not guaranteed that the resulting tokens will be part of the corpus. Consequently, these out-of-vocabulary tokens will be replaced with a special token[3], adding unnecessary noise and impeding the model from proper learning.

IV. EXPERIMENTS

The study aims to leverage the capacity of language models of code to capture the semantics of source code to suggest effective identifiers. Toward this goal, we formulate these research questions.

RQ1: *To what extent does the proposed masked language modeling-based method enhance the model's performance in identifier renaming tasks?*

This research question aims to evaluate the impact of our approach in enabling the model to accurately understand the context surrounding variables, ultimately leading to the generation of more appropriate and meaningful alternative identifiers.

RQ2: *What is the effect of using the proposed confidence-based inference in generating identifier names?*

We aim to evaluate the impact of the proposed inference method on the quality of generated identifier names, particularly in addressing the limitations of the tokenizer used by the GRAPHCODEBERT. Specifically, the goal is to measure the extent to which this *confidence-based* approach improve the model's performance during inference compared to relying on a typical tokenization.

RQ3:*How effective is the overall approach in generating high quality in practice?*

To further validate the effectiveness of the proposed method, we conduct a survey involving software practitioners to assess the quality of the generated identifiers from their perspective. This survey aims to provide a comprehensive evaluation by comparing the identifiers generated by our approach with those produced by state-of-the-art large language models, such as Gemini [28] and GPT4 [29]. We seek to assess the practical utility and acceptability of the generated identifiers within the software development community.

A. Model and baseline selection

In this paper, we use GRAPHCODEBERT [20] as our masked language model to process the code snippet and fine-tune it to generate identifier names as it has been used in many studies recently [30], [31], [32]. In addition, we opted for this specific transformer model given that it was pre-trained on different modalities of code, including data flow graphs that capture the semantics between the variables of a code snippet, making it a suitable candidate for the identifier renaming task.

As a baseline for comparison, we employ the GRAPHCODEBERT without any fine-tuning. This serves as a valid baseline given that it has been pre-trained on a large corpus of code,

[2]This is the tokenized version. Original variable name is `setValue`

[3]Many language models use **`[UNK]`** to represent out-of-vocabulary tokens.

enabling it to capture the general patterns and semantics of programming languages. By evaluating the performance of GraphCodeBERT without our fine-tuning strategy, we can assess the effectiveness of our proposed method in improving the model's ability to generate appropriate identifier names beyond its initial pre-training.

Identifier renaming overcomes the challenge of ensuring the new name is not only relevant, but also better than other relevant names. This distinction separates this problem from others in similar domains such as code completion and method naming [33] [34]. Such domains are similar yet sufficiently different than identifier renaming because, for example, code completion methods generate all kinds of identifiers and are not explicitly trained to suggest good variable names. Also, for many tokens, code completion methods are expected to generate only one correct result (for example, API calls). Finally, code completion methods are trained to generate the whole statement or even sometimes the block of code, which is helpful but quite different from the specific application to suggest alternative identifiers. Moreover, code completion usually takes a single stream of tokens where the n-th token is generated based on [0, n-1] tokens. In our case, we incorporate multiple streams where each represents the context of the occurrence of the variable in the code snippet to generate multiple identifier names with varying lengths.

B. Experimental setup

We ran experiments on a machine equipped with AMD Rome 7532 64-core CPU and 490 GB memory allocated per job. The experiment code is implemented in Python 3.10 using the *transformers* and *Pytorch* libraries. The *Javalang* library is used for parsing the Java code samples As for the hyperparameters, we fine-tuned all models for three epochs and used the AdamW optimizer with a learning rate of $2e-5$. In addition, all experiments were run on the same seed value to account for randomness.

V. Results and Analysis

In this section, we present the results of the experiments to answer our research questions.

A. RQ1: Impact of the fine-tuning method

In Table II, we report the performance of GraphCodeBERT using our fine-tuning technique compared to the baseline in terms of pseudo-likelihood (PLL). The overall PLL of a model is calculated by taking the average of the PLL values of each training sample using Equation 4. Similarly, the pseudo-perplexity (PPPL) value is computed in two steps. First, we sum up the unnormalized pseudo log-likelihood (PLL) values for each predicted token, obtaining the total PLL for all predictions. Then, we divide this total PLL by the total number of predicted tokens. Finally, taking the exponential of this resulting quotient gives us the final pseudo-perplexity value.

The results demonstrate a substantial reduction in both pseudo-likelihood and pseudo-perplexity for the fine-tuned model compared to the base model. This considerable decrease in values indicates that our fine-tuning method significantly enhanced model performance in predicting identifier names. Specifically, the lower PLL and pseudo-perplexity values signify that the fine-tuned model was far more likely to generate the ground truth identifier names than the base model. In other words, our fine-tuning procedure enabled more accurate predictions and a higher probability of producing the correct variable names from the model.

TABLE II: Pseudo-likelihood (PLL) and pseudo-perplexity (PPPL) for the fine-tuned and base model. The lower metric values are better.

	PLL	PPPL
Baseline model	5.572	363.209
Fine-tuned model	**3.157**	**36.796**

Such a performance improvement can be attributed to a multitude of reasons. First, despite the relatively limited size of the training dataset, each example contained extensive code as context coupled with high-quality variable names. Exposure to these well-formed and expressive identifier names increased the model's proficiency in generating identifier names with similar properties and reduces reliance on generic and less expressive names. Additionally, differences in training objectives contributed to the performance gap. The base model approached the task as a generic code completion task rather than being geared toward identifier prediction. This resulted in a higher likelihood of predicting keywords over coherent names, inflating perplexity values for the base model. In contrast, the fine-tuned model is explicitly optimized for producing complete variable names, making it suitable for the intended task.

Summary: The proposed fine-tuning method for GraphCodeBERT led to a substantial improvement in predicting identifier names compared to the base model and the generations from the fine-tuned model integrate well with the code base. This is evident from the much lower PLL and PPPL values observed on the test set.

B. RQ2: Impact of confidence-based inference on performance

The confidence-based approach is a method introduced to address the limitations of the tokenizer used by the GraphCodeBERT model, or any other language model. During inference, we replace an identifier with a configurable number of masks. In our experiments, we tested values that range from one to six. The candidate with the highest confidence score (*i.e.,* lowest perplexity) is then selected as the final generated identifier name. This approach helps to mitigate the issue of the tokenizer splitting identifiers in a way that does not align with common naming conventions.

For example, consider the identifier `getStudentName`. The tokenizer might split it into `getStud`, `ent`, and `Name`,

TABLE III: Comparison of model outputs using different inference methods.

Code Snippet	Normal inference	Confidence-based inference
`public void injectExceptionAndMessage( Throwable throwable, String [MASK] ) { exceptionsThrown.add(Pair.of( [MASK] , throwable)); }`	`message`	`errorMessage`
`Diff getDiff( WalkableGraph [MASK] , Version fromVersion, Version toVersion);`	`graphGraph`	`walkableGraph`
`final JButton [MASK] = new JButton("Run␣Test");`	`list`	`runTestButton`

which is not *semantically* correct. As a result, it will steer the model to generate values of three tokens, rather than two. Another scenario is when an identifier is represented by only one mask when using the language model's tokenizer. This limits the *expressiveness* of the generated name.

TABLE IV: PLL and PPPL upon using the number of tokens in the original variable name and upon using the confidence-based approach for choosing the number of tokens. The lower metric values are better.

	PLL	PPPL
Normal inference	3.157	36.796
Confidence-based approach	**1.497**	**5.776**

Table IV summarizes the performance between confidence-based inference and normal inference. Our inference strategy achieves significantly lower values for both PLL (1.497) and PPPL (5.776), compared to the baseline (PLL: 3.157, PPPL: 36.796). These results indicate that the confidence-based approach generates more probable, coherent, and meaningful identifier names by considering multiple candidate names with a configurable number of masks and selecting the most confident one.

In Table III, we illustrate some code examples to showcase the difference between both methods during inference. In the first code snippet, the normal inference method generates the identifier `message` for the masked variable, which is a reasonable choice given the context. However, the confidence-based inference method generates the identifier `errorMessage`, which provides a more specific and descriptive name for the variable. The term *error* suggests that the message is associated with an exception or an error condition, making the code more readable and self-explanatory. The second example involves a method that retrieves the difference between two objects of type *WalkableGraph*. The normal inference method yields the name `graphGraph` for the masked parameter, which is redundant and does not add any meaningful information. In contrast, our technique generates the identifier `walkableGraph`, which correctly matches the type of the parameter. The final example showcases the creation of a `JButton` with the label *Run Test*. Normal inference renames the original to identifier `list`, which is not semantically meaningful in this context. On the other hand, the confidence-based inference method generates the identifier `runTestButton`, which accurately describes the purpose and functionality of this button. By incorporating the action (*run*) and the object (*test*) along with the UI component type (*button*), the generated identifier provides a clear and concise name for the variable. This further highlights how confidence-based generation increases the expressiveness of identifiers.

Choosing the number of tokens for identifier renaming during inference is a key step for better generation. Although it can be slower given that it requires multiple passes, it results in superior performance. This is especially significant for low-quality code, where single-character names are frequently used. Depending solely on the original name lengths can hinder performance by constraining token counts.

Summary: Confidence-based generation outperforms normal inference in producing meaningful and contextually appropriate identifier names. By considering multiple candidate names and selecting the one with the highest confidence score, this approach generates identifiers that accurately reflect the intended meaning and functionality of the variables. Moreover, experimenting with various candidates allows for greater variety and adaptability to different coding contexts. In contrast, normal inference tends to generate more generic or less relevant identifiers, which may hinder code comprehension and maintainability.

C. RQ3: Practical effectiveness of the proposed method

To evaluate the practical effectiveness of our approach, we conducted an online developers' survey [4]. We invited recent graduates working in software development organizations or graduate (senior masters' or PhD) students working mainly on software engineering topics at the Computer Science depart-

[4]Survey link: https://forms.office.com/r/xrmcwzk44H

ment of XYZ[5] university. Eight developers accepted the invite; the developers are knowledgeable in Java with an average of four years of software development experience. The response rate of GitHub developers, contributors of open-source projects, or professional developers is typically very low for such surveys. To overcome this challenge, we relied on recent students whose backgrounds span various levels of expertise in software development and open-source contributions, ensuring a diverse set of responses. Other studies in this domain have used this choice of participants

Our aim with the survey was to present the original identifier, along with the identifier generated from our approach, as well as the identifiers generated from two state-of-the-art large language models, GPT-4 Turbo and Gemini Pro.

As the first step, we took 100 Java classes from our test dataset. Then, for each class, we randomly chose one identifier and replaced all occurrences in the class with the mask token. We created a form, using Microsoft Forms, with 100 questions; each question presenting the required code to understand the question and the context, as well as the four options (*i.e.,* the original identifier, the identifier generated by our approach, and the identifiers generated from GPT-4 and Gemini Pro models). In the form, we presented the relevant portion of the code with context balancing between providing sufficient information without overwhelming the participant with the code of the entire Java class. In addition, we provided the link to the entire Java class in case the participant would like to refer to the code in its entirety. We presented the options after shuffling to the participants without informing them of their source. The participants were asked to rank the variable names from most to least suitable.

We designed a prompt following best practices [35] to generate the identifier names. We use the same prompt with the two aforementioned language models with *temperature* set to 0.2 for both models.

I have very long Java classes with all instances of one variable in the code replaced with [MASK]. I want you to predict what the ideal variable name should be that replaces [MASK]. Output the variable name and nothing else.

Given the high number of questions (*i.e.,* 100), we left the survey open for two weeks, and participants were allowed to stop and resume later. In addition, they were not required to answer all questions. Once the survey was concluded, we picked the samples that at least two respondents answered.

Figure 4 illustrates the number of times an option was chosen as the most suitable option by the participants. The results show that GPT-4 Turbo generated the most preferred names, receiving 52 votes, followed by our fine-tuned GRAPHCODEBERT model with 39 votes. The original variable names were voted the most apt in 32 cases and Gemini Pro comes the last with 24 votes. We also calculate a weighted average score, where 1 is assigned to the highest-rated option and 4 to the lowest-rated option. We observe a similar trend, with GPT Pro leading the chart with a 1.4 average score and our fine-tuned model scoring 1.7. Original identifiers and Gemini Pro obtained 1.8 and 2.0, respectively.

[5]University name hidden due to double-blind guidelines.

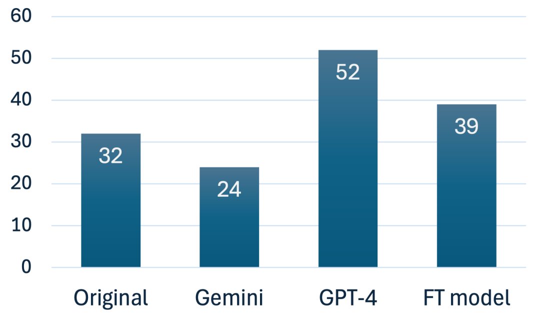


Fig. 4: Comparing number of times a model generated the most suitable identifier name for a code snippet; FT model refers to our fine-tuned model.

Our fine-tuned GRAPHCODEBERT model, despite having a smaller size compared to the commercial models, performed better than the original variable names and the Gemini Pro. This suggests that the proposed fine-tuning approach is effective in generating high-quality identifiers that are acceptable to developers. However, it is important to note that it was outperformed by GPT-4 Turbo. Naturally, this could be attributed to the larger size and more advanced architecture of GPT-4 Turbo. Further, since the training and testing datasets were open-source, GPT-4 is likely to have had access to these samples during its own training. Nevertheless, the proposed approach offers a reasonable performance, especially considering its smaller size.

Table V presents a few examples covering a variety of cases where code identifiers are generated by the fine-tuned model using our proposed method and those produced by other large language models. The first code snippet showcases an instance where the fine-tuned model generates a more contextually relevant variable name, `textWidth`, compared to the GPT-4 generation, `width`. This example suggests that the fine-tuning approach has the potential to incorporate more detailed information from the surrounding context. On the other hand, the second code snippet highlights a situation where the fine-tuned model generates an ambiguous abbreviation, `frameLayoutPar`, likely due to insufficient fine-tuning. In contrast, GPT-4 produces a more readable and self-explanatory name, `layoutParams`.

The third code snippet presents an instance where GPT-4 generates a refined variable name, `deobfuscationCheckBox`, that effectively captures the purpose of the variable. In comparison, the fine-tuned model's generation, `deobfuscationOn`, appears to be more influenced by the surrounding context.

Code snippets four and five demonstrate instances where the Gemini Pro model generates variable names that differ significantly from the other approaches. While these examples showcase Gemini's tendency to produce less predictable

TABLE V: Comparison of model outputs in the survey.

Code Snippet	**FT Model**	**LLM (GPT4 / Gemini)**	**Highest Votes**

`float x = 100, y = 20; float [MASK];`
`[MASK] = 280;`
`font.draw(`
`spriteBatch, text, x,`
`viewHeight - y, [MASK], Align.right);`
	`textWidth`	`width`	FT Model

`protected FrameLayout.LayoutParams`
`createLayoutParams(){`
`FrameLayout.LayoutParams [MASK]= new FrameLayout.`
`LayoutParams(`
`android.view.ViewGroup.LayoutParams.MATCH_PARENT,`
`android.view.ViewGroup.LayoutParams.MATCH_PARENT);`
`[MASK].gravity = Gravity.CENTER;`
`return [MASK]; }`
	`frameLayoutPar`	`layoutParams`	GPT4 / Gemini

`private SettingsGroup makeDeobfuscationGroup() {`
`JCheckBox [MASK] = new JCheckBox();`
`[MASK].setSelected(settings.isDeobfuscationOn());`
`[MASK].addItemListener(e -> {`
`settings.setDeobfuscationOn(`
`e.getStateChange() == ItemEvent.SELECTED);`
`needReload(); });`
	`deobfuscationOn`	`deobfuscationCheckBox`	GPT4

`float [MASK] = step / steps;`
`float x = bottomLeftX+graphSize*[MASK];`
`float y = bottomLeftY+graphSize*interpolation`
`.apply([MASK]);`
	`percentOffset`	`t`	FT Model

`/**`
`Sets whether to round as circle. *`
`@param [MASK] whether or not to round as circle`
`@return modified instance */`
`public RoundingParams setRoundAsCircle`
`(boolean [MASK]){`
`mRoundAsCircle = [MASK];`
`return this; }`
	`roundAsCircle`	`borderWidth`	FT Model

names, such as `t` and `borderWidth`, it's crucial to recognize that this behavior may not be consistent across all cases.

While these instances provide insights, it is important to note that they represent specific cases and may not necessarily reflect the consistent performance of each model across all scenarios. Nonetheless, they still highlight the effectiveness of *smaller* models when fine-tuned with proper modeling techniques.

Summary: The survey results highlight that the fine-tuned model, despite its smaller size compared to commercial models, generates high-quality identifiers that are acceptable to the software development community. This finding emphasizes the importance of properly modeling tasks through fine-tuning procedures, as it allows for the creation of smaller, more specialized models that can perform on par with larger models. The practical utility of our method and acceptability of the generated identifiers within the software development community, further validate the proposed approach.

VI. Threats to Validity

Construct validity refers to the extent to which a study accurately measures the concept it intends to assess. To establish construct validity in evaluating the effectiveness of our identifier generation approach, we employed both human evaluation and automated metrics. For human evaluation, we conducted a survey with experienced Java developers, providing them with sufficient context and randomly selecting identifiers to replace with mask tokens. The participants ranked the generated identifiers against relevant alternatives, including the original names and those produced by commercial models, using a structured ranking system. This approach aimed to mitigate potential biases and ensure a comprehensive assessment of identifier suitability. In addition to human evaluation, we utilized perplexity and pseudo-perplexity as automated metrics to evaluate the model's performance. These metrics provide an objective measure of the model's ability to generate appropriate and coherent identifiers.

Internal validity assesses potential confounding factors influencing outcomes. We isolated the effect of our fine-tuning techniques through controlled experiments compared to the base model, and running the experiments multiple times with different seeds.

External validity refers to the extent to which the findings of a study can be generalized to other contexts. Although we focused on evaluating our approach using Java code snippets, our proposed method is not inherently programming language-dependent. It does not rely on specific constructs or features unique to Java. The underlying ideas of our approach, notably, the use of masked language modeling and confidence-based inference, can potentially be applied to other programming languages as well. To enhance the external validity of our research, future work could involve expanding the evaluation to include a broader range of programming languages, incorporating more diverse training datasets, and fine-tuning other language models for code. Additionally, conducting studies with different participant populations, such as developers with varying levels of expertise or from different domains, could further strengthen the external validity of these findings.

VII. Related Work

A. Effect of identifier names on code readability

Previous studies evaluating code readability have highlighted the importance of apt identifier names in improving readability. Specifically, Hofmeister *et al.* [36] explored how words, letters, and abbreviations as variable names lead to differences in comprehension speed. Their results showed that proper words lead to 19% faster code comprehension than letters and abbreviations. Similarly, the impact of identifier quality on code quality has been thoroughly covered in multiple studies [37], [38], [39]. These studies establish a correlation between readability, focused on variable names, and the number of bugs in the code. For example, a study by Stegeman *et al.* [40] used a readability metric to prove that low identifier quality is associated with less readable, more complex, and less maintainable code. Similar works on code readability [41] highlighted the importance of identifier names on overall readability. Peruma *et al.* [42] analyzed general trends in identifier renaming and identified code elements that are looked during naming identifiers. Arnaoudova *et al.* [43] identified a subset of linguistic anti-patterns that were unanimously agreed to decrease code readability.

B. Identifier Renaming

Feitelson *et al.* [44] concluded from a survey of 334 developers that longer identifier names that use more concepts should be encouraged for better names. *JSNice* [45] and *JSNaughty* [46] were state-of-the-art statistical renaming tools before machine-learning approaches were used in this domain. Allamanis *et al.* [47] proposed an approach to variable naming. The authors used a log-bilinear neural language model for source code using embeddings. Their model predicted subtokens, which helped generate variable names outside the training corpus or neologisms. Bavishi *et al.* [48] proposed *Context2Name* to capture context while predicting variable names. The authors designed the tool to recreate variable names from *minified* JavaScript code.

Many Integrated Development Environments (IDEs), such as IntelliJ also include suggestions for variable names. Simpler code completion methods are the basis of these features, and the identifier names are suggested by using Abstract Syntax Tree (AST) analysis with statistical modeling similar to that adopted by JSNice.

Mastropaolo *et al.* [12] conducted an investigation into the latest techniques for identifier renaming. They rigorously tested and compared the effectiveness of existing code completion methods. The methods under scrutiny encompassed the n-gram cached language model [49] as well as transformer-based models [50] [51]. The evaluation involved training and testing on method-level code snippets. The reference standard was established by identifier names introduced or modified during a code review process. The models' performance on this task was assessed using both complete token match and partial token match metrics. The CugLM-based transformer [52] achieved the highest accuracy of 63.46%.

Limitations: Previous studies are not able to achieve comparable performance to the CugLM transformer fine-tuned on the code completion task. The limitation in using transformers for identifier renaming arises from a scarcity of research on fine-tuning transformer-based models specifically for this task. Additionally, the inputs for this task are methods, which tend to be shorter and capture less context compared to classes. Mastropaolo *et al.* [12] underscore the issue by demonstrating a significant drop in performance for the best model when dealing with methods that are 400 tokens or longer, as opposed to methods with lengths between 0 to 50 tokens. To fully leverage the potential of transformers, training with longer contexts is required. However, handling these longer sequences presents its own set of challenges. Further, there is a lack of analysis on how the transformer generated identifiers fit into the original codebase.

VIII. Conclusions

In this work, we presented a transformer-based approach for automatic identifier renaming. We curated a dataset of 236,745 variable name and Java class pairs from top GitHub repositories to fine-tune GraphCodeBERT, employing techniques to handle long code sequences. Our proposed fine-tuning method significantly improved identifier prediction over the non-finetuned base model. By incorporating the confidence-based method to choose the ideal number of tokens for prediction, our approach can generate names that are competitive with state-of-the-art LLMs, despite having a much smaller model size.

In the future, several areas could be explored to further enhance the performance and usability of our approach. One direction is to investigate training objectives that jointly predict the number of tokens needed to rename an identifier and its values. This could potentially improve the model's ability to generate identifiers more efficiently. To further enhance the practicality of our approach, future work could explore the integration of our identifier renaming model into integrated development environments (IDEs) or code editors. This would allow developers to receive real-time suggestions for identifier names as they write code, improving the overall coding experience and reducing the cognitive burden of coming up with appropriate names.

References

[1] S. Fakhoury, Y. Ma, V. Arnaoudova, and O. Adesope, "The effect of poor source code lexicon and readability on developers' cognitive load," in *Proceedings of the 26th Conference on Program Comprehension*, 2018, pp. 286–296.

[2] P. Relf, "Tool assisted identifier naming for improved software readability: an empirical study," in *2005 International Symposium on Empirical Software Engineering, 2005.*, 2005, pp. 10 pp.–.

[3] L. Jiang, H. Liu, and H. Jiang, "Machine learning based recommendation of method names: How far are we," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 602–614.

[4] T. Sedano, "Code readability testing, an empirical study," in *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*, 2016, pp. 111–117.

[5] G. Beniamini, S. Gingichashvili, A. K. Orbach, and D. G. Feitelson, "Meaningful identifier names: The case of single-letter variables," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, 2017, pp. 45–54.

[6] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "Effective identifier names for comprehension and memory," *Innovations in Systems and Software Engineering*, vol. 3, pp. 303–318, 2007.

[7] C. Charitsis, C. Piech, and J. C. Mitchell, "Function names: Quantifying the relationship between identifiers and their functionality to improve them," in *Proceedings of the Ninth ACM Conference on Learning @ Scale*, ser. L@S '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 93–101. [Online]. Available: https://doi.org/10.1145/3491140.3528269

[8] V. van der Werf, E. Aivaloglou, F. Hermans, and M. Specht, "(how) should variables and their naming be taught in novice programming education?" in *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 2*, ser. ICER '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 53–54. [Online]. Available: https://doi.org/10.1145/3501709.3544288

[9] Y. Wainakh, M. Rauf, and M. Pradel, "Idbench: Evaluating semantic representations of identifier names in source code," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 562–573.

[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[11] Z. Chen and M. Monperrus, "A literature study of embeddings on source code," *arXiv preprint arXiv:1904.03061*, 2019.

[12] A. Mastropaolo, E. Aghajani, L. Pascarella, and G. Bavota, "Automated variable renaming: are we there yet?" *Empirical Software Engineering*, vol. 28, no. 2, p. 45, 2023.

[13] Anonymous, "Replication package," https://github.com/scam2024-conf-reidentify/scam-submission-24, Jan. 2023.

[14] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE Press, 2012, p. 837–847.

[15] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, T. Cohn, Y. He, and Y. Liu, Eds. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139

[16] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy,

D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder: may the source be with you!" 2023.

[17] J. von der Mosel, A. Trautsch, and S. Herbold, "On the validity of pre-trained transformers for natural language processing in the software engineering domain," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1487–1507, 2023.

[18] T. Zhang, B. Xu, F. Thung, S. A. Haryono, D. Lo, and L. Jiang, "Sentiment analysis for software engineering: How far can pre-trained transformer models go?" in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 70–80.

[19] Q. Zhang and B. Wu, "Software defect prediction via transformer," in *2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, vol. 1. IEEE, 2020, pp. 874–879.

[20] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[21] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[22] D. Colla, M. Delsanto, M. Agosto, B. Vitiello, and D. P. Radicioni, "Semantic coherence markers: The contribution of perplexity metrics," *Artificial Intelligence in Medicine*, vol. 134, p. 102393, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0933365722001440

[23] J. Salazar, D. Liang, T. Q. Nguyen, and K. Kirchhoff, "Masked language model scoring." Association for Computational Linguistics, 2020. [Online]. Available: https://doi.org/10.18653%2Fv1%2F2020.acl-main.240

[24] C. Kauf and A. Ivanova, "A better way to do masked language model scoring," *arXiv preprint arXiv:2305.10588*, 2023.

[25] S. Bird and E. Loper, "NLTK: The natural language toolkit," in *Proceedings of the ACL Interactive Poster and Demonstration Sessions*. Barcelona, Spain: Association for Computational Linguistics, Jul. 2004, pp. 214–217. [Online]. Available: https://aclanthology.org/P04-3031

[26] S. Butler, M. Wermelinger, and Y. Yu, "A survey of the forms of java reference names," in *2015 IEEE 23rd International Conference on Program Comprehension*, 2015, pp. 196–206.

[27] A. Peruma and C. D. Newman, "Understanding digits in identifier names: An exploratory study," in *Proceedings of the 1st International Workshop on Natural Language-Based Software Engineering*, ser. NLBSE '22. New York, NY, USA: Association for Computing Machinery, 2023, p. 9–16. [Online]. Available: https://doi.org/10.1145/3528588.3528657

[28] G. Team, R. Anil, S. Borgeaud, Y. Wu, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth *et al.*, "Gemini: a family of highly capable multimodal models," *arXiv preprint arXiv:2312.11805*, 2023.

[29] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[30] A. Karmakar and R. Robbes, "What do pre-trained code models know about code?" in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 1332–1336.

[31] Y. Wan, W. Zhao, H. Zhang, Y. Sui, G. Xu, and H. Jin, "What do they capture? a structural analysis of pre-trained language models for source code," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 2377–2388. [Online]. Available: https://doi.org/10.1145/3510003.3510050

[32] Z. Yang, J. Shi, J. He, and D. Lo, "Natural attack for pre-trained models of code," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1482–1493. [Online]. Available: https://doi.org/10.1145/3510003.3510146

[33] J. Zhu, L. Li, L. Yang, X. Ma, and C. Zuo, "Automating method naming with context-aware prompt-tuning," in *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*. IEEE, 2023, pp. 203–214.

[34] J. Lacomis, P. Yin, E. Schwartz, M. Allamanis, C. Le Goues, G. Neubig, and B. Vasilescu, "Dire: A neural approach to decompiled identifier naming," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 628–639.

[35] OpenAI, "GPT best practices," https://platform.openai.com/docs/guides/gpt-best-practices, 2023.

[36] J. Hofmeister, J. Siegmund, and D. V. Holt, "Shorter identifier names take longer to comprehend," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 217–227.

[37] S. Butler, "The effect of identifier naming on source code readability and quality," in *Proceedings of the Doctoral Symposium for ESEC/FSE on Doctoral Symposium*, ser. ESEC/FSE Doctoral Symposium '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 33–34. [Online]. Available: https://doi.org/10.1145/1595782.1595796

[38] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Relating identifier naming flaws and code quality: An empirical study," in *2009 16th Working Conference on Reverse Engineering*, 2009, pp. 31–35.

[39] ——, "Exploring the influence of identifier names on code quality: An empirical study," in *2010 14th European Conference on Software Maintenance and Reengineering*, 2010, pp. 156–165.

[40] M. Stegeman, E. Barendsen, and S. Smetsers, "Towards an empirically validated model for assessment of code quality," in *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*, ser. Koli Calling '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 99–108. [Online]. Available: https://doi.org/10.1145/2674683.2674702

[41] R. P. Buse and W. R. Weimer, "A metric for software readability," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 121–130. [Online]. Available: https://doi.org/10.1145/1390630.1390647

[42] A. Peruma, "Towards a model to appraise and suggest identifier names," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 639–643.

[43] V. Arnaoudova, M. Di Penta, and G. Antoniol, "Linguistic antipatterns: What they are and how developers perceive them," *Empirical Software Engineering*, vol. 21, pp. 104–158, 2016.

[44] D. G. Feitelson, A. Mizrahi, N. Noy, A. B. Shabat, O. Eliyahu, and R. Sheffer, "How developers choose names," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 37–52, 2022.

[45] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from "big code"," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 111–124. [Online]. Available: https://doi.org/10.1145/2676726.2677009

[46] B. Vasilescu, C. Casalnuovo, and P. Devanbu, "Recovering clear, natural identifiers from obfuscated js names," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 683–693. [Online]. Available: https://doi.org/10.1145/3106237.3106289

[47] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 38–49. [Online]. Available: https://doi.org/10.1145/2786805.2786849

[48] R. Bavishi, M. Pradel, and K. Sen, "Context2name: A deep learning-based approach to infer natural variable names from usage contexts," 2018.

[49] R.-M. Karampatsis and C. Sutton, "Maybe deep neural networks are the best choice for modeling source code," 2019.

[50] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," 2023.

[51] A. Mastropaolo, E. Aghajani, L. Pascarella, and G. Bavota, "An empirical study on code comment completion," 2021.

[52] F. Liu, G. Li, Y. Zhao, and Z. Jin, "Multi-task learning based pre-trained language model for code completion," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 473–485.