Generating Variable Explanations via Zero-shot Prompt Learning

Chong Wang, Yiling Lou*, Junwei Liu, and Xin Peng
School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China
{wangchong20, yilinglou}@fudan.edu.cn, 22210240218@m.fudan.edu.cn, pengxin@fudan.edu.cn

***Abstract*—As basic elements in program, variables convey essential information that is critical for program comprehension and maintenance. However, understanding the meanings of variables in program is not always easy for developers, since poor-quality variable names are prevalent while such variable are less informative for program comprehension. Therefore, in this paper, we target at generating concise natural language explanations for variables to facilitate program comprehension. In particular, there are two challenges in variable explanation generation, including the lack of training data and the association with complex code contexts around the variable. To address these issues, we propose a novel approach ZEROVAR, which leverages code pre-trained models and zero-shot prompt learning to generate explanations for the variable based on its code context. ZEROVAR contains two stages: (i) a pre-training stage that continually pre-trains a base model (i.e., CodeT5) to recover the randomly-masked parameter descriptions in method docstrings; and (ii) a zero-shot prompt learning stage that leverages the pre-trained model to generate explanations for a given variable via the prompt constructed with the variable and its belonging method context.**

We then extensively evaluate the quality and usefulness of the variable explanations generated by ZEROVAR. We construct an evaluation dataset of 773 variables and their reference explanations. Our results show that ZEROVAR can generate higher-quality explanations than baselines, not only on automated metrics such as *BLEU* and *ROUGE*, but also on human metrics such as *correctness*, *completeness*, and *conciseness*. Moreover, we further assess the usefulness of ZEROVAR-generated explanations on two downstream tasks related to variable naming quality, i.e., abbreviation expansion and spelling correction. For abbreviation expansion, the generated variable explanations can help improve the *present rate* (+13.1%), *precision* (+3.6%), and *recall* (+10.0%) of the state-of-the-art abbreviation explanation approach. For spelling correction, by using the generated explanations we can achieve higher *hit@1* (+162.9%) and *hit@3* (+49.6%) than the recent variable representation learning approach.

***Index Terms*—variable explanation, naming quality, code pre-trained models, prompt learning**

I. INTRODUCTION

Variables play a critical role in programming languages by serving as the basic building blocks for storing and manipulating data within software programs. Beyond their functional role, variables also provide essential information that supports program comprehension and maintenance by conveying the meaning of the data they store. Properly assigning meaning to variables is crucial and involves selecting names that accurately reflect the purpose and significance of the data they represent. By doing so, other developers can reduce the time and effort required for debugging, maintenance, and reuse by quickly grasping the intent of code.

However, poor-quality variable names are prevalent in practice, since naming variables appropriately based on their meanings is not a simple task for developers, which should not only select proper descriptive nouns or noun phrases but should also ensure the consistency and conciseness in naming [1]. As a result, it is common that poor-quality variable names are introduced into program due to the developer oversight, time constraints, or lack of clarity in requirements. Such poor-quality variable names increase the difficulty for developers to understand the meaning of the variables, which further leads to decreased readability and maintainability of the whole program.

Therefore, in this work, we propose to generate concise natural language explanations for variables, so as to facilitate program comprehension and maintenance. Although there are already existing works that generate explanations for coarse-grained code elements (such as methods or blocks) [2], [3], [4], [5], [6], [7], we still argue that generating variable-level explanation is necessarily beneficial, since such fine-grained explanations could contain detailed information that is complementary to coarse-grained explanations. For example, variable-specific refactoring (i.e., variable rename) and bug localization can benefit from better variable comprehension [8], [9]. However, generating variable explanations can be very challenging. First, different from coarse-grained explanation generation (e.g., method explanation) which has sufficient available method-comment pairs as training data, there is a lack of *high-quality training data* for variable explanations in the wild, since it is verbose and uncommon for developers to write such fine-grained explanations in practice. Even there might be some inline comments within the code, they do not necessarily align with specific variables in the code [10], [11]. Second, the variable names alone cannot provide sufficient information to infer their meaning, e.g., it is common that two variables with a same name but in the different code contexts have different meanings at all. Therefore, generating precise explanations for a variable should be fully aware of the complex code context around the variable.

To address the aforementioned challenges, our idea is to use a code pre-trained model as a code semantic knowledge base,

* Y. Lou is the corresponding author

and generate natural language explanations for variables using zero-shot prompt learning. Prompt learning is an emerging paradigm for pre-trained models, which can effectively tackle the *few- or zero-shot issues* in downstream problems [12]. Its success lies in modeling *downstream problems* in the same form as the *pre-training objectives*, and constructing *appropriate prompt templates* to facilitate knowledge transfer between pre-training objectives and downstream problems [13]. More specifically, we observe that methods often have corresponding docstrings (e.g., Javadoc comments in Java), which describe the functionality of the entire method contexts and the method parameters. We can use such methods with docstrings to pre-train a model that can generate parameter descriptions. Since variables and parameters are similar in form and functionality, we can treat variables as a special type of parameter and use zero-shot prompt learning to solve the variable explanation problem (i.e., the downstream problem) by aligning it to parameter description generation (i.e., the pre-training objective).

Our proposed approach, called ZeroVar, aims to generate natural language explanations for variables via zero-shot prompt learning. The approach consists of two stages: (i) continual pre-training for parameter description generation, and (ii) zero-shot prompt learning for variable explanation generation. During the continual pre-training stage, we employ CodeT5 [14] as our base model, which can process bimodal input (i.e., method docstring and code) and support varying-length description generation. We continually pre-train the base model by using description masking objective, which randomly masks parameter descriptions in method docstrings and have the model learn to recover them. During the zero-shot prompt learning stage, given a method and a variable that requires explanation, we first use the Unixcoder model [15] to generate a docstring for the method, then insert the variable into the docstring. This process allows us to create a prompt where the variable is simulated as a pseudo parameter, which the continually pre-trained CodeT5 can utilize to generate the corresponding explanation for the variable.

We then extensively evaluate the *quality* and *usefulness* of the variable explanations generated by ZeroVar. For the quality assessment, we construct a *reference* dataset of 773 variables and their reference explanations, and then apply ZeroVar to generate variable explanations for the variables in the dataset. The results show that ZeroVar substantially outperforms the baseline model (i.e., the base CodeT5 model) on different automated metrics such as BLEU [16] and ROUGE [17]. Moreover, we conduct a manual evaluation to compare the quality of the variable explanations generated by ZeroVar, the ones generated by the baseline model, and the reference explanations in the dataset. The results show that the explanations generated by ZeroVar are more correct, complete, and concise for manually understanding the meaning of variables. For the usefulness assessment, we investigate the usefulness of the variable explanations generated by ZeroVar in two downstream tasks related to variable naming quality improvement, i.e., abbreviation expansion and spelling correction. For abbreviation expansion, we leverage the generated variable explanations as additional contexts and find they can help improve the present rate (+13.1%), precision (+3.6%), and recall (+10.0%) of state-of-the-art abbreviation explanation approach [18] on a dataset containing 868 abbreviation instances. For spelling correction, we apply a simple rule to select corrections from the generated variable explanations and achieve higher hit@1 (+162.9%) and hit@3 (+49.6%) than a recent variable representation learning approach [19] on a dataset containing 1,023 misspelling instances.

To summarize, this paper makes the following contributions:

- A novel approach ZeroVar that automatically generates variable explanations based on code contexts by leveraging code pre-trained models with zero-shot prompt learning;
- Extensive evaluation that demonstrate the high quality of the variable explanations generated by ZeroVar in terms of both automated metrics and human evaluation metrics;
- Extensive evaluation on two downstream tasks (i.e., abbreviation expansion and spelling correction) that confirm the usefulness of the variable explanations generated by ZeroVar for improving naming quality of variables.

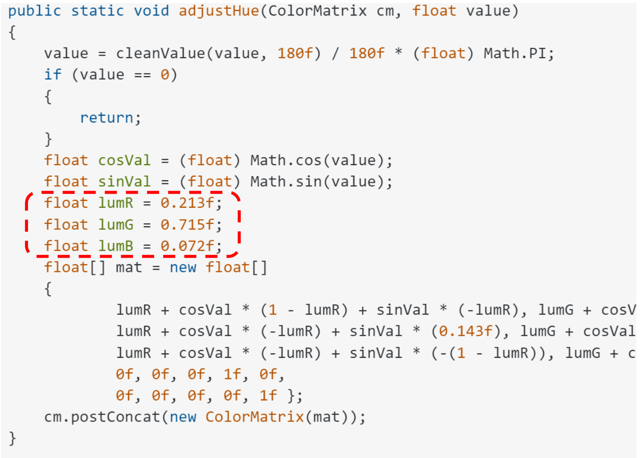
II. Motivating Example

The code example presented in Figure 1(a) is extracted from the accepted answer of a Stack Overflow question that demonstrates how to adjust the hue[1] of a drawable using color matrix. Although the code is functionally correct, its reusability is sometimes hindered by difficulties in understanding the true meaning of certain variables. For instance, Figure 1(b) showcases two questions asked by other developers regarding the meaning of the variables *lumR*, *lumG*, and *lumB* and their corresponding values. It is noteworthy that the prefix "lum" in the three variables stands for "luminance", and the values relate to calculating relative luminance from linear RGB components[2]. For developers lacking awareness that the prefixes "lum" and "r" signify "the luminance" and "red color", respectively, the task of locating pertinent resources (e.g., Wikipedia pages) and comprehending the underlying principles associated with these variables becomes challenging. Therefore, providing developers with an explanation for the "lum" and "r" in the variable can enhance their understanding and enable them to reuse the code more effectively.

On the other hand, leveraging code pre-trained models such as Unixcoder [15] allows us to generate a docstring for the code example, as shown in Figure 2. Interestingly, the model generates a more descriptive and accurate description for the parameter *value*, using the phrase "The hue", which is more consistent with the true meaning that the parameter should convey. This success in generating an accurate description for the parameter is due to the model's ability to learn and leverage knowledge about the semantic relationship between parameters and code context during pre-training. Furthermore, given that variables share functional and formal similarities with parameters, they can be treated as a special type of parameters. As a

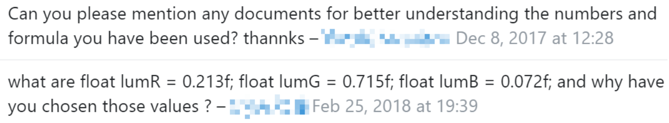
[1] https://stackoverflow.com/questions/4354939

[2] https://en.wikipedia.org/wiki/Relative_luminance



```java
public static void adjustHue(ColorMatrix cm, float value)
{
    value = cleanValue(value, 180f) / 180f * (float) Math.PI;
    if (value == 0)
    {
        return;
    }
    float cosVal = (float) Math.cos(value);
    float sinVal = (float) Math.sin(value);
    float lumR = 0.213f;
    float lumG = 0.715f;
    float lumB = 0.072f;
    float[] mat = new float[]
    {
            lumR + cosVal * (1 - lumR) + sinVal * (-lumR), lumG + cosV
            lumR + cosVal * (-lumR) + sinVal * (0.143f), lumG + cosVal
            lumR + cosVal * (-lumR) + sinVal * (-(1 - lumR)), lumG + c
            0f, 0f, 0f, 1f, 0f,
            0f, 0f, 0f, 0f, 1f };
    cm.postConcat(new ColorMatrix(mat));
}
```

(a) Code Example in Accepted Answer

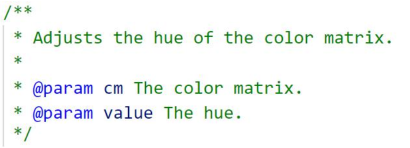


Can you please mention any documents for better understanding the numbers and formula you have been used? thannks – [redacted] Dec 8, 2017 at 12:28

what are float lumR = 0.213f; float lumG = 0.715f; float lumB = 0.072f; and why have you chosen those values ? – [redacted] Feb 25, 2018 at 19:39

(b) Developer Questions on the Code Example

Figure 1. Stack Overflow Code Example and User Questions



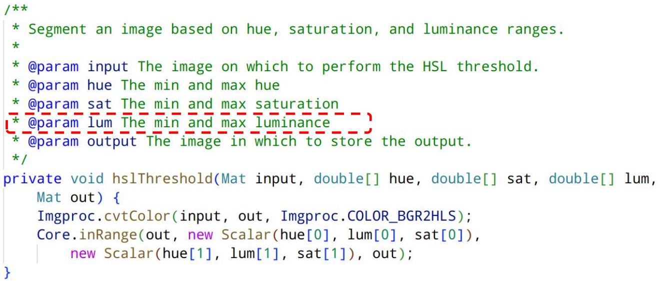
```java
/**
 * Adjusts the hue of the color matrix.
 *
 * @param cm The color matrix.
 * @param value The hue.
 */
```

Figure 2. Docstring Generated for the Code Example by Unixcoder

result, we are motivated to utilize code pre-trained models as code semantic knowledge bases and design appropriate approaches (e.g., prompt learning) to activate the models' capacity for explaining variables. For instance, assume that the code snippet depicted in Figure 3 is utilized during pre-training. In this case, the code pre-trained models can learn the association between the parameter and code context, such as the "lum" in the code refers to "luminance" according to the descriptions of the method and parameters. When presented with the code in Figure 1(a), which is contextually similar (both related to "hue" and "color"), the models can potentially deduce that the meaning of "lum" is "luminance" if it is treated as a parameter.

III. Approach

In this section, we present a novel approach called ZeroVar for addressing the variable explanation problem, which can be considered a type of parameter description generation task. The proposed approach consists of two key stages: continual pre-training for parameter description generation and zero-shot prompt learning for variable explanation generation, as shown in Figure 4.



```java
/**
 * Segment an image based on hue, saturation, and luminance ranges.
 *
 * @param input The image on which to perform the HSL threshold.
 * @param hue The min and max hue
 * @param sat The min and max saturation
 * @param lum The min and max luminance
 * @param output The image in which to store the output.
 */
private void hslThreshold(Mat input, double[] hue, double[] sat, double[] lum,
    Mat out) {
    Imgproc.cvtColor(input, out, Imgproc.COLOR_BGR2HLS);
    Core.inRange(out, new Scalar(hue[0], lum[0], sat[0]),
        new Scalar(hue[1], lum[1], sat[1]), out);
}
```

Figure 3. Another Code Snippet about Hue and Luminance

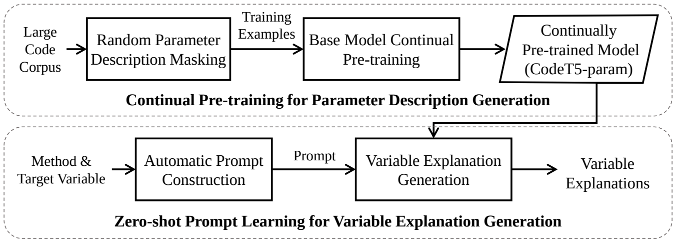

Large Code Corpus
Random Parameter Description Masking
Training Examples
Base Model Continual Pre-training
Continually Pre-trained Model (CodeT5-param)
Continual Pre-training for Parameter Description Generation
Method & Target Variable
Automatic Prompt Construction
Prompt
Variable Explanation Generation
Variable Explanations
Zero-shot Prompt Learning for Variable Explanation Generation

Figure 4. Approach Overview

The continual pre-training stage is designed to improve the model's ability to generate accurate parameter descriptions. It involves continually pre-training a base model on a large corpus of code snippets using the description masking objective, a variant of span masking. This process enables the model to acquire extensive knowledge of parameter usages and meanings, which can later be utilized in generating variable explanations.

The zero-shot prompt learning stage is responsible for generating variable explanations using the continually pre-trained model without any fine-tuning on specific labeled data for the variable explanation task. This is achieved by simulating the given variable as a special parameter and automatically constructing a prompt in the same format as the input of the description masking objective. The continually pre-trained model is then utilized to generate explanations for the variable based on the prompt.

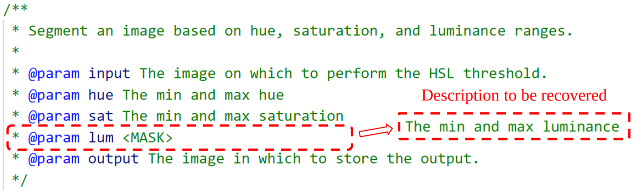
A. *Continual Pre-Training for Parameter Description Generation*

As stated previously, obtaining training data for directly explaining variables is challenging; however, we can leverage pre-trained models' ability to generate parameter descriptions. While no pre-trained model exists explicitly trained for parameter description generation, training data for this task is readily available as many methods have corresponding docstrings that contain parameter descriptions. Therefore, we begin by selecting a base pre-trained model and utilizing continual pre-training to enhance its ability to generate parameter descriptions.

1) Pre-training Objective: The continual pre-training approach employs a description masking objective, which is a variant of the span masking objective commonly used in pre-trained models such as T5 [20], CodeT5 [14], and Unixcoder [15]. The original span masking objective involves randomly masking spans with arbitrary lengths in the source input, such as method code with docstring, and then requires the model to recover the original content by predicting the masked spans. For our parameter description generation task, we adapt this objective by limiting the masked spans to parameter descriptions in docstrings. Specifically, given a method and its corresponding docstring, we randomly mask some parameter descriptions in the docstring with the placeholder <MASK>, and require the model to recover the masked descriptions based on the noised docstring and the code context.

2) Base Model Selection: To date, most state-of-the-art pre-trained language models are based on the Transformer architecture [21], an encoder-decoder model, we therefore choose this architecture as the foundation for our parameter description generation model. Meanwhile, recent advancements in the field have led to the development of numerous pre-trained code models (e.g., CodeBERT [22] and CodeT5 [14]), which have been shown to effectively capture rich code information and achieve promising results in downstream tasks. Thus, instead of training a model from scratch, we select CodeT5[3] as the base model and continue pre-training it for the purpose of generating parameter descriptions. We choose CodeT5 based on the following two criteria.

- **Bimodal Input.** Parameter descriptions are typically written in natural language and included in docstrings in the form of "*@param [paramName] [description]*" (e.g., "*@param lum* The min and max luminance" in Figure 3). Therefore, an effective pre-trained model for generating parameter descriptions must be able to process both natural language docstrings and code as input. We treat code and docstrings as separate input modalities and expect the model to learn the relationship and complex interactions between them. In this way, the model can generate natural language descriptions for parameters based on a conprehensive understanding of the code. For example, if the description of "lum" in Figure 3 is masked, the model must be able to predict it based on the remaining docstring and the code context.
- **Varying-length Description.** As parameter descriptions can have varying lengths, an effective pre-trained model for generating them must be able to control the number of tokens it needs to generate based on each unique code context. In other words, the length of the predicted description cannot be predetermined. For instance, the descriptions of "input" and "lum" in Figure 3 have different lengths, so we cannot pre-define a fixed length for each masked description. This requirement rules out encoder-only models (i.e., models that include only the encoder of the Transformer) such as CodeBERT, as they must determine the length of the descriptions before prediction. Instead, encoder-decoder models (i.e., models that include both the encoder and decoder of the Transformer) such as CodeT5 are suitable for this purpose, as they can generate token sequences of arbitrary length through the decoding process. Specifically, the decoder keeps generating tokens until it reaches the <EOS>token, ensuring that the length of the generated description is appropriate for the input code context.



```
/**
 * Segment an image based on hue, saturation, and luminance ranges.
 *
 * @param input The image on which to perform the HSL threshold.
 * @param hue The min and max hue
 * @param sat The min and max saturation
 * @param lum <MASK>
 * @param output The image in which to store the output.
 */
```

Figure 5. Noised Docstring after Masking the Description of *lum*

3) Random Parameter Description Masking: We randomly mask the parameter descriptions in method docstrings to create the training examples for the continual pre-training. Our approach currently focuses on the Java language, and we collect the Java code corpus, CodeSearchNet [23], which is utilized by the base model CodeT5 during its pre-training phase. The Java code corpus is divided into three parts upon release, namely the training set, validation set, and test set, with CodeT5 being pre-trained on the training set. We take the training set and process all the instances in the following manner. For each instance, which includes a method code and its corresponding docstring, we first use a regular expression to match the lines in the form of "*@param [paramName] [description]*" to find all the parameter descriptions in the docstring. If the docstring does not contain any parameter descriptions, the instance is skipped. Otherwise, we randomly sample one of the descriptions, mask it with <MASK>, and obtain a noised docstring. For example, consider the method and docstring in Figure 3, we can mask the description of the parameter *lum* (i.e., "The min and max luminance"), resulting in a noised docstring shown in Figure 5. Subsequently, a training example (x, y) is generated, where x is the concatenation of the noised docstring and the method code, and y is the masked description that is expected to be recovered. Based on the random description masking, we totally obtain 424,701 training examples.

4) Model Continual Pre-training: To continually pre-train the base model CodeT5, we utilize the collected training examples as follows. We begin by tokenizing each training example (x_i, y_i) into token sequences tx_i and ty_i, respectively, using the byte-level BPE tokenizer [24] that is trained by CodeT5. Next, we feed the token sequence tx_i into the encoder of CodeT5 to compute the contextual embeddings. Finally, we train CodeT5 to use its decoder to generate a token sequence ty_i' that minimizes the cross-entropy loss between ty_i' and y_i (the same to span masking objective [20]). The cross-entropy loss measures the dissimilarity between two probability distributions by calculating the negative log-likelihood of the correct distribution y_i given the predicted distribution ty_i'. To train the model parameters, we utilized the Adam optimizer [25] and set the training epoch number, learning rate, and batch size to 3, 1e-5, and 8, respectively. We denote the continually pre-trained CodeT5 as CodeT5-param.

[3] https://huggingface.co/Salesforce/codet5-base

B. Zero-shot Prompt Learning for Variable Explanation Generation

In this stage, we formulate the problem of variable explanation generation in a same format to that of the parameter description generation task and construct appropriate prompts to leverage the model's ability.

1) Automatic Prompt Construction: Given a method code snippet *md* and a variable *var* within it that requires explanation, we automatically construct a prompt for them as follows. If there are no corresponding docstrings for the method, we use the Unixcoder model [15] to generate a docstring in the desired format (e.g., javadoc format for Java language). We select the Unixcoder model[4] as the docstring generation model since it can generate docstrings directly without any further fine-tuning, while most other models, such as CodeT5, only generate one-sentence method summaries. Subsequently, we add a line *@param var <MASK>* underneath the final parameter in the docstring to simulate the variable *var* as a special parameter in the method. We then concatenate the noised docstring, after inserting *var*, with the *md* to form the prompt that is used as input to the continually pre-trained model (i.e., CodeT5-param). It should be noted that each prompt is constructed for explaining a single variable, and hence, multiple individual prompts need to be constructed when there are multiple variables to explain in a method.

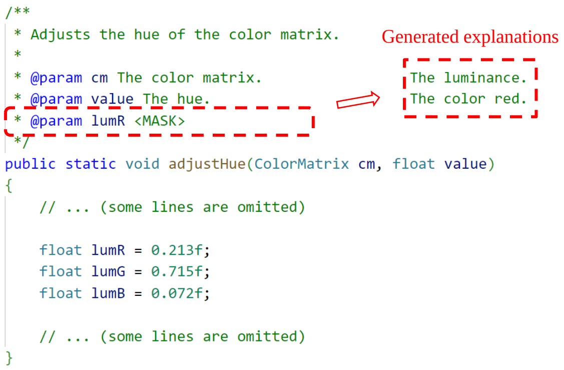
To illustrate the process, we use the example method presented in Figure 1(a). First, we generate the corresponding docstring using the Unixcoder model, as shown in Figure 2. Then, we insert a line "@param lumR <MASK>" for the variable *lumR* into the docstring, and concatenate the modified docstring with the method to create a prompt, as shown in Figure 6.

2) Variable Explanation Generation: We tokenize the constructed prompt and feed it into the encoder of the continually pre-trained CodeT5-param using the same approach as in the continual pre-training, as explained in Section III-A4. This allows the docoder of CodeT5-param to generate explanations (beam search can be applied during the docoding steps [26], [27]) for the variable *var* that describe its meaning based on the modified docstring and code context.

For example, for the prompt shown in Figure 6, we use the continually pre-trained CodeT5-param to generate explanations for the variable *lumR* based on this prompt. The model generates two possible explanations for *lumR*, which are "The luminance" and "The color red".

IV. Evaluation

We perform extensive experiments to evaluate both the quality and usefulness of the variable explanations generated by ZeroVar. For **quality**, we create a *reference* dataset comprising 773 variables and their reference explanations to evaluate the quality of generated variable explanations. In particular, we include both automated metrics and human evaluation metrics: (i) the automated metrics assess whether the variable explanations generated by ZeroVar are textually similar to the references (**RQ1.a**), and (ii) the human evaluation metrics evaluate whether the quality of the generated variable explanations is sufficient for human understanding (**RQ1.b**). For **usefulness**, we evaluate the usefulness of ZeroVar by investigating how its generated variable explanations could help improve the quality of variable naming. To this end, we investigate whether the variable explanations generated by ZeroVar are useful in two downstream variable naming tasks, *i.e.,* abbreviation expansion (**RQ2.a**) and spelling correction (**RQ2.b**).



```
/**
 * Adjusts the hue of the color matrix.
 *
 * @param cm The color matrix.
 * @param value The hue.
 * @param lumR <MASK>
 */
public static void adjustHue(ColorMatrix cm, float value)
{
    // ... (some lines are omitted)

    float lumR = 0.213f;
    float lumG = 0.715f;
    float lumB = 0.072f;

    // ... (some lines are omitted)
}
```

Figure 6. Constructed Prompt for the Variable *lumR* in Figure 1(a)

All the research questions are listed as follows.

- **RQ1 (Quality)**
  - **RQ1.a (Automated Metrics):** How textually similar are the variable explanations generated by ZeroVar to the references in terms of automated metrics?
  - **RQ1.b (Human Evaluation Metrics):** How is the quality of the explanations generated by ZeroVar for variable understanding on human metrics?
- **RQ2 (Usefulness)**
  - **RQ2.a (Abbreviation Expansion):** How can the variable explanations generated by ZeroVar boost abbreviation expansion?
  - **RQ2.b (Spelling Correction):** How can the variable explanations generated by ZeroVar boost variable spelling correction?

All the data can be found in our replication package[5].

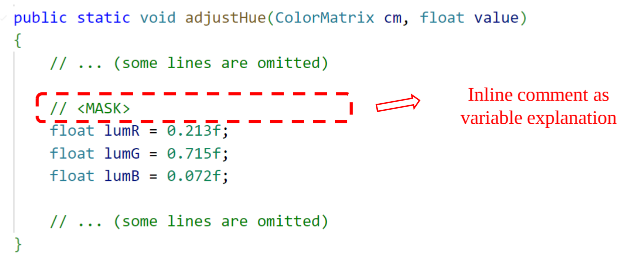
A. RQ1.a: Quality on Automated Metrics

We collect a reference dataset and investigate the textually similarity between the generated explanations and the references on automated metrics.

1) Dataset: We collect inline comments from the test set of Java language in CodeSearchNet to build a *reference dataset* for variable explanations. For each method in the test set, we use javalang [28] to convert it to an abstrct syntax tree (AST) and extract all variable declarations in it. For each variable declaration, we check whether the first previous non-empty line is a inline comment. If it is, we remove the inline comment from the method and produce a data instance consisting of

[4] https://huggingface.co/microsoft/unixcoder-base

[5] https://anonymous.4open.science/r/VarExp-481D/README.md



```
public static void adjustHue(ColorMatrix cm, float value)
{
    // ... (some lines are omitted)

    // <MASK>
    float lumR = 0.213f;
    float lumG = 0.715f;
    float lumB = 0.072f;

    // ... (some lines are omitted)
}
```

Figure 7. Constructed Prompt for Baseline Approach

the method, the variable name, and the inline comment. After processing all the methods, we obtained 3,565 such instances. Furthermore, we automatically and manually filter out the instances that meet one of the following criteria:

Automatic filtering includes three criteria: (i) the comment contains non-English text; (ii) the length of the comment exceeds 10 words, which may suggest that it explains code blocks rather than individual variables; (iii) after stopword removal, the number of words in the comment is equal to or less than the number of tokens in the variable name (i.e., non-informative comment).

Manual filtering involves assessing the relevance of comments to the variable meaning based on the code context. The comments that are confirmed to be relevant to the variable meaning are retained. In addition, if a comment can be converted to a variable explanation through minor modifications, such as removing starting verbs, it is modified and retained. This process is conducted through discussions between two of the authors.

Through the filtering, we obtain the reference dataset containing 773 instances, where the method and variable are the input, and the comment serves as the reference explanation.

2) *Baseline:* As demonstrated, we can acquire variable explanations to a certain extent from inline comments that come before variable declarations. To this end, we introduce a baseline approach based on prompt learning, which involves adding an *inline-comment-like prompt* "// <MASK>" before the declaration statement of the variable that requires an explanation. For instance, in Figure 1(a), we add the line "// <MASK>" before "float lumR = 0.213f;" to create a prompt for the variable *lumR*. We then leverage CodeT5 for generating the variable explanations from the prompt. We choose CodeT5 for the baseline because it is pre-trained on the span masking objective and used as the base model in ZEROVAR.

3) *Metrics:* BLEU (Bilingual Evaluation Understudy) [16] and ROUGE (Recall-Oriented Understudy for Gisting Evaluation) [17] are two popular evaluation metrics for text generation tasks.

BLEU is a metric commonly used to evaluate the quality of machine-generated text by comparing it to one or more reference texts. It measures the similarity between the generated text and the reference texts by computing the N-gram (continuous n words) overlap between them. BLEU considers *precision* (how many N-grams in the generated text are in the reference text) and brevity (how much shorter or longer the generated text is compared to the reference text). The resulting score ranges from 0 to 1, where higher scores indicate better quality.

Table I
SIMILARITY BETWEEN GENERATED AND REFERENCE EXPLANATIONS

	BLEU-1	BLEU-2	BLEU-3	BLEU-4	ROUGE-1	ROUGE-2	ROUGE-L
Baseline	0.317	0.216	0.162	0.122	0.306	0.120	0.298
ZEROVAR	0.351	0.253	0.186	0.126	0.445	0.240	0.430

ROUGE is another metric used to evaluate the quality of text generation systems. It also measures the similarity between the generated text and the reference texts by computing the overlap between their N-grams. ROUGE considers *recall* (how many N-grams in the reference text are in the generated text). The resulting score ranges from 0 to 1, where higher scores indicate better quality.

Both BLEU and ROUGE have several variations, such as BLEU-1, BLEU-2, BLEU-3, and BLEU-4, which represent the N-gram order used in the computation. Similarly, ROUGE-1, ROUGE-2, and ROUGE-L are the most commonly used variations of the metric. We tokenize the generated explanations and reference explanations into words and then compute the BLEU and ROUGE metrics following the descriptions in [16] and [17].

4) *Evaluation Procedure:* For each instance (md, var, ref) in the reference dataset, we use the method md and the variable var to create two prompts, $pmpt^{\text{ZEROVAR}}$ and $pmpt^{baseline}$, for ZEROVAR and the baseline, respectively. Next, we input $pmpt^{\text{ZEROVAR}}$ into ZEROVAR to generate a variable explanation, exp^{ZEROVAR}, while the baseline takes $pmpt^{baseline}$ to generate its own variable explanation, $exp^{baseline}$. Finally, we calculate the BLEU and ROUGE metrics based on ref, exp^{ZEROVAR}, and $exp^{baseline}$.

5) *Results:* Table I presents the evaluation results of ZEROVAR and the baseline. The results show that ZEROVAR outperforms the baseline on all BLEU (**+3.3%**∼**21.0%**) and ROUGE (**+44.3%**∼**100.0%**) metrics . It is worth noting that BLEU and ROUGE are precision- and recall-oriented metrics, respectively. Therefore, ZEROVAR produces variable explanations that have better N-gram overlapping with the reference explanations while introducing less noise. In particular, ZEROVAR achieves a significant advantage on ROUGE, indicating that the explanations generated by ZEROVAR provide a more comprehensive explanation of the variable meanings.

ZEROVAR and the baseline are both based on prompt learning, but differ in the way the prompt is constructed. ZEROVAR achieves better results because we formalize the variable explanation problem as parameter description generation task, which naturally aligns the downstream problem and pre-training objective in both *form* and *purpose*. Although the baseline also achieves *formal alignment* between the downstream problem and pre-training objective (i.e., span masking) through inline-comment prompts, the primary purpose of most inline comments is not to explain variables, resulting in a *significant gap* between the downstream problem and pre-training objective.

Finding 1: Our proposed approach, ZeroVar, is capable of generating variable explanations that are more similar to the reference explanations than the baseline, as evidenced by the improvements on all BLEU and ROUGE metrics. Moreover, the prompt constructed by ZeroVar facilitates better knowledge transfer between the downstream problem and pre-training objective.

B. RQ1.b: Quality on Human Metrics

In fact, the reference variable explanations we collect through inline comments may have quality issues. Although we ensure the final reference explanations are relevant to the variables through manual filtering, the original purpose of inline comments is not to explain variables, so they may not fully reflect the meaning of the variables. Therefore, we further evaluate the quality of the generated explanations through an emprical evaluation on some human metrics.

1) Dataset: We randomly sample 30 instances from the reference dataset and take the explanations generated by ZeroVar and the baseline for evaluation.

2) Participants: We invite 10 developers who have more than 3 years of Java programming experience to participate in this evaluation. These developers have not been involved in this work before and thus have no potential conflicts of interest.

3) Metrics: We consider the following metrics for the evaluation, assessed based on a 4-point Likert scale [29] (1-disagree; 2-somewhat disagree; 3-somewhat agree; 4-agree), similar to previous works [30], [31], [32], [33], [34]:

- **Correctness**: The explanation correctly explains the variable.
- **Completeness**: The explanation contains all the necessary information for explaining the variable.
- **Conciseness**: The explanation contains no unnecessary or redundant information for explaining the variable.

4) Evaluation Procedure: For each sampled instance, we mix the reference, the explanation generated by ZeroVar, and the explanation generated by the baseline. The participants are asked to rate the three versions of explanations in terms of correctness, completeness, and conciseness on the 4-point Likert scale according to the metric statements. Note that in order to reduce bias, the second statement (i.e., the statement for completeness) are phrased negatively to maintain the interpretation of the answers similar to all three statements. After the participants finish the evaluation, we ask them to explain their low ratings (i.e., 1 or 2).

5) Results: The results of the ratings for the different explanations are shown in Table II and Figure 8. For *correctness* of ZeroVar, **64.7**% of the answers are 4 (agree), **32.0**% are 3 (somewhat agree), **3.0**% are 2 (somewhat disagree), and **0.3**% are 1 (disagree) answers. For *completeness* of ZeroVar, **63.0**% of the answers are 4 (agree), **32.0**% are 3 (somewhat agree), **4.7**% are 2 (somewhat disagree), and **0.3**% are 1 (disagree). For *conciseness* of ZeroVar, **35.7**% of the answers are 4 (agree), **45.7**% are 3 (somewhat agree), **16.0**% are 2 (somewhat disagree), and **2.6**% are 1 (disagree) answers.

Table II
Ratings of Correctness, Completeness, and Conciseness

	Correctness				Completeness				Conciseness			
	1	2	3	4	1	2	3	4	1	2	3	4
Baseline	38	130	90	42	40	147	83	30	34	111	90	65
Reference	7	41	131	121	18	105	128	49	2	33	119	146
ZeroVar	1	9	96	194	1	14	96	189	8	48	137	107

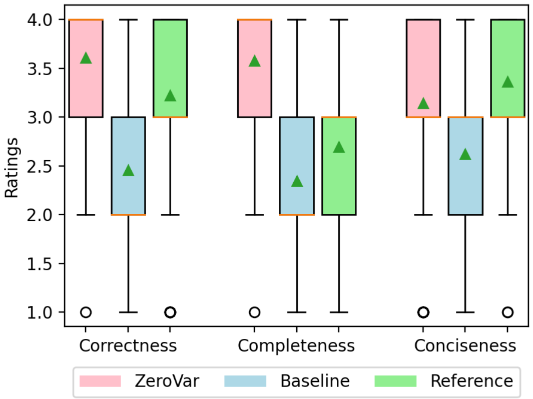


Figure 8. Ratings of Correctness, Completeness, and Conciseness

The ratings for the explanations generated by ZeroVar are much better than the baseline on all three metrics. To verify the statistical significance of the difference between the participants' ratings on the explanations generated by ZeroVar and the baseline, we use a two-sided independent T-test [35]. The null hypothesis is that the ratings of correctness, completeness, and conciseness for the two independent groups have identical average (expected) values. The results indicate that for each metric, the statistical difference is significant ($p << 0.01$), leading to the rejection of the null hypothesis. Compared to the reference, the ratings for the explanations generated by ZeroVar are much better on correctness and completeness and worse on conciseness.

The results suggest that ZeroVar generates more accurate and comprehensive explanations for variables, indicating that developers can obtain a better understanding of the variable meanings by reading the explanations generated by ZeroVar than the inline comments. This also confirms that there is a quality problem in the variable explanations collected using inline comments, which can only be used as *references* rather than *ground truth*. Furthermore, the results also highlight the difficulty of obtaining high-quality variable explanations as training data, emphasizing the necessity and value of zero-shot prompt learning. For conciseness, we analyze the feedback of participants and find that the main reason for the lower conciseness ratings than references is that the explanations generated by ZeroVar are generally longer and unnecessary for some simple variables.

Finding 2: Our approach, ZeroVar, has demonstrated its capability to generate variable explanations that are correct, complete, and concise. These explanations can effectively support developers in gaining an accurate and comprehensive understanding of variable meanings.

C. RQ2.a: Usefulness for Abbreviation Expansion

Most abbreviation expansion approaches [36], [37], [38], [39], [40], [41], [42], [43], [18] can generally be divided into two steps: (i) identifying candidate expansions for abbreviations by exploiting certain *contexts*, such as words in code comments; and (ii) ranking the candidates based on some strategies, such as leveraging learning models. Thus, the coverage of the exploited contexts (i.e., the possibility of including the correct expansion) is a crucial factor that affects the performance of abbreviation expansion. In this evaluation, we aim to explore the effectiveness of utilizing the explanations generated by our approach to enrich the exploited contexts and improve the performance of KgExpander [18], which is a state-of-the-art approach for abbreviation expansion.

1) Dataset: We utilize the dataset created by Jiang et al. [18] to evaluate KgExpander, consisting of 9 open-source projects from diverse application domains, namely, DB-Manager, Batik, Portecle, PDFsam, Retrofit, Bootique, CheckStyle, Maven, and FileBot. The dataset comprises 200 abbreviations sampled from identifiers in each project, which are manually expanded to their corresponding expansions. We only considered abbreviations that originate from variables and exclude those from other identifiers, such as method names, resulting in an dataset of 868 abbreviations and their corresponding expansions.

2) Context Enrichment: For each abbreviation in the dataset, we locate the variable it originates from and retrieve the enclosed method code. Then, we utilize ZeroVar to generate an explanation for the variable and insert it as an inline comment before the declaration statement of the variable. The generated explanation is then used as additional exploited context by KgExpander during the abbreviation expansion process, as KgExpander will consider the inline comments when identifying candidate expansions for abbreviations from variables [18].

3) Evaluation Procedure: We conduct two separate runs of KgExpander on the original projects and the enriched projects, respectively. Note that, to better focus on the effects of the exploited contexts, we remove the two abbreviation dictionaries that KgExpander uses for its original evaluation. The evaluation is conducted on two metrics to assess the usefulness of the generated explanations for enriching the exploited contexts. The first metric is the *present rate*, which measures the proportion of correct expansions that are present in the exploited contexts. This metric is an indicator of the coverage of the exploited contexts. The second metric is *precision* and *recall*, which measure the accuracy of the expanded abbreviations and the proportion of correct expansions generated by KgExpander, respectively. These two metrics provide a comprehensive evaluation of the overall expansion performance.

4) Results: Table III presents the evaluation results. As shown in the table, the expansion performance is improved in all the metrics after the enrichment. Specifically, the present rate, precision, and recall gain an improvement of **13.1%**, **3.6%**, and **10.0%**, respectively.

Table III
Comparison Results of Abbreviation Expansion

	Present Rate %	**Precision %**	**Recall %**
Before Enrichment	66.1%	45.0%	40.2%
After Enrichment	74.9%	46.6%	44.2%

The improved present rate suggests that the explanations generated by ZeroVar can provide correct expansions of abbreviations that are missing in the original exploited contexts. One of the benefits of this is that KgExpander can now provide correct expansions for the abbreviations that were previously unexpandable or incorrectly expanded, resulting in improved precision and recall.

Finding 3: The usefulness of the explanations generated by ZeroVar in enhancing the performance of abbreviation expansion is confirmed. The enriched context by these explanations enables KgExpander, a SOTA abbreviation expansion approach, to achieve higher present rate, precision, and recall.

D. RQ2.b: Usefulness for Variable Spelling Correction

To investigate the usefulness of the generated explanations for variable spelling correction, we construct a variable misspelling dataset similar to previous work by Chen et al. [19] and develop a lightweight spelling correction approach.

1) Dataset: Similar to Chen et al., we perturb variables to generate the *misspelling dataset*. Specifically, for each Java method in the CodeSearchNet test set, we use javalang [28] to parse it into an AST and extract variable names by looking up variable declaration nodes. In this way, we collect a set of 18,093 different variable names. For each collected variable name, we then find a method that contains the variable in the CodeSearchNet test set. Note that we do not directly use the variable name set adopted by Chen et al. in their spelling correction experiment because those variable names were collected from JavaScript code, which is difficult to find their code contexts in CodeSearchNet. There is no bias due to language difference here because Chen et al. trained their model using variable names extracted from C# code, which are also different from the variable names used in their spelling correction experiment. Next, we sample 1,023 (the same to Chen et al.) variable names from the collected variable name set and use nlpaug [44] tool to perturb the words in a similar way (i.e., simulating keyboard distance errors) as Chen et al. to generate misspelling instances. During this process, we replace all occurrences of these variables in the corresponding method code with the perturbed variable names and ensure that there are no correct corrections in other parts of the same code contexts (e.g., other identifier names).

2) Our Correction Approach: For a given perturbed variable, we utilize ZeroVar to generate three explanations, and use the words contained in these explanations as a spelling set, denoted as S. We then split the perturbed variable name into a word sequence $V = [w_1, w_2, ..., w_N]$ using camel case, and attempt to replace at least one word in V with a word from the spelling set S. After such replacement, we obtain a candidate correction $V' = [w'_1, w'_2, ..., w'_N]$ for the variable. Since both the variable and the spelling set contain multiple words, there are multiple possible replacement combinations resulting in different candidate corrections. Thus, we need to select the optimal one. To achieve this, we design a method based on edit distance to calculate the replacement cost for each candidate correction V'. Specifically, we first measure the cost of replacing a word $w_i \in V$ with a word $w'_i \in V'$ using the *normalized* Damerau-Levenshtein distance, denoted as $DL(w_i, w'_i)$. The Damerau-Levenshtein distance is the minimum number of operations required to change w_i into w'_i, which may include insertions, deletions, substitutions of a single character, or transposition of two adjacent characters. The *normalized* $DL(w_i, w'_i)$ (denoted as $normalized\text{-}DL(w_i, w'_i)$) is calculated by dividing $DL(w_i, w'_i)$ by the maximum length of w_i and w'_i, represented as $max(|w_i|, |w'_i|)$, where $|w_i|$ and $|w'_i|$ denote the lengths of w_i and w'_i, respectively. The overall cost of the candidate correction V' is the sum of the replacement costs for all the words in V, expressed as follows:

$$cost(V \rightarrow V') = \sum_{i=1}^{N} normalized\text{-}DL(w_i, w'_i)$$

Finally, we can select the top K candidate corrections with the lowest costs and convert them into camel-case variable names, which become the final corrections for the perturbed variable.

3) Baseline: We adopt VarCLR, a model proposed by Chen et al., as our baseline approach. VarCLR is an embedding model that is trained on variable renaming histories to measure the similarity between variables. To apply VarCLR on the spelling correction task, a set of candidate corrections needs to be chosen and their embedding vectors are generated. For a given perturbed variable, VarCLR generates an embedding vector and calculates the cosine similarity between it and all the candidates based on their embedding vectors. The top K candidate with the highest similarity scores are selected as the final corrections for the perturbed variable. In our evaluation, we use a set of 18,093 variable names that we collected as the candidate corrections for VarCLR. This enables us to ensure that the expected corrections are incorporated for the VarCLR approach.

4) Evaluation Procedure: we apply both our correction approach and the baseline approach to each instance in the misspelling dataset to obtain their respective corrections for the perturbed variable. Following Chen et al., we employ the metrics $hit@1$ and $hit@3$ to measure the correction accuracy. Specifically, $hit@1$ indicates whether the expected correction appears as the top one candidate, while $hit@3$ indicates whether it appears within the top three candidates.

5) Results: Table IV displays the evaluation results for our proposed lightweight correction approach and the baseline approach. The results demonstrate that our approach outperforms the baseline by a significant margin, achieving a **162.9%** improvement on $hit@1$ and a **49.6%** improvement on $hit@3$. This highlights the effectiveness of our approach in accurately correcting misspelled variable names.

Table IV
COMPARISON RESULTS OF VARIABLE SPELLING CORRECTION

	hit@1 %	**hit@3 %**
Baseline (VarCLR)	6.2%	11.7%
Our Correction Approach	16.3%	17.5%

In fact, both our correction approach and the baseline approach rely on selecting candidate corrections based on some similarity/distance measures. However, our approach achieves higher accuracy for two main reasons. First, the variable explanations generated by ZeroVar include the expected corrections, despite the perturbations in the variables. This suggests that ZeroVar can filter out the noise caused by the misspellings and reveal the true meaning of the variables. Second, we select the final correction from a much smaller candidate set (constructed from generated explanations) than the baseline approach. Although the larger candidate set used by the baseline ensures the inclusion of the expected corrections, it makes it difficult for the baseline approach to distinguish between similar candidates, whereas our approach can more accurately identify the best candidate correction.

> **Finding 4:** The usefulness of the explanations generated by ZeroVar in improving variable spelling correction is confirmed by our evaluation. Our lightweight correction approach, which uses the explanations as candidate corrections, achieves better accuracy on both $hit@1$ and $hit@3$ metrics compared to the baseline.

V. Discussion

A. Prompt Engineering

In this paper, we employ zero-shot prompt learning to generate variable explanations by leveraging pre-trained models. The effectiveness of prompt learning, in fact, heavily relies on the quality of the designed prompts, especially for few-shot and zero-shot scenarios. For the same problem, different modeling approaches can be aligned to different pre-training objectives and be designed with different prompt templates. This is known as prompt engineering.

If a large-scale pre-trained model is used, such as GPT-3 (175 billion model parameters), the problem is typically transformed into an autoregressive text generation form by continuing to generate subsequent content from left to right based on the previous text. This is because these large-scale models are mostly decoder-only architectures trained through autoregression. However, for small-scale models like CodeT5 (220 million model parameters), problem modeling is crucial, and different problem modeling methods will require different prompt templates and produce different results. For instance, in our variable explanation problem, our approach ZeroVar and the baseline approach in RQ1.a respectively model the problem as a parameter explanation generation task and an inline comment generation task. Both modeling approaches can align the problem *in form* to the span masking objective of CodeT5 pre-training stage. However, there are significant differences in the prompt templates required by the two approaches, and the quality of the generated explanations also differs greatly from the results of RQ1.a and RQ1.b. The parameter explanation generation task is *more natural in purpose* for variable explanation problem than the inline comment generation task, and it can better utilize the knowledge learned by the pre-trained model during pre-training (see Section IV-A5).

B. More Applications

We evaluate the quality of the explanations generated by our app, and the experimental results demonstrate that the quality of the generated explanations is relatively high. Such high-quality explanations are helpful for developers to gain an accurate and comprehensive understanding of variable meanings. In addition to abbreviation expansion and spelling correction, the generated variable explanations may also be used in more other applications. Some possible application scenarios are presented below.

For code search, the generated explanations can serve as additional information to enhance the context of the code and bridge the lexical gap between the search query and the code [45], thereby improving accuracy. For code comprehension, the generated explanations can help link the terms in variables to corresponding concepts, thereby helping developers better understand and reuse code [46]. For code review, the generated explanations can help reviewers better understand the code and identify any potential issues. For mining approaches [47], [48] rely on code identifiers, the the generated explanations can boost these approach by providing more meaningful contexts.

C. Threats to Validity

The threats to the internal validity of our studies lie in the randomness of data sampling and the subjectiveness in data annotation. To mitigate these threats, we follow commonly-used data sampling strategy and mix explanations from different sources, and involve multiple annotators to minimize preference bias. The threats to the external validity lies in the benchmarks used by our work, which cannot guarantee the generality of our findings. To minimize such threats, we leverage a large scale of code corpus and include two different application scenarios for evaluation. We believe it is interesting future work to extend ZEROVAR to other programming languages and incorporating ZEROVAR with more variable relevant downstream applications.

VI. RELATED WORK

A. Code Explanation Generation

Code explanation generation has been approached from various angles, leading to a number of techniques in the field.

Code Summarization. Deep learning models have been employed to improve the performance of code summarization [2], [3], [4], [5], [6], [7] based on code contexts. These approaches generate method-level summaries for code. However, generating fine-grained variable explanations presents a new challenge, as such explanations do not exist in the wild. Therefore, it is infeasible to train a model based on an existing corpus of code and its relevant variable explanations. In our work, we address this challenge by constructing prompts with the target variables in the format of docstring and enabling zero-shot learning with pre-trained language models.

Parameter Description. Some research have investigated the issues about parameter descriptions in code summarization [49] and proposed parameter description generation approaches [50], [51]. These approaches generate descriptions for method formal parameters based on heuristic rules or learning models, but they are not designed for variables. In this work, we consider variables as special parameters and leverage code pre-trained models and prompt learning to generate high-quality variable explanations.

Code Augmentation. Previous research has attempted to augment code by leveraging external resources such as Stack Overflow discussions relevant to a given API or code snippet [52], [53], [54] or extracting concept explanations from Wikipedia using identifiers [46]. Our work provides fine-grained explanations for variables based solely on code contexts without relying on external resources. Furthermore, our approach can potentially enhance these previous studies by providing more comprehensive and accurate meanings of variables.

B. Prompt Learning

Prompt learning is a recent and promising approach that has been applied to various natural language processing tasks.

Pre-trained Language Models as Knowledge Bases. Recent studies have demonstrated that pre-trained language models (PLMs) can be utilized as knowledge bases by using synthetic tasks similar to the pre-training objective to retrieve the knowledge/information stored in the models [13], [55]. These works have shown that language models can recall factual knowledge without any fine-tuning by using proper prompts. In our work, we also consider code PLMs as knowledge bases that contain explanations of parameters and variables. We leverage zero-shot prompt learning to retrieve the knowledge necessary for explaining variable meanings.

Prompt Engineering. Several works have focused on exploring effective prompt templates to improve performance of PLMs on downstream problems. In a recent survey by Liu et al. [56], prompt templates are broadly classified into two categories: *cloze prompts* [57], [13], which entail filling in the blanks in text or code, and *prefix prompts* [58], [59], which continue generating content following a specified prefix. In our study, we consider the unique characteristics of our problem and devise a *cloze* prompt template to fill in missing descriptions in method docstrings. This prompt template naturally aligns the variable explanation problem with the pre-training objective of description masking.

Prompt Learning in Software Engineering. Recent studies have explored the application of prompt learning in various software engineering tasks. Wang et al. [60] investigated the effectiveness of prompt learning in code intelligence tasks such as clone detection and code summarization. Huang et al. [61] used prompt learning for type inference. In contrast, our work introduces a novel prompt learning approach for generating variable explanations, which is a crucial problem in software engineering.

VII. Conclusions and Future Work

In this paper, we introduce a new approach for generating variable explanations called ZeroVar, which utilizes code pre-trained models and zero-shot prompt learning. The approach consists of two stages: (i) a continual pre-training stage where a base model, CodeT5, is continually pre-trained using the description masking objective, and (ii) a zero-shot prompt learning stage where a prompt is created based on the given method and variable, and the continually pre-trained model is used to generate the variable explanations. To evaluate the effectiveness of ZeroVar, we collected a dataset containing 773 reference variable explanations. Our evaluation results demonstrate that ZeroVar generates higher quality explanations compared to the baseline approach, not only based on automated metrics such as BLEU and ROUGE, but also based on human metrics such as correctness, completeness, and conciseness. Moreover, we conducted two experiments to examine the usefulness of the generated explanations in improving variable naming quality, specifically in abbreviation expansion and spelling correction. Our experimental results confirm the usefulness of ZeroVar in enhancing these two applications. Future research may explore the potential of ZeroVar in generating variable explanations for other programming languages and incorporating it with more downstream applications.

Acknowledgment

This work was supported by the National Natural Science Foundation of China under Grant 61972098.

References

[1] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Software Quality Journal*, vol. 14, pp. 261–282, 2006.

[2] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in *Proceedings of the 28th International Conference on Program Comprehension, ICPC 2020, Seoul, Republic of Korea, July 13-15, 2020*. ACM, 2020, pp. 184–195.

[3] Q. Chen and M. Zhou, "A neural framework for retrieval and summarization of source code," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. ACM, 2018, pp. 826–831.

[4] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. ACM, 2018, pp. 397–407.

[5] W. U. Ahmad, S. Chakraborty, B. Ray, and K. Chang, "A transformer-based approach for source code summarization," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*. Association for Computational Linguistics, 2020, pp. 4998–5007.

[6] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, "Code generation as a dual task of code summarization," in *Proceedings of Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, Vancouver, BC, Canada, December 8-14, 2019*, 2019, pp. 6559–6569.

[7] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, ser. JMLR Workshop and Conference Proceedings, vol. 48. JMLR.org, 2016, pp. 2091–2100.

[8] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy, "Using information retrieval based coupling measures for impact analysis," *Empirical software engineering*, vol. 14, pp. 5–32, 2009.

[9] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Communications of the ACM*, vol. 59, no. 5, pp. 122–131, 2016.

[10] M. J. Kaelbling, "Programming languages should not have comment statements," *ACM SIGPlan Notices*, vol. 23, no. 10, pp. 59–60, 1988.

[11] B. Fluri, M. Würsch, E. Giger, and H. C. Gall, "Analyzing the co-evolution of comments and source code," *Software Quality Journal*, vol. 17, pp. 367–394, 2009.

[12] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[13] F. Petroni, T. Rocktäschel, S. Riedel, P. S. H. Lewis, A. Bakhtin, Y. Wu, and A. H. Miller, "Language models as knowledge bases?" in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*. Association for Computational Linguistics, 2019, pp. 2463–2473.

[14] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*. Association for Computational Linguistics, 2021, pp. 8696–8708.

[15] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*. Association for Computational Linguistics, 2022, pp. 7212–7225.

[16] K. Papineni, S. Roukos, T. Ward, and W. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*. ACL, 2002, pp. 311–318.

[17] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Text summarization branches out*, 2004, pp. 74–81.

[18] Y. Jiang, H. Liu, and L. Zhang, "Semantic relation based expansion of abbreviations," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. ACM, 2019, pp. 131–141.

[19] Q. Chen, J. Lacomis, E. J. Schwartz, G. Neubig, B. Vasilescu, and C. L. Goues, "Varclr: Variable semantic representation pre-training via contrastive learning," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2327–2339.

[20] 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," *J. Mach. Learn. Res.*, vol. 21, pp. 140:1–140:67, 2020.

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, 2017, pp. 5998–6008.

[22] 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, Online Event, 16-20 November 2020*, ser. Findings of ACL, vol. EMNLP 2020. Association for Computational Linguistics, 2020, pp. 1536–1547.

[23] H. Husain, H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *CoRR*, vol. abs/1909.09436, 2019.

[24] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[25] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proceedings of 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[26] A. Graves, "Sequence transduction with recurrent neural networks," *arXiv preprint arXiv:1211.3711*, 2012.

[27] N. Boulanger-Lewandowski, Y. Bengio, and P. Vincent, "Audio chord recognition with recurrent neural networks." in *ISMIR*. Curitiba, 2013, pp. 335–340.

[28] (2023) javalang. [Online]. Available: https://github.com/c2nes/javalang

[29] R. Likert, "A technique for the measurement of attitudes." *Archives of psychology*, 1932.

[30] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and V. Shanker, "Automatic Generation of Natural Language Summaries for Java Classes," in *21st IEEE International Conference on Program Comprehension (ICPC'13)*. San Francisco, USA: IEEE, 2013, pp. 23–32.

[31] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards Automatically Generating Summary Comments for Java Methods," in *25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10)*, Antwerp, Belgium, 2010, pp. 43–52.

[32] M. Liu, X. Peng, A. Marcus, Z. Xing, W. Xie, S. Xing, and Y. Liu, "Generating query-specific class API summaries," in *Proceedings of 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, August 26-30, 2019, Tallinn, Estonia*, 2019, pp. 120–130.

[33] Y. Liu, M. Liu, X. Peng, C. Treude, Z. Xing, and X. Zhang, "Generating concept based api element comparison using a knowledge graph," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 834–845.

[34] M. Liu, X. Peng, A. Marcus, C. Treude, J. Xie, H. Xu, and Y. Yang, "How to formulate specific how-to questions in software development?" in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 306–318.

[35] A. Ross and V. L. Willson, "Two-sided independent t-test," in *Basic and advanced statistical tests*. Springer, 2017, pp. 9–12.

[36] E. Adar, "Sarad: a simple and robust abbreviation dictionary," *Bioinform.*, vol. 20, no. 4, pp. 527–533, 2004.

[37] B. Caprile and P. Tonella, "Restructuring program identifier names," in *Proceedings of the 2000 International Conference on Software Maintenance, ICSM 2000, San Jose, California, USA, October 11-14, 2000*. IEEE Computer Society, 2000, pp. 97–107.

[38] N. Madani, L. Guerrouj, M. D. Penta, Y. Guéhéneuc, and G. Antoniol, "Recognizing words from source code identifiers using speech recognition techniques," in *Proceedings of the 14th European Conference on Software Maintenance and Reengineering, CSMR 2010, Madrid, Spain, March 15-18, 2010*. IEEE Computer Society, 2010, pp. 68–77.

[39] D. J. Lawrie, D. W. Binkley, and C. Morrell, "Normalizing source code vocabulary," in *Proceedings of the 17th Working Conference on Reverse Engineering, WCRE 2010, Beverly, MA, USA, October 13-16, 2010*. IEEE Computer Society, 2010, pp. 3–12.

[40] E. Hill, Z. P. Fry, H. Boyd, G. Sridhara, Y. Novikova, L. L. Pollock, and K. Vijay-Shanker, "AMAP: automatically mining abbreviation expansions in programs to enhance software maintenance tools," in *Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR 2008 (Co-located with ICSE), Leipzig, Germany, May 10-11, 2008, Proceedings*. ACM, 2008, pp. 79–88.

[41] D. J. Lawrie, H. Feild, and D. W. Binkley, "Extracting meaning from abbreviated identifiers," in *Proceedings of the 7th IEEE International Workshop on Source Code Analysis and Manipulation, SCAM 2007, Paris, France, September 30 - October 1, 2007*. IEEE Computer Society, 2007, pp. 213–222.

[42] D. J. Lawrie and D. W. Binkley, "Expanding identifiers to normalize source code vocabulary," in *Proceedings of the 27th IEEE International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011*. IEEE Computer Society, 2011, pp. 113–122.

[43] A. Corazza, S. D. Martino, and V. Maggio, "LINSEN: an efficient approach to split identifiers and expand abbreviations," in *Proceedings of the 28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012*. IEEE Computer Society, 2012, pp. 233–242.

[44] (2023) nlpaug. [Online]. Available: https://github.com/makcedward/nlpaug

[45] C. Wang, X. Peng, Z. Xing, Y. Zhang, M. Liu, R. Luo, and X. Meng, "Xcos: Explainable code search based on query scoping and knowledge graph," *ACM Transactions on Software Engineering and Methodology*, 2023.

[46] C. Wang, X. Peng, Z. Xing, and X. Meng, "Beyond literal meaning: Uncover and explain implicit knowledge in code through wikipedia-based concept linking," *IEEE Transactions on Software Engineering*, pp. 1–15, 2023.

[47] C. Wang, X. Peng, M. Liu, Z. Xing, X. Bai, B. Xie, and T. Wang, "A learning-based approach for automatic construction of domain glossary from source code and documentation," in *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2019, pp. 97–108.

[48] C. Wang, Y. Lou, X. Peng, J. Liu, and B. Zou, "Mining resource-operation knowledge to support resource leak detection," in *Proceedings of the 2023 31st ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2023.

[49] Q. Chen, X. Xia, H. Hu, D. Lo, and S. Li, "Why my code summarization model does not work: Code comment improvement with category prediction," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, pp. 25:1–25:29, 2021.

[50] G. Sridhara, L. L. Pollock, and K. Vijay-Shanker, "Generating parameter comments and integrating with method summaries," in *Proceedings of The 19th IEEE International Conference on Program Comprehension, ICPC 2011, Kingston, ON, Canada, June 22-24, 2011*. IEEE Computer Society, 2011, pp. 71–80.

[51] Q. Chen, Z. Yang, Z. Liu, S. Li, and C. Gao, "Parameter description generation with the code parameter flow," in *Proceedings of 22nd IEEE International Conference on Software Quality, Reliability and Security, QRS 2022, Guangzhou, China, December 5-9, 2022*. IEEE, 2022, pp. 884–895.

[52] C. Vassallo, S. Panichella, M. D. Penta, and G. Canfora, "CODES: mining source code descriptions from developers discussions," in *Proceedings of the 22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014*. ACM, 2014, pp. 106–109.

[53] C. Treude and M. P. Robillard, "Augmenting API documentation with insights from stack overflow," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. ACM, 2016, pp. 392–403.

[54] M. M. Rahman, C. K. Roy, and I. Keivanloo, "Recommending insightful comments for source code using crowdsourced knowledge," in *Proceedings of the 15th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2015, Bremen, Germany, September 27-28, 2015*. IEEE Computer Society, 2015, pp. 81–90.

[55] Z. Jiang, F. F. Xu, J. Araki, and G. Neubig, "How can we know what language models know?" *Transactions of the Association for Computational Linguistics*, vol. 8, pp. 423–438, 2020.

[56] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Comput. Surv.*, vol. 55, no. 9, pp. 195:1–195:35, 2023.

[57] L. Cui, Y. Wu, J. Liu, S. Yang, and Y. Zhang, "Template-based named entity recognition using BART," in *Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, Online Event, August 1-6, 2021*, ser. Findings of ACL, vol. ACL/IJCNLP 2021. Association for Computational Linguistics, 2021, pp. 1835–1845.

[58] B. Lester, R. Al-Rfou, and N. Constant, "The power of scale for parameter-efficient prompt tuning," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*. Association for Computational Linguistics, 2021, pp. 3045–3059.

[59] X. L. Li and P. Liang, "Prefix-tuning: Optimizing continuous prompts for generation," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*. Association for Computational Linguistics, 2021, pp. 4582–4597.

[60] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu, “No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. ACM, 2022, pp. 382–394.

[61] Q. Huang, Z. Yuan, Z. Xing, X. Xu, L. Zhu, and Q. Lu, “Prompt-tuned code language model as a neural knowledge base for type inference in statically-typed partial code,” in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 2022, pp. 79:1–79:13.