PPM: Automated Generation of Diverse Programming Problems for Benchmarking Code Generation Models

SIMIN CHEN, UT Dallas, USA
XIAONING FENG, Taiyuan University of Technology, China
XIAOHONG HAN, Taiyuan University of Technology, China
CONG LIU, UC Riverside, USA
WEI YANG, UT Dallas, USA

In recent times, a plethora of Large Code Generation Models (LCGMs) have been proposed, showcasing significant potential in assisting developers with complex programming tasks. Within the surge of LCGM proposals, a critical aspect of code generation research involves effectively benchmarking the programming capabilities of models. Benchmarking LCGMs necessitates the creation of a set of diverse programming problems, and each problem comprises the prompt (including the task description), canonical solution, and test inputs. The existing methods for constructing such a problem set can be categorized into two main types: manual methods and perturbation-based methods. However, manual methods demand high effort and lack scalability, while also risking data integrity due to LCGMs' potentially contaminated data collection, and perturbation-based approaches mainly generate semantically homogeneous problems with the same canonical solutions and introduce typos that can be easily auto-corrected by IDE, making them ineffective and unrealistic. Addressing the aforementioned limitations presents several challenges: (1) How to automatically generate semantically diverse Canonical Solutions to enable comprehensive benchmarking on the models, (2) how to ensure long-term data integrity to prevent data contamination, and (3) how to generate natural and realistic programming problems. To tackle the first challenge, we draw key insights from viewing a program as a series of mappings from the input to the output domain. These mappings can be transformed, split, reordered, or merged to construct new programs. Based on this insight, we propose programming problem merging, where two existing programming problems are combined to create new ones. In addressing the second challenge, we incorporate randomness into our programming problem generation process. Our tool can probabilistically guarantee no data repetition across two random trials. To tackle the third challenge, we propose the concept of a Lambda Programming Problem, comprising a concise one-sentence task description in natural language accompanied by a corresponding program implementation. Our tool ensures the program prompt is grammatically correct. Additionally, the tool leverages return value type analysis to verify the correctness of newly created Canonical Solutions. In our empirical evaluation, we utilize our tool on two widely-used datasets and compare it against nine baseline methods using eight code generation models. The results demonstrate the effectiveness of our tool in generating more challenging, diverse, and natural programming problems, compared to the baselines.

CCS Concepts: • **Software and its engineering → Automatic programming**; • **Computing methodologies → Machine learning**; **Artificial intelligence**;

Additional Key Words and Phrases: datasets, neural networks, program synthesis, large language model

Authors' addresses: Simin Chen, simin.chen@UTDallas.edu, UT Dallas, Dallas, USA; XiaoNing Feng, fengxiaoning1746@link.tyut.edu.cn, Taiyuan University of Technology, Taiyuan, China; Xiaohong Han, hanxiaohong@tyut.edu.cn, Taiyuan University of Technology, Taiyuan, China; Cong Liu, congl@ucr.edu, UC Riverside, Riverside, USA; Wei Yang, wei.yang@utdallas.edu, UT Dallas, Dallas, USA.

ACM 2994-970X/2024/7-ART54
https://doi.org/10.1145/3643780

ACM Reference Format:
Simin Chen, XiaoNing Feng, Xiaohong Han, Cong Liu, and Wei Yang. 2024. PPM: Automated Generation of Diverse Programming Problems for Benchmarking Code Generation Models. *Proc. ACM Softw. Eng.* 1, FSE, Article 54 (July 2024), 22 pages. https://doi.org/10.1145/3643780

1 INTRODUCTION

Recently, large code generation models (LCGMs) have garnered significant attention due to their remarkable performance in tackling complex programming problems. For instance, a prime example of such advancement is OpenAI's CodeX [10], which can offer real-life help to software engineers and enhance their productivity.

Amidst the proposal of numerous LCGMs, a crucial aspect of code generation research revolves around effectively benchmarking the programming abilities of each model. One distinguishing characteristic sets benchmarking LCGMs apart when benchmarking their programming abilities, as opposed to benchmarking other machine learning models: To evaluate LCGMs, the model outputs (*i.e.,* the generated code snippets) must be executed with designated test inputs to observe its runtime behavior. In NLP, metrics like accuracy or BLEU score can quickly give an idea of the model's performance by comparing the output against a ground truth. However, in the context of code generation, such match-based metrics are insufficient for assessing the correctness of generated programs. Instead, determining correctness requires executing the generated program on designated test inputs and then conducting a functional equivalence comparison with a canonical solution. Due to such distinct characteristics, a benchmark should encompass both the problem description and the canonical solution, along with test inputs for evaluating correctness.

To construct a programming problem dataset suitable for benchmarking purposes, one common approach involves employing manually-based methods [10]. However, we have identified two primary limitations associated with these methods. ① Firstly, these methods necessitate a significant amount of human effort. This includes formulating precise programming task descriptions, creating accurate canonical solution implementations, and thoughtfully constructing comprehensive test inputs. These challenges result in a limited availability of evaluation data based on program execution. For instance, existing execution-based datasets like *HumanEval* [10] are notably scarce, featuring only a small number (a total of 164) of programming problems. ② Secondly, these manually-based methods tend to generate concrete programming problem sets. Such concrete problem sets face the ongoing challenge of *long-term data integrity*. In other words, once a programming problem dataset is published on the Internet for benchmarking purposes, future models may inadvertently utilize this dataset as part of their training data due to the extensive and sometimes indiscriminate collection of training data in LCGMs. This scenario may lead to unintended benchmark dataset leakage during the training of modern NLP models [30].

Another type of existing method belongs to perturbation-based methods [33]. These methods operate on the assumption that slight modifications to programming task descriptions should not alter the corresponding canonical solutions. Thus, they create programming problems by perturbing the programming task descriptions. However, this method, while addressing certain automation limitations, encounters two significant drawbacks: ① Firstly, the programming problems generated by this method lack semantic diversity. Benchmarking a model using a semantic-homogeneous dataset may not accurately showcase the model's ability to effectively navigate the solution space. This is because it may fail to uncover edge cases or specific types of errors. Additionally, such datasets often exhibit varying levels of complexity, aiding in assessing the model's proficiency in handling tasks ranging from simpler to more intricate ones. ② Secondly, such perturbation-based methods introduce unnatural and unrealistic alterations (e.g., typos, excessive blank lines) into

the programming task descriptions. These alterations can be easily detected by modern Integrated Development Environments (IDEs), potentially diminishing the reality of the generated problems.

We have identified three primary challenges in designing an automated method capable of effectively addressing the aforementioned limitations. The first challenge is automating the generation of canonical solutions based on programming problem descriptions. This challenge raises a dilemma regarding the potential obviation of the need for LCGMs if such a method exists that can autonomously create canonical solutions for any programming task. The second challenge relates to ensuring long-term data integrity while also prioritizing transparency and public accessibility in benchmarking LCGMs. Lastly, the third challenge involves generating programming problems that are natural and realistic.

Our Idea. To tackle the challenges mentioned earlier, we have the following ideas: ❶ To facilitate the automatic semantic programming problem generation, we make a crucial observation: A program's essence lies in its ability to map input from the domain to the corresponding output domain. Furthermore, we recognize the potential for using one program's output as another program's input. Building upon these insights, we introduce the concept of *Programming Problem Merging* (PPM), wherein we combine two pre-existing programming problems to create a new one. We employ this *Programming Problem Merging* concept as the foundation of our method to generate new programming problems. ❷ In the pursuit of ensuring long-term data integrity, our approach diverges from the creation of static programming problems. Instead, we propose injecting an element of randomness into our method. This entails defining an expansive random search space, allowing PPM to yield distinct programming problems with a high likelihood of avoiding repetition. While it's true that this element of randomness may introduce variability in benchmarking results, our contention is that repeated measurements can effectively mitigate this potential issue. ❸ To address the challenge of generating natural and realistic programming problems, we introduce a novel concept known as the *Lambda Programming Problem*. This *Lambda Programming Problem* comprises a concise one-sentence task description in natural language, accompanied by a corresponding program implementation. By incorporating the *Lambda Programming Problem*, PPM can guarantee the grammatical correctness of newly generated task descriptions because both the seed task description and the task description in our *Lambda Programming Problem* are grammatically correct. Furthermore, we conduct an analysis of the data type of a given seed programming problem and meticulously select an appropriate *Lambda Programming Problem*. Consequently, PPM is equipped to ensure not only the novelty of canonical solutions but also their syntactical correctness. Thus, the problems generated by PPM are more natural and realistic.

Implementation and Evaluation. We conducted comprehensive experiments to evaluate the effectiveness of PPM. Specifically, we applied PPM to two widely used real-world public datasets: *HumanEval* and *MBPP*. We benchmarked eight popular LCGMs that were trained on different corpora and featured diverse model architectures, sizes, and working mechanisms. To gauge the performance of PPM, we compared it against nine state-of-the-art methods that are specifically designed for benchmarking code generation models. Our evaluation results demonstrated that PPM outperforms these methods, proving its high efficacy in generating test inputs that effectively degrade computation efficiency.

Contributions. Our contributions are summarized as follows:

- We present the concept of *Programming Problem Merging* (PPM), a novel methodology designed to create programming problems with diverse semantics. These problems are particularly valuable for benchmarking large code generation models.

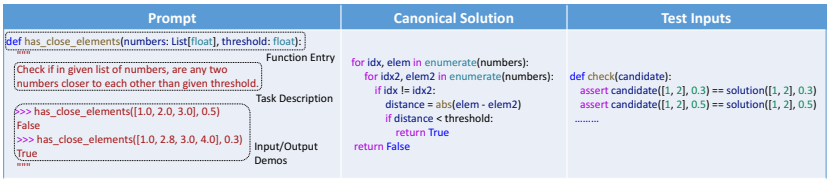

Fig. 1. Programming problem example.

- Based on the core principles of PPM, we offer two practical implementations that leverage lambda programming tasks: *type-aware value transformation* (PPM-T) and *pure value transformation* (PPM-V). These adaptations enable us to generate new semantic diverse programming problems.
- In our empirical evaluation of PPM, the results underscore its exceptional capacity to generate programming problems characterized by remarkable semantic diversity. This capability, in turn, sheds light on the inherent limitations of existing LCGMs, setting PPM apart from other methods that predominantly yield problems with homogeneous semantics. Furthermore, PPM excels in generating natural programming problems that maintain long-term data integrity, while providing stable benchmarking results.

2 BACKGROUND

2.1 Large Code Generation Models

Training of LCGMs. As the size of these models continues to grow, so does the training corpus they require. Due to the vastness of these training corpora for large code generation models, it becomes challenging to precisely identify the sources, for example, the training corpus of CodeX includes more than 54 million public software repositories [10], and in some cases, the training data of the model remains unpublic [28]. Once the training corpus is crawled from the internet, the large code generation models are trained using the objective function, such as predicting the next tokens or predicting random masked tokens within the corpus. This training objective enables these models to learn from the vast data and improve their code generation capabilities.

Evaluation of LCGMs. When comparing evaluation methods for machine learning models, assessing large code generation models reveals two unique characteristics. (1). In evaluating ML models for NLP, common metrics like accuracy or BLEU score are used. However, these metrics fall short of capturing functionally equivalent programs, making it essential to assess the semantics of generated code by executing it on test inputs and comparing outputs with a reference solution. (2). In contrast to traditional ML models that have predetermined training/testing datasets [11], LCGMs don't have such splits due to their extensive training on a substantial portion of GitHub. The GitHub code corpus often already encompasses solutions to problems from various sources [10]. For example, the newly introduced *APPS* dataset comprises over ten public repositories with solutions to Codeforces problems [10]. As a result, evaluating the programming capabilities of these models usually requires generating new custom programming problems.

Fig. 1 showcases a programming problem example from the manually crafted *HumanEval* dataset [10], which is a widely used dataset for evaluating LCGMs. As shown in this figure, each programming problem comprises three parts: the prompt, the canonical solution, and a set of test inputs. The prompt encompasses several crucial elements, namely a function definition that declares the entry point for the function, an extensive task description that provides detailed instructions on how to solve the problem, and illustrative input/output demonstrations. The canonical solution

Table 1. Feature support in existing work within the optimal design space. ●represents supported, ◐represents partial supported, ○represents not supported, and – represents not applicable.

Methods	Automation	Semantic Diversity	Long-Term Integrity	Naturalness	Representative Works
Hand-written	○	–	–	●	Human-Eval, MBPP
Description Perturbation	●	○	◐	◐	`Character Mutation`, `Token Mutation`
Sytanx Perturbation	●	○	◐	●	`Insert Blank Line`
Demo Perturbation	●	○	◐	●	`Add Demo`, `Replace Demo`
Ours	●	●	●	●	PPM

is a manually crafted code snippet, carefully constructed to serve as the ground truth solution to the problem presented in the prompt. The test inputs are designed to assess the correctness of programs generated by code generation models when solving the problem defined in the prompt.

2.2 Programming Problem Generation Methods

Manually-based Methods. This category [4, 5, 10] encompasses methods that involve human expertise in crafting programming problems from scratch. A noteworthy example is OpenAI's approach to assessing the code generation proficiency of Codex through the introduction of *HumanEval*, a comprehensive collection of 164 Python programming challenges, carefully designed and curated.

Perturbation-based Methods. The previously mentioned manually-based methods demand significant human effort for the creation of dedicated programming problems and do not delve into the realm of robustness evaluation. To overcome this limitation, a range of perturbation-based methods has emerged [23, 24, 33]. These methods operate on the fundamental assumption that making slight modifications to the prompts in programming problems should not alter the corresponding canonical solutions. One notable example of a perturbation-based method is ReCode [33], which incorporates the concept of *adversarial NLP*. It introduces a series of perturbations, such as token mutation, character mutation, and syntax mutation, to manipulate the prompts and assess the programming capabilities effectively.

Optimal Design Space & Limitation of Existing Methods. We have outlined four key features of an ideal method for creating programming problems, as summarized in Table 1. ❶ *Automation*: The method must be automated to eliminate the necessity for extensive manual dataset creation. Manual methods are labor-intensive and cannot produce scalable datasets. ❷ *Semantic Diversity*: The method should generate programming problems that exhibit semantic diversity. Failing to do so, by generating a set of semantically uniform problems, would result in an incomplete evaluation of the programming capabilities of LCGMs, potentially yielding falsely elevated results. ❸ *Long-Term Integrity*: The method should include mechanisms to resist data leakage, especially for maintaining long-term data integrity. Given LCGM's reliance on internet-sourced training data, pre-existing manually crafted datasets could unintentionally become part of future LCGM models, rendering them inaccessible for benchmarking purposes. ❹ *Naturalness*: The method should generate natural and realistic programming problems. Modern Integrated Development Environments (IDEs) excel at detecting typos and grammatical errors, rendering grammar-incorrect programming problems unrealistic and impractical. Regrettably, as indicated in Table 1, existing methods do not comprehensively support these four essential features. Hence, there is a compelling need to design a new method that addresses these limitations.

3 CHALLENGES & HIGH-LEVEL SOLUTIONS

Challenge 1: Automated Generation of Semantic Diverse Problems. Generating programming problems with diverse semantics automatically poses a significant challenge. This challenge arises

from the need for unique canonical solutions. Automatically generating correct canonical solutions for any programming problem presents a considerable challenge. The reason is that if we already have an automated method capable of creating canonical solutions, the existing LCGM models would be redundant. Such an automated method could then be used to directly assist developers in generating programs, eliminating the need for developing different LCGMs.

Solution 1: To tackle this challenge and create programming problems that exhibit semantic diversity, we introduce a novel idea, *Programming Problem Merging*. wherein we combine two pre-existing programming problems to create a new one. The newly created programming problem will possess distinct semantics compared to the original two problems. Additionally, we can leverage the canonical solutions from the two problems to automatically derive the correct canonical solution for the merged problem.

Challenge 2: Dilemma of Long-Term Integrity and Public Available. Any specific, concrete programming problems have inherent limitations of long-term integrity once they are published on the Internet due to the specific training data collection mechanism in LCGMs (§2.1). However, benchmarking LCGMs necessitates the benchmarks being transparent and publicly accessible—a requirement that seemingly contradicts the concept of long-term integrity.

Solution 2: In response to this challenge, instead of generating specific, concrete programming problems, we introduce a methodology that is attuned to randomness, allowing us to generate diverse programming problems. The search space in our random parameter search is substantial, minimizing the likelihood of producing identical programming problems. By adopting this randomness-aware idea, our method ensures Long-term integrity. In the event that a particular set of concrete programming problems becomes exposed, we can confidently generate non-repeating, randomized problems for evaluation, significantly reducing the risk of data compromise.

Challenge 3: Natural and Realistic Problems. The final challenge pertains to generating natural and realistic programming problems, which require grammatical correctness in the programming task description. However, directly combining two existing complex programming task descriptions often results in grammar errors and makes the new problem unnatural.

Solution 3: To overcome this challenge, we introduce the concept of a *lambda programming task*. This involves a concise one-sentence description aimed at manipulating the output of an existing programming task description. Since both the original programming task description and our proposed *lambda programming task* are grammatically correct, we can guarantee the correctness of the newly generated programming problem.

4 APPROACH

4.1 Problem Formulation of Programming Problem Merging

In formal terms, let us examine a seed programming problem, represented as $\mathcal{P} =< P, S, T >$, where: (1) input prompt P symbolically expressed as $P =< f, t, d >$. In this context, f denotes the function entry declaration, t represents the task description and d encompasses input/output demonstrations; (2) S represents the canonical solution of the problem; (3) Test inputs T validate whether LCGM generated program is correct by comparing the generated solution with the canonical solution.

Our objective is to search for a concise one-sentence natural language description ϕ along with its corresponding implementation, λ. The purpose of ϕ is to provide clear instructions on how to handle the output value of the seed canonical solution, and λ translates these instructions into code implementation, aligning with the one-sentence natural language description ϕ. Consequently, the newly generated programming problem can be represented as $\mathcal{P}_{new} =< \phi \circ P, \lambda \circ S(\cdot), T >$.

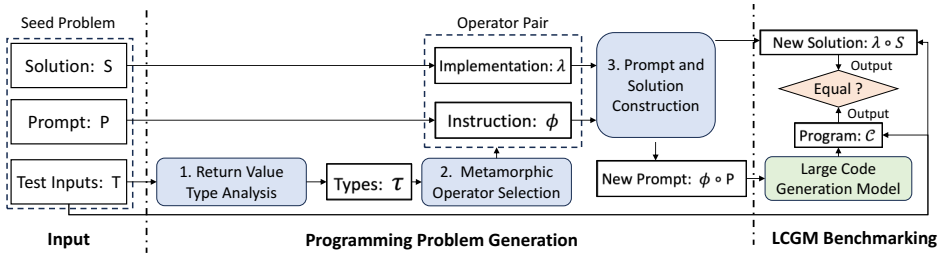

Fig. 2. Design overview of PPM.

4.2 Design Overview

The design overview of PPM is shown in Fig. 2, PPM accepts a seed problem as input and generates a new programming problem by performing three main steps:

(1) *Return Value Type Analysis.* In the first step, PPM performs return value analysis for each seed programming problem. It collects the returned values of the canonical solution and extracts their corresponding data types.
(2) *Metamorphic Operator Selection.* Following the analysis of return values, PPM proceeds to select an appropriate metamorphic operator based on the extracted data type. These operators, drawn from a predefined set (PPM provides templates for the user to expand this set), consist of paired transformations for both the prompt (ϕ) and the canonical solution (λ). Additionally, PPM randomly generates the parameters required for these selected transformations.
(3) *Prompt and Solution Construction.* Building upon the chosen metamorphic operator, PPM constructs a new programming problem by skillfully merging the prompt description with the solution, creating a new programming problem.

4.3 Return Value Type Analysis

The fundamental concept behind PPM is the fusion of two programming problems to generate a novel programming challenge with semantic diversity. Specifically, our aim is to leverage the output of a seed programming problem as input for our *lambda programming problem*. To select a *lambda programming problem* that avoids the introduction of type errors, the initial step necessitates the compilation of return value types from the seed problem. It's worth highlighting that specific strongly typed programming languages (such as C/C++ and Java) explicitly specify return types within function entry declarations. Regrettably, the prevailing trend in existing programming problem benchmarks leans heavily toward Python. Python, being one of the most widely used programming languages in the field of machine learning, is inherently weakly typed, lacking explicit return value type definitions.

In order to gather return value types in weakly typed programming languages, our approach entails the acquisition of actual return values through execution, followed by an analysis to ascertain their respective return types. Delving into further detail, our process begins with the definition of basic data types. Following this, we execute the canonical solution on the test inputs associated with the seed problem and subsequently analyze for potential complex data types.

Basic Data Type Definition. We have defined four fundamental built-in data types in Python, which serve as our core data types: *int*, *float*, *string*, *boolean*. These defined basic data types encompass a wide range of programming language constructs and can be extended for use in other programming languages.

Algorithm 1: Data Type Abstraction Algorithm. `Abstract` (·)

Input: Value list $\mathcal{V}$.
Output: Set of possible data types of returned values $\vec{\tau}$.

```
1: τ⃗ = { }                                  // Initialize possible data types as an empty set.
2: for each v in V do
3:     if Type(v) ∈ Basic Types then
4:         τ⃗ = τ⃗ + Type(v)                  // Add current basic data type to the set.
5:     else if Type(v) ∈ Enumerable Types then
6:         τ⃗ = τ⃗ + Abstract(ToList(v))      // Iteratively add each element type into the set.
7:     end if
8: end for
9: return τ⃗
```

Table 2. Metamorphic transformation operator set in PPM.

src_type	tgt_type	Natural Language Description	Implementation
Int	**float**	Change all src_type type values of the return values to tgt_type type, and add offset.	$\lambda(x)$: float(x) + offset
	string	Change all src_type type values of the return values to tgt_type type, return the string value of answer + offset.	$\lambda(x)$: str(x + offset)
	boolean	Change all src_type type values of the return values to tgt_type type, change all odd results to offset, and all even results to not offset.	$\lambda(x)$: offset if x % 2 else not offset
float	***int***	*Change all src_type type values of the return values to tgt_type type, keep the integer part of the result plus offset.*	$\lambda(x)$: str(x + offset)
	string	*Change all src_type type values of the return values to tgt_type type, return the string value of answer + offset.*	$\lambda(x)$: int(x) + offset
	boolean	*Change all src_type type values of the return values to tgt_type type, if the answer is larger than 0.0, return offset, else return not offset.*	$\lambda(x)$: offset if x > 0.0 else not offset
string	**int**	Change all src_type type values of the return values to tgt_type type, and return the length of the string plus offset.	$\lambda(x)$: len(x) + offset
	float	Change all src_type type values of the return values to tgt_type type, and return the length of the string plus offset.	$\lambda(x)$: len(x) + offset
	boolean	Change all src_type type values of the return values to tgt_type type, change all odd-length strings to offset, and all even-length strings to not offset.	$\lambda(x)$: offset if len(x) % 2 else not offset
boolean	**int**	Change all src_type type values of the return values to tgt_type type, and add offset	$\lambda(x)$: int(x) + offset
	float	Change all src_type type values of the return values to tgt_type type, and add offset	$\lambda(x)$: int(x) + offset
	string	Change all src_type type values of the return values to tgt_type type, and change True to offset, and False to chr(ord(offset) + 1).	$\lambda(x)$: offset if x else chr(ord(offset) + 1)
int	**int**	For all src_type type values in the return results, increase each value by offset.	$\lambda(x)$: x + offset
float	**float**	For all src_type type values in the return results, increase each value by offset.	$\lambda(x)$: x+ offset
string	**string**	For all src_type values in the return results, map each character in the src_type value to the character whose ASCII number is the current ASCII value plus offset.	$\lambda(x)$: "join([chr(ord(char) + offset) for char in x])
boolean	**boolean**	For all src_type values in the return results, invert True to False and False to True.	$\lambda(x)$: not x

Data Type Abstraction. Given the seed programming problem, we begin by subjecting all test cases from the test inputs to the canonical solution to execute and gather the output values. Subsequently, we analyze these output values and extract basic data types from the collected set. In addition to basic data types, a programming problem may produce other enumerable data types (*e.g.,* lists or tuples in Python). Therefore, we propose a recursive algorithm to abstract the data types of the return values. Our data type abstraction algorithm, as depicted in Algorithm 1, takes a list of returned values as input and iterates through each value in the list. If the value's type belongs to our predefined set of basic data types, we include that data type in our collection (Line 4). If the data value belongs to enumerable data types, we convert the value into a list and determine the types of all its elements (Line 6). Through this process, we are able to gather all basic data types present within the returned values.

4.4 Metamorphic Transformation Operator Selection

Once we have gathered the set of return value types, denoted as $\vec{\tau}$, from the canonical solution, we proceed by selecting a data type at random from this set. Additionally, we randomly choose a transformation operator from our predefined set of operators. Although PPM can support any user-defined operators, as a proof of concept, we introduce two types of transformation operators: *data type-aware value transformation* and *pure value transformation*. Our predefined operator set is detailed in Table 2, where column *src_type* represents the randomly selected source data type, *tgt_type* signifies the target data type to which we aim to transform, *Natural Language Description* provides a concise one-sentence natural language description denoted as ϕ, which we append

to the original task description, and *Implementation* column presents the corresponding code implementation λ based on the aforementioned natural language description.

In the *Implementation* column, it's worth noting that the lambda expression requires two parameters: x, which represents the output value of the seed canonical solution, and $offset$, a randomly generated variable. The introduction of this random variable $offset$ in the metamorphic transformation operators serves the purpose of injecting randomness into the newly generated programming problem. This injection of randomness is crucial for achieving long-term problem integrity. Furthermore, the space for the random variable $offset$ is user-configurable. By allowing users to define a sufficiently large search space for $offset$, PPM can effectively generate unique programming problems with a high probability of distinctiveness.

We present a theoretical analysis elucidating the probability of each proposed operator generating identical programming problems across two distinct trials on our website.

4.5 Prompt and Canonical Solution Construction

After generating a randomized metamorphic operator, comprising a one-sentence natural language description ϕ and its corresponding implementation λ in the previous step, PPM proceeds with the following steps to craft a novel programming problem.

Prompt Construction. In the prompt creation phase, PPM initially concatenates the one-sentence natural language description with the original task description, resulting in a new task description. Subsequently, PPM collects output demonstrations from the input/output demonstrations associated with the prompt and inputs them into the corresponding implementation to generate new output demonstrations. Through the combination of the original function entry declaration, the newly crafted task description, and the newly generated output demonstrations, PPM creates a new prompt.

Canonical Solution Construction. To generate a new canonical solution, PPM concatenates the implementation λ with the original solution construction S. In essence, we treat the output of the original solution construction as the input for the lambda implementation. It's important to note that our one-sentence natural language description ϕ specifies only one data type. Consequently, we apply a filter to ensure that the data type aligns with our natural language description before feeding it into our lambda implementation. PPM regards the ordered sequence of function calls within S and λ as our newly defined canonical solution.

4.6 Benchmarking Stage

To employ our newly created programming problem for benchmarking LCGMs, we first feed the new prompt $\phi \circ P$ to the LCGM and collect the generated program $C(\cdot)$. Then we evaluate whether the equation holds for all test inputs $C(x) == \lambda \circ S(x) \quad \forall x \in T$. If the equation holds, then the generated program $C(\cdot)$ is the correct one. By measuring the accuracy of the generated programming problem set, PPM can benchmark LCGMs.

5 EXPERIMENTAL SETUP

We present an empirical evaluation and aim to address the following research questions:

RQ1 Diversity: How diverse are the programming problems generated by PPM?
RQ2 Effectiveness: How effectively do the generated programming problems reveal the issues and limitations of existing LCGMs?
RQ3 Naturalness: How natural and realistic are the programming problems generated by PPM?
RQ4 Stability: Can PPM consistently produce stable benchmarking results despite randomness and under varying hyperparameters?

5.1 Datasets

We perform experiments using two datasets: *HumanEval* [10] and *MBPP-Sanitized* [5].
(1) *HumanEval.* The *HumanEval* dataset is an open-sourced benchmark proposed by OpenAI for evaluating the code-generation ability of pre-trained LCGM. It comprises 164 Python programming problems, each of which includes a prompt, a canonical solution, and a set of test inputs. The prompt consists of a natural language description of the problem to be solved, a function definition, and several input/output pairs.
(2) *MBPP-Sanitized.* The dataset utilized for our experiment comprises 427 Python programming questions collected from crowdsourcing. This dataset is considered a zero-shot dataset since it lacks any input/output demonstrations in its prompts. To enhance the experiment's effectiveness, we applied a prompt format modification. In detail, each problem was processed by adding a function header and converting the natural language instructions into function docstrings.

5.2 Pre-trained Code Generation Models

In this work, we perform a comprehensive evaluation of PPM on eight popular public LCGM. The selected LCGM are diverse in terms of model architecture, model size, and training methods. **(1) CodeGen Family** This type of model [27] belongs to a family of autoregressive language models that undergo pretraining on code data. Our experimentation encompasses the utilization of two variants within the open-source CodeGen model family, scaling 6B-half and 2B separately. **(2) CodeGen2 Family** CodeGen2 [26] is an upgraded version of the CodeGen family, which falls under the category of autoregressive language models. It builds upon the capabilities of CodeGen and introduces additional features such as code filling. **(3) InCoder Family** The InCoder model [18] is available in two sizes: 1B and 6B. The decoder-only transformer utilizes a causal-masked objective during training to effectively handle code generation. **(4) SantaCoder Family** SantaCoder [1] is a model that utilizes a multi-head attention mechanism as its primary model architecture. Despite having a relatively compact size with 1.1 billion model parameters, SantaCoder delivers exceptional performance surpassing that of other models with similar parameter sizes. **(5) PloyCoder** This model utilizes the GPT2 architecture and underwent training on an extensive dataset of 249 GB of code encompassing 12 programming languages. It comprises an impressive 2.7 billion parameters trained over 100,000 to 150,000 steps.

5.3 Comparison Baselines

We compare PPM against night baseline methods designed for creating programming problems. **(1) `Base`** This approach involves directly utilizing the prompts provided in the manually crafted dataset to generate code snippets without any modification. **(2) `Add Demo`** This method involves the random selection of a test case from the test inputs. It proceeds by calculating the expected output through the application of the provided canonical solution. Subsequently, it adds this new input/output case into the seed prompt, alongside the original input/output pairs, to create a new prompt. **(3) `Remove Demo`** Similar to the prompt crafting method in `Add Demo`, this method randomly removes an input/output demo from the original prompt. **(4) `Replace Demo`** Similar to the prompt crafting method in `Add Demo`, this method randomly replaces an input/output demo in the original prompt with the one in the test inputs. **(5) `Token Mutation`** This method is widely used in existing work to evaluate the robustness of natural language processing (NLP) models [33]. Specifically, this method randomly replaces a token in the task description with another token using token substitution. **(6)`Character Mutation`** This method entails the random alteration of a token within the natural language task description at the character level. In our experiment, we examine four distinct types of character mutation operators: (1) neighbor character swap, (2) random character

insertion, (3) random character deletion, and (4) homoglyph character replacement. **(7) `FuncName Mutation`** This method introduces perturbations to function names within the function entry declaration using the following operators: (1) CamelCase transformation, which involves altering function names between camel-case and snake-case (*e.g.,* "findCharLong" to "find_char_long"). (2) Application of all character mutation operators found in the `Character Mutation` method. **(8) `Empty Line Insertion`** This method [33] involves the insertion of an empty line into the seed prompt to generate a new prompt variant. **(9) `CommSyntax`** This method [33] transforms the syntax of the docstring section, including the task description and input/output demonstrations, from its original format (*e.g.,* """docstring""") into comment syntax (*e.g.,* #docstring).

5.4 Evaluation Metrics

Diversity Metrics. To assess the diversity of the generated programming problems, we conduct a measurement encompassing both the prompt and canonical solution aspects. In evaluating prompt diversity, we employ two metrics: (1) *BLEU score* and (2) *Semantic Similarity*. As for assessing diversity in canonical solutions, we rely on the *Different Implementation Rate* metric. *BLEU score* is widely employed as a quantitative measure of the linguistic diversity of natural language sentences. The *BLEU score* is formally defined in Eq.(1),

$$\text{BLEU} = \text{BP} \times \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \text{BP} = \begin{cases} 1 & \text{if c} \geq \text{r} \\ \exp^{(1-\frac{\text{r}}{\text{c}})} & \text{otherwise} \end{cases} \tag{1}$$

where BP is the brevity penalty, which is a correction factor that accounts for the difference in length between the candidate and reference sentences. Following previous works, we use *BLEU-4*, where N is set to 4, and w_n is set to 0.25 for all n (uniform weighting). Lower *BLUE scores* indicate a greater mismatch to the original problems and reflect a higher level of diversity. *Semantic Similarity (SemSim)* serves as a crucial metric in evaluating the semantic diversity of generated programming problem prompts. It quantifies the similarity in semantic meaning between the generated problems and seed problems. The *Semantic Similarity* is computed using the following equations:

$$\text{Semantic Similarity} = \frac{1}{N}\sum_{i=1}^{N} \cos(\text{Embed}(P_i),\ \text{Embed}(P_i')) \tag{2}$$

In this equation, Embed$(\cdot)$ represents the embedding function that projects sentences to numeric vectors. P_i and P_i' are the i^{th} original problem and generated problem descriptions, respectively. In our approach, we utilize the open-source embedding function known as `sentence-transformers`, which is accessible through HuggingFace [1]. This serves as our chosen embedding function for the task. The function $\cos(\cdot)$ measures the cosine similarity between two numeric vectors. A lower *Semantic Similarity* score suggests greater diversity among the generated programming problems, as they exhibit distinct variations in semantic meaning compared to the original problems. *Different Implementation Rate (DiffImp).* In order to assess the diversity of the canonical solutions, we employ the *Different Implementation Rate* metric. This metric is calculated by determining the percentage of generated programming problems that employ distinct canonical solutions compared to the seed programming problems.

Effectiveness Metrics. To evaluate the programming capability of each LCGM with the crafted programming problems, we adopt a well-established methodology used in previous research [10]. For this assessment, we utilize the *Pass@k* metric, which serves as a measure of the functional

[1] https://huggingface.co/sentence-transformers/bert-base-nli-mean-tokens

correctness of the generated code by executing test cases. The *Pass@k* metric is defined as follows:

$$\text{Pass@k} = \mathbb{E}_{\text{problems}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right] \tag{3}$$

In the equation, the symbol $\mathbb{E}_{\text{problems}}$ represents the expected value calculated over the entire set of programming problems. The parameter n indicates the number of generated code snippets for each programming problem, and the variable c denotes the count of correct code snippets that successfully pass all the test cases. A lower *Pass@k* indicates that the model is less accurate in solving the programming problems, and thus these programming problems are less likely to overlap with the models' training dataset. Consistent with previous research, we set $k = 1, 10, 100$ for our evaluation. Additionally, we use the relative *Pass@k* drop as another evaluation metric. It allows us to directly compare the performance of the original and newly generated programming problems. This metric is calculated as the percentage difference in *Pass@k* values between the two sets.

Naturalness Metrics. To measure the naturalness of the generated programming problems, we utilize three metrics. Our first metric is *Perplexity*, which is widely used to measure the naturalness of natural language sentences. *Perplexity* is a metric that evaluates how well a language model predicts a sequence of words. Formally, given a tokenized problem description as $(x_1, x_2, \cdots, x_t)$, *perplexity* is computed as follows:

$$\text{Perplexity} = \text{EXP}\left\{-\frac{1}{t}\sum_{i=1}^{t}\log p(x_i; |x_1, \cdots, x_{i-1})\right\} \tag{4}$$

Here, $p(x_i; |x_1, \cdots, x_{i-1})$ represents the probabilities obtained using a well-trained language model to predict the i^{th} token. A well-trained language model is expected to assign higher probabilities to more natural and fluent sentences, while less natural or grammatically incorrect sentences will receive lower probabilities. A lower *perplexity* value indicates that the language model is more confident and accurate in predicting the sentence, thereby suggesting that the sentence is more natural and fluent. Our second metric is the *Number of IDE Warnings*, which measures the total number of warnings of the generated programming problem prompts. A lower number of IDE warnings signifies that the programming prompts are more natural and realistic. Our final metric for assessing naturalness is *Human Scores*, wherein we enlist the help of volunteers to assign scores to each programming problem prompt. Following established practices [33], we distribute each programming prompt to five human annotators who possess familiarity with Python. These annotators are tasked with rating the naturalness on a scale ranging from 0 (not natural) to 0.5 (possible but rare in practice) to 1 (natural).

5.5 Experiment Process

RQ1 Process. To tackle RQ1, we perform two series of experiments aimed at evaluating the *external diversity* and *internal diversity* of the generated programming problems, respectively. External diversity quantifies the dissimilarity between the newly generated programming problems and the original seed programming problems. Conversely, internal diversity measures the variability within each problem-generation method across multiple trials. In *external diversity* evaluation, each programming problem within our evaluation datasets serves as a seed problem. We then apply each problem-generation method to create new programming problems and measure diversity metrics by comparing the seed problem to the generated one. For our *internal diversity* evaluation, we conducted two experiments. In our first experiment, we conducted two runs for each method. As a result, for each seed problem, we obtain two versions of the new problem generated by each method. We compute the diversity metric by comparing these two versions of the problems. In our second experiment, our goal is to showcase the effective ability of PPM in ensuring long-term

data integrity. To achieve this, we run PPM once on the seed problem, resulting in the creation of the initial version of the new programming problems. Following this, we iteratively run PPM K times, generating K distinct versions. After that, we calculate the different implementation rates (*DiffImp*) by comparing the initial generated version with the K subsequent versions. If a problem in the initial version does not match any implementation in the K versions, we consider it to have a different implementation. Otherwise, we treat it as the same implementation. To ensure statistical robustness and mitigate the influence of randomness, we repeat each experiment ten times and report the average values of the metrics.

RQ2 Process. To address RQ2, we conducted two experiments. In our first experiment, we aim to explore the effectiveness of our generated semantic-diverse programming problems in uncovering the limitations of existing LCGMs. To achieve this objective, we utilize all problem-generation methods to create programming problems. Subsequently, we input each problem's prompt into each LCGM and collect the generated solution programs. Following this, we execute the canonical solution on the test inputs to obtain the expected outputs. Finally, we execute these generated solution programs on the provided test inputs and collect the resulting outputs. Ultimately, we compare the executed outputs with the expected outputs to analyze the effectiveness of the methods. If our generated semantic-diverse programming problems are effective in uncovering the limitations of existing LCGMs, then each LCGM would show a significant drop on *Pass@k* metric. For our second experiment, our goal is to investigate whether the drop in correctness of LCGM is primarily due to inherent limitations rather than the level of challenge presented by our *lambda programming problem*. To do this, we feed our *lambda programming problem* to each LCGM and measure the *Pass@k* metric with the generated programs. If each LCGM can produce high *Pass@k* on our *lambda programming problem*, then the effectiveness drop in our first experiment can be attributed to the inherent limitations of LCGM rather than the complexity of our *lambda programming problem*.

RQ3 Process. We begin by performing a quantitative evaluation to measure the *perplexity* of the problem prompts generated by each method. Calculating the *perplexity* of these prompts (Eq.(4)) requires a well-trained language model to predict each token probability in the prompts. We follow established methodology [29] and employ GPT-2 as our language model due to it being trained on large-scale natural language corpus and widely used in existing work for measuring *perplexity*. Later, we copy all the generated programming problem prompts into a widely used Python Integrated Development Environment (IDE), namely *PyCharm*. We then count the IDE warnings of each method, considering that the original programming problem prompts contained some pre-existing warnings stemming from typos errors. Hence, when calculating the number of IDE warnings, we exclude those that were part of the original dataset. Finally, we conduct a human study to investigate the naturalness of the generated programming problem prompt. Specifically, we follow existing work [33] and conduct a human study. In the human study, we assign each problem description to five human annotators who possess familiarity with the Python language. These annotators are asked to rate the naturalness of the problems on a scale from 0 to 1. Specifically, the scale is defined as follows: 0 denotes *not natural*, 0.5 indicates *possible to appear in practice but rare*, and 1 represents *natural*. For the evaluation of naturalness, we exclude demo modification methods (*e.g.*, `Add Demo`, `Remove Demo`, and `Replace Demo`) as they do not modify the problem description.

RQ4 Process. To showcase the stability of benchmarking results and consistent performance across varied hyperparameters by PPM, we conduct two series of experiments. In our first experiment, we aim to illustrate that despite the introduction of randomness in the programming problem generation process, this randomness does not significantly impact benchmarking results. To achieve this objective, we run PPM five trials and utilize the programming problems generated in each trial to benchmark every LCGM. Subsequently, we present the averaged *Pass@k* and its variance. In the second series of experiments, our aim is to demonstrate the robustness of PPM concerning

Table 3. Diversity results.

Methods	External Diversity						External Diversity					
	HumanEval			MBPP			HumanEval			MBPP		
	BLEU-4 ↓	SemSim ↓	DiffImp ↑	BLEU-4 ↓	SemSim ↓	DiffImp ↑	BLEU-4 ↓	SemSim ↓	DiffImp ↑	BLEU-4 ↓	SemSim ↓	DiffImp ↑
Base	1.00	1.00	0.00	1.00	1.00	0.00	1.00	1.00	0.00	1.00	1.00	0.00
Add Demo	1.00	1.00	0.00	0.86	1.00	0.00	1.00	1.00	0.00	1.00	1.00	0.00
Del Demo	1.00	1.00	0.00	-	-	0.00	1.00	1.00	0.00	1.00	1.00	0.00
Rep Demo	1.00	1.00	0.00	-	-	0.00	1.00	1.00	0.00	1.00	1.00	0.00
Token Mutation	0.82	0.96	0.00	0.76	0.95	0.00	**0.72**	**0.95**	0.00	**0.66**	**0.92**	0.00
Char Mutation	0.84	0.97	0.00	0.78	0.92	0.00	0.81	0.97	0.00	0.78	0.94	0.00
Func Mutation	0.98	1.00	0.00	0.98	1.00	0.00	1.00	1.00	0.00	1.00	1.00	0.00
Insert Line	1.00	1.00	0.00	1.00	1.00	0.00	1.00	1.00	0.00	1.00	1.00	0.00
CommSyntax	0.81	0.98	0.00	0.73	0.99	0.00	1.00	1.00	0.00	1.00	1.00	0.00
PPM-V	0.69	**0.89**	**1.00**	0.57	**0.84**	**1.00**	0.97	0.96	0.75	0.96	0.94	0.76
PPM-T	**0.66**	0.90	**1.00**	**0.54**	0.90	**1.00**	0.84	0.97	**0.98**	0.81	0.96	**0.97**

hyperparameter configurations in LCGMs. Specifically, we conduct two experiments to explore the impact of different LCGM inference settings on the performance of PPM. Our focus is on analyzing PPM's responsiveness to two critical hyperparameters: the number of code candidates (denoted as n in Eq.(3)) and sampling temperatures. Regarding the hyperparameter concerning the number of code candidates, we set n within a range spanning from 10 to 100 and evaluate the efficacy metric *Pass@1*. Regarding the temperature hyperparameter, we vary the temperature across the range of 0.1 to 0.9, and we present the *Pass@1* and *Pass@10* metrics of PPM. Due to limitations in space, we present results for the *HumanEval* dataset with four models exclusively.

5.6 Implementation Details

We conducted our evaluation on a server equipped with an Intel Xeon E5-26 CPU and eight NVIDIA A4500 GPUs. For each problem, we generated 100 candidate solutions per model and computed the effectiveness metric. We then examined the impact of the number of candidates in §6.4 to gain insights into how it affects performance. In using LCGMs to generate code solutions, we set the inference temperature as 0.7, following existing literature, and we further explored the effect of temperature in §6.4. To ensure the reliability of our findings for each research question, we ran each method multiple times and reported averaged results, mitigating randomness influence.

6 EVALUATION RESULTS

6.1 RQ1 Diversity

External Diversity Results. The results of external diversity are presented in the left section of Table 3. From the results, we have the following observations: Firstly, regarding prompt diversity, both linguistic and semantic aspects of the prompts exhibit lower diversity in our approaches compared to the baseline methods. This indicates that our approach excels in generating diverse problem descriptions, primarily by proposing a concrete and novel programming problem description distinct from the original, whereas the baseline methods merely modify certain tokens in the descriptions. Secondly, in the context of solutions, PPM achieves a 100% different implementation rate, while all baseline methods consistently yield a rate of 0. This stems from our innovative programming problem merging technique, allowing PPM to generate varied canonical solutions unlike the original canonical solutions, a capability lacking in the baseline methods.

Internal Diversity Results. The internal diversity results are presented in the right section of Table 3. From the results, we note that `Token Mutation` achieves a higher level of diversity in the prompt part. This is attributed to the random token mutations performed in each run by `Token Mutation`. On the other hand, PPM modifies only the *offset* value within its template, resulting in lower prompt diversity when comparing two different trial runs. However, PPM demonstrates a notable advantage in solution diversity. Specifically, in our *pure value transformation*, we observed considerably lower solution diversity compared to our *type-aware value transformation*. This

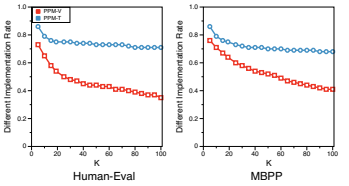

Fig. 3. *DiffImp* of PPM.

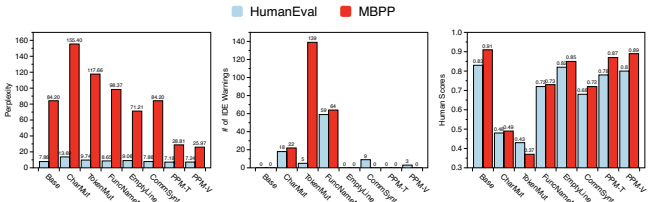

Fig. 4. Naturalness results.

Table 4. The effectiveness evaluation on the HumanEval dataset.

Methods	CodeGen-2b			CodeGen-6b			CodeGen2-3.7b			CodeGen2-1b		
	Pass@1	Pass@10	Pass@100	Pass@1	Pass@10	Pass@100	Pass@1	Pass@10	Pass@100	Pass@1	Pass@10	Pass@100
Base	0.2	0.41	0.6	0.24	0.48	0.69	0.11	0.23	0.37	0.08	0.15	0.25
Add Demo	0.2 (0.00%)	0.41 (0.00%)	0.62 (+3.33%)	0.24 (0.00%)	0.49 (+2.08%)	0.75 (+8.70%)	0.11 (0.00%)	0.24 (+4.35%)	0.4 (+8.11%)	0.07 (-12.50%)	0.15 (0.00%)	0.29 (+16.00%)
Del Demo	0.2 (0.00%)	0.41 (0.00%)	0.64 (+6.67%)	0.24 (0.00%)	0.49 (+2.08%)	0.7 (+1.45%)	0.11 (0.00%)	0.22 (-4.35%)	0.37 (0.00%)	0.08 (0.00%)	0.15 (0.00%)	0.25 (0.00%)
Rep Demo	0.2 (0.00%)	0.4 (-2.44%)	0.61 (+1.67%)	0.24 (0.00%)	0.48 (0.00%)	0.73 (+5.80%)	0.11 (0.00%)	0.24 (+4.35%)	0.39 (+5.41%)	0.08 (0.00%)	0.15 (0.00%)	0.26 (+4.00%)
Token Mutation	0.19 (-5.00%)	0.4 (-2.44%)	0.58 (-3.33%)	0.22 (-8.33%)	0.44 (-8.33%)	0.67 (-2.90%)	0.1 (-9.09%)	0.22 (-4.35%)	0.35 (-5.41%)	0.06 (-25.00%)	0.14 (-6.67%)	0.26 (+4.00%)
Char Mutation	0.17 (-15.00%)	0.36 (-12.20%)	0.53 (-11.67%)	0.22 (-8.33%)	0.44 (-8.33%)	0.63 (-8.70%)	0.1 (-9.09%)	0.23 (0.00%)	0.39 (+5.41%)	0.07 (-12.50%)	0.13 (-13.33%)	0.21 (-16.00%)
FuncName Mutation	0.19 (-5.00%)	0.4 (-2.44%)	0.59 (-1.67%)	0.23 (-4.17%)	0.47 (-2.08%)	0.72 (+4.35%)	0.1 (-9.09%)	0.23 (0.00%)	0.39 (+5.41%)	0.06 (-25.00%)	0.14 (-93.33%)	0.24 (-4.00%)
Insert Line	0.19 (-5.00%)	0.41 (0.00%)	0.67 (+11.67%)	0.24 (0.00%)	0.48 (0.00%)	0.7 (+1.45%)	0.11 (0.00%)	0.25 (+8.70%)	0.39 (+5.41%)	0.08 (0.00%)	0.14 (-6.67%)	0.24 (-4.00%)
CommSyntax	0.17 (-15.00%)	0.37 (-9.76%)	0.63 (+5.00%)	0.19 (-20.83%)	0.43 (-10.42%)	0.7 (+1.45%)	0.07 (-36.36%)	0.19 (-17.39%)	0.34 (-8.11%)	0.06 (-25.00%)	0.13 (-13.33%)	0.22 (-12.00%)
PPM-V	**0.02 (-90.00%)**	**0.1 (-75.61%)**	0.25 (-58.33%)	**0.03 (-87.50%)**	0.12 (-75.00%)	0.26 (-62.32%)	**0 (-99.99%)**	**0.03 (-86.96%)**	**0.09 (-75.68%)**	**0 (-99.99%)**	**0.02 (-86.67%)**	**0.08 (-68.00%)**
PPM-T	0.01 (-95.00%)	0.07 (-82.93%)	**0.16 (-73.33%)**	0.02 (-91.67%)	**0.09 (-81.25%)**	**0.21 (-69.57%)**	0.01 (-90.91%)	0.03 (-86.96%)	0.08 (-78.38%)	0 (-99.99%)	0.01 (-93.33%)	0.06 (-76.00%)

Methods	Incoder-1b			Incoder-6b			Santacoder-1.1b			PolyCoder		
	Pass@1	Pass@10	Pass@100	Pass@1	Pass@10	Pass@100	Pass@1	Pass@10	Pass@100	Pass@1	Pass@10	Pass@100
Base	0.06	0.16	0.28	0.11	0.28	0.48	0.15	0.29	0.45	0.04	0.1	0.21
Add Demo	0.06 (0.00%)	0.15 (-6.25%)	0.22 (-21.43%)	0.11 (0.00%)	0.27 (-3.57%)	0.42 (-12.50%)	0.15 (0.00%)	0.3 (+3.45%)	0.47 (+4.44%)	0.04 (0.00%)	0.1 (0.00%)	0.17 (-19.05%)
Del Demo	0.07 (+16.67%)	0.16 (0.00%)	0.27 (-3.57%)	0.1 (-9.09%)	0.26 (-7.14%)	0.44 (-8.33%)	0.15 (0.00%)	0.3 (+3.45%)	0.48 (+6.67%)	0.04 (0.00%)	0.1 (0.00%)	0.18 (-14.29%)
Rep Demo	0.07 (+16.67%)	0.17 (+6.25%)	0.31 (+10.71%)	0.11 (0.00%)	0.27 (-3.57%)	0.45 (-6.25%)	0.15 (0.00%)	0.3 (+3.45%)	0.49 (+8.89%)	0.04 (0.00%)	0.1 (0.00%)	0.16 (-23.81%)
Token Mutation	0.05 (-16.67%)	0.15 (-6.25%)	0.24 (-14.29%)	0.1 (-9.09%)	0.25 (-10.71%)	0.45 (-6.25%)	0.14 (-6.67%)	0.28 (-3.45%)	0.47 (+4.44%)	0.03 (-25.00%)	0.1 (0.00%)	0.19 (-9.52%)
Char Mutation	0.06 (0.00%)	0.15 (-6.25%)	0.29 (+3.57%)	0.09 (-18.18%)	0.24 (-14.29%)	0.42 (-12.50%)	0.13 (-13.33%)	0.28 (-3.45%)	0.45 (0.00%)	0.03 (-25.00%)	0.09 (-10.00%)	0.18 (-14.29%)
FuncName Mutation	0.06 (0.00%)	0.15 (-6.25%)	0.26 (-7.14%)	0.1 (-9.09%)	0.26 (-7.14%)	0.43 (-10.42%)	0.14 (-6.67%)	0.28 (-3.45%)	0.48 (+6.67%)	0.04 (0.00%)	0.1 (0.00%)	0.20 (-4.76%)
Insert Line	0.07 (+16.67%)	0.16 (0.00%)	0.27 (-3.57%)	0.11 (0.00%)	0.26 (-7.14%)	0.45 (-6.25%)	0.15 (0.00%)	0.3 (+3.45%)	0.49 (+8.89%)	0.04 (0.00%)	0.09 (-10.00%)	0.19 (-9.52%)
CommSyntax	0.04 (-33.33%)	0.13 (-18.75%)	0.23 (-17.86%)	0.09 (-18.18%)	0.25 (-10.71%)	0.42 (-12.50%)	0.12 (-20.00%)	0.28 (-3.45%)	0.44 (-2.22%)	0.03 (-25.00%)	0.09 (-10.00%)	0.18 (-14.29%)
PPM-V	**0.01 (-83.33%)**	0.04 (-75.00%)	0.12 (-57.14%)	0.02 (-81.82%)	0.08 (-71.43%)	0.16 (-66.67%)	**0.02 (-86.67%)**	**0.07 (-75.86%)**	0.16 (-64.44%)	0 (-99.99%)	0.01 (-90.00%)	0.08 (-61.90%)
PPM-T	0 (-99.99%)	**0.02 (-87.50%)**	**0.07 (-75.00%)**	**0.01 (-90.91%)**	**0.04 (-85.71%)**	**0.11 (-77.08%)**	0.01 (-93.33%)	0.04 (-86.21%)	**0.12 (-73.33%)**	0.01 (-75.00%)	0.01 (-90.00%)	0.06 (-71.43%)

disparity arises from certain programming problems in the original dataset that only return *boolean* type values. Without altering the return value type, the random search space for *boolean* values remains limited to two possibilities, such as *True* and *False*.

The different implementation rates achieved by PPM through multiple trials are depicted in Fig. 3. Examining the outcomes, we observe that as k increases from 0 to 100, the programming problems exhibiting different semantics (i.e., requiring distinct implementations) decrease initially and then stabilize. These findings indicate that even after 100 attempts, approximately 40% of the programming problems for PPM-V and 70% for PPM-T remain unrepeated. These results imply the efficacy of PPM in maintaining long-term data integrity and its resilience against potential risks of training data leakage, even when a version of programming problems from PPM is publicly available on the Internet.

> Answers to **RQ1**: Based on our experimental findings, it is evident that PPM can generate programming problems that deviate from the initial seed, presenting diverse problem descriptions and solutions. Moreover, upon executing PPM multiple times, the likelihood of generating programming problems that share identical solutions is minimal.

6.2 RQ2: Effectiveness

New Problem Results. The effectiveness results are presented in Table 4 and Table 5, where Table 4 corresponds to the *HumanEval* dataset, and Table 5 corresponds to the *MBPP* dataset. As the *MBPP* dataset is a zero-shot dataset with no demonstrations in its prompts, the evaluation methods `Del Demo` and `Replace Demo` cannot be applied to this dataset. The numbers displayed without

Table 5. The effectiveness evaluation on the MBPP dataset.

Methods	CodeGen-2b			CodeGen-6b			CodeGen2-3.7b			CodeGen2-1b		
	Pass@1	Pass@10	Pass@100	Pass@1	Pass@10	Pass@100	Pass@1	Pass@10	Pass@100	Pass@1	Pass@10	Pass@100
Base	0.36	0.67	0.82	0.39	0.70	0.84	0.17	0.45	0.65	0.13	0.37	0.57
Add Demo	0.37 (+2.78%)	0.68 (+1.49%)	0.83 (+1.22%)	0.45 (+15.38%)	0.73 (+4.29%)	0.86 (+2.38%)	0.22 (+29.41)	0.52 (+15.56%)	0.71 (+9.23%)	0.13 (0.00%)	0.39 (+5.41%)	0.61 (+7.02%)
Token Mutation	0.32 (-11.11%)	0.63 (-5.97%)	0.79 (-3.66%)	0.37 (-5.13%)	0.67 (-4.29%)	0.83 (-1.19%)	0.16 (-5.88%)	0.43 (-4.44%)	0.62 (-4.62%)	0.11 (-15.38%)	0.35 (-5.41%)	0.55 (-3.51%)
Char Mutation	0.26 (-27.78%)	0.56 (-16.42%)	0.76 (-7.32%)	0.3 (-23.08%)	0.61 (-12.86%)	0.78 (-7.14%)	0.13 (-23.53%)	0.37 (-17.78%)	0.57 (-12.31%)	0.09 (-30.77%)	0.32 (-13.51%)	0.51 (-10.53%)
FuncName Mutation	0.34 (-5.56%)	0.66 (-1.49%)	0.81 (-1.22%)	0.39 (0.00%)	0.7 (0.00%)	0.85 (+1.19%)	0.17 (0.00%)	0.45 (0.00%)	0.65 (0.00%)	0.12 (-7.69%)	0.37 (0.00%)	0.57 (0.00%)
Insert Line	0.35 (-2.78%)	0.65 (-2.99%)	0.82 (0.00%)	0.39 (0.00%)	0.7 (0.00%)	0.85 (+1.19%)	0.16 (-5.88%)	0.44 (-2.22%)	0.63 (-3.08%)	0.12 (-7.69%)	0.35 (-5.41%)	0.56 (-1.75%)
CommSyntax	0.26 (-27.78%)	0.61 (-8.96%)	0.8 (-2.44%)	0.29 (-25.64%)	0.65 (-7.14%)	0.83 (-1.19%)	0.12 (-29.41%)	0.39 (-13.33%)	0.57 (-12.31%)	0.08 (-38.46%)	0.3 (-18.92%)	0.53 (-7.02%)
PPM-V	0.04 (-88.89%)	0.18 (-73.13%)	0.37 (-54.88%)	0.04 (-89.74%)	0.18 (-74.29%)	0.37 (-55.95%)	0.01 (-94.12%)	0.07 (-84.44%)	0.18 (-72.31%)	0.01 (-92.31%)	0.07 (-81.08%)	0.21 (-63.16%)
PPM-T	0.06 (-83.33%)	0.22 (-67.16%)	0.39 (-52.44%)	0.06 (-84.62%)	0.21 (-70.00%)	0.4 (-52.38%)	0.03 (-82.35%)	0.12 (-73.33%)	0.24 (-63.08%)	0.02 (-84.62%)	0.11 (-70.27%)	0.25 (-56.14%)
Methods	**Incoder-1b**			**Incoder-6b**			**Santacoder-1.1b**			**PolyCoder**		
	Pass@1	Pass@10	Pass@100	Pass@1	Pass@10	Pass@100	Pass@1	Pass@10	Pass@100	Pass@1	Pass@10	Pass@100
Base	0.12	0.39	0.59	0.13	0.44	0.65	0.26	0.56	0.71	0.08	0.30	0.52
Add Demo	0.14 (+16.67%)	0.41 (+5.13%)	0.61 (+3.39%)	0.21 (+61.54%)	0.5 (+13.64%)	0.69 (+6.15%)	0.31 (+19.23%)	0.58 (+3.57%)	0.75 (5.63%)	0.10 (+25.00%)	0.32 (+6.67%)	0.53 (+1.92%)
Token Mutation	0.1 (-16.67%)	0.37 (-5.13%)	0.56 (-5.08%)	0.12 (-7.69%)	0.41 (-6.82%)	0.64 (-1.54%)	0.24 (-7.69%)	0.54 (-3.57%)	0.7 (-1.41%)	0.06 (-25.00%)	0.28 (-6.67%)	0.51 (-1.92%)
Char Mutation	0.09 (-25.00%)	0.33 (-15.38%)	0.52 (-11.86%)	0.1 (-23.08%)	0.36 (-18.18%)	0.6 (-7.69%)	0.19 (-26.92%)	0.47 (-16.07%)	0.68 (-4.23%)	0.07 (-12.50%)	0.25 (-16.67%)	0.44 (-15.38%)
FuncName Mutation	0.12 (0.00%)	0.38 (-2.56%)	0.57 (-3.39%)	0.12 (-7.69%)	0.42 (-4.55%)	0.64 (-1.54%)	0.25 (-3.85%)	0.56 (0.00%)	0.73 (+2.82%)	0.08 (0.00%)	0.3 (0.00%)	0.5 (-3.85%)
Insert Line	0.12 (0.00%)	0.39 (0.00%)	0.58 (-1.69%)	0.12 (-7.69%)	0.41 (-6.82%)	0.63 (-3.08%)	0.22 (-15.38%)	0.53 (-5.36%)	0.71 (0.00%)	0.07 (-12.5%)	0.28 (-6.67%)	0.51 (-1.92%)
CommSyntax	0.05 (-58.33%)	0.23 (-41.03%)	0.48 (-18.64%)	0.09 (-30.77%)	0.36 (-18.18%)	0.6 (-7.69%)	0.23 (-11.54%)	0.55 (-1.79%)	0.74 (+4.23%)	0.08 (0.00%)	0.28 (-6.67%)	0.5 (-3.85%)
PPM-V	0.01 (-91.67%)	0.08 (-79.49%)	0.23 (-61.02%)	0.02 (-84.62%)	0.11 (-75.00%)	0.27 (-58.46%)	0.03 (-88.46%)	0.14 (-75.00%)	0.32 (-54.93%)	0.01 (-87.50%)	0.07 (-76.67%)	0.18 (-65.38%)
PPM-T	0.02 (-83.33%)	0.1 (-74.36%)	0.23 (-61.02%)	0.02 (-84.62%)	0.1 (-77.27%)	0.25 (-61.54%)	0.04 (-84.62%)	0.14 (-75.00%)	0.31 (-56.34%)	0 (-99.99%)	0.06 (-80.00%)	0.17 (-67.37%)

Table 6. The Pass@k of each LCGM on our lambda programming problem.

	CodeGen-2b			CodeGen-6b			CodeGen2-3.7b			CodeGen2-1b		
	Pass@1	Pass@10	Pass@100	Pass@1	Pass@10	Pass@100	Pass@1	Pass@10	Pass@100	Pass@1	Pass@10	Pass@100
	0.51	0.91	0.92	0.50	0.89	0.92	0.13	0.59	0.75	0.14	0.56	0.83
PPT-V	Incoder-1b			Incoder-6b			Santacoder-1.1b			PolyCoder		
	Pass@1	Pass@10	Pass@100	Pass@1	Pass@10	Pass@100	Pass@1	Pass@10	Pass@100	Pass@1	Pass@10	Pass@100
	0.16	0.55	0.83	0.42	0.87	0.92	0.48	0.88	0.92	0.06	0.36	0.67
	CodeGen-2b			CodeGen-6b			CodeGen2-3.7b			CodeGen2-1b		
	Pass@1	Pass@10	Pass@100	Pass@1	Pass@10	Pass@100	Pass@1	Pass@10	Pass@100	Pass@1	Pass@10	Pass@100
	0.91	1.00	1.00	0.97	1.00	1.00	0.73	0.99	1.00	0.59	1.00	1.00
PPT-T	Incoder-1b			Incoder-6b			Santacoder-1.1b			PolyCoder		
	Pass@1	Pass@10	Pass@100	Pass@1	Pass@10	Pass@100	Pass@1	Pass@10	Pass@100	Pass@1	Pass@10	Pass@100
	0.60	0.93	1.00	0.66	0.97	1.00	0.87	1.00	1.00	0.44	0.88	1.00

brackets in the tables represent the measured *Pass@k* values, reflecting the functional correctness of the generated code. On the other hand, the numbers enclosed in brackets represent the relative *Pass@k* drop compared with the Base model. This drop quantifies the change in performance between the original programming problems and the newly crafted programming problems.

From the results in Table 4 and Table 5, we observe: *(1)* In all settings, PPM demonstrates a remarkable capability to significantly decrease the code generation model's performance, setting it apart from the baseline. For example, the *Pass@1* value decreased by 90% and 85% for our approaches. In contrast, the *Pass@1* values either remained the same or slightly dropped by about 5% to 15% for the baseline methods. This is because our newly crafted programming problem changes the semantics of the problem description, while the baseline approaches do not. *(2)* The baseline method can even increase the model's functional correctness. For instance, the *Pass@1* values increase by 16.77% when deleting or replacing input/output demos in the prompts. *(3)* More demos do not always yield better correctness. For instance, when k is set at 10 and 100, the *Pass@k* values for the Incoder-1b drop by about 6.25% and 21.43%, respectively, in the *HumanEval* dataset.

Lambda Programming Problem Results. The *Pass@k* scores for each LCGM in our *lambda programming problem* are presented in Table 6. For each type of *lambda programming problem*, each LCGM achieves almost perfect accuracy. Upon comparing these scores with the *Pass@1* values from Table 4 and Table 5, a notable observation emerges: the *Pass@k* scores for our *lambda programming problem* significantly surpass those of the original dataset. These results imply that our proposed *lambda programming problem* is not considerably more challenging for LCGM to understand. Consequently, the reduction in LCGM's *Pass@k* for the merged programming problem can be attributed to the merging concept, rather than our specific *lambda programming problem*.

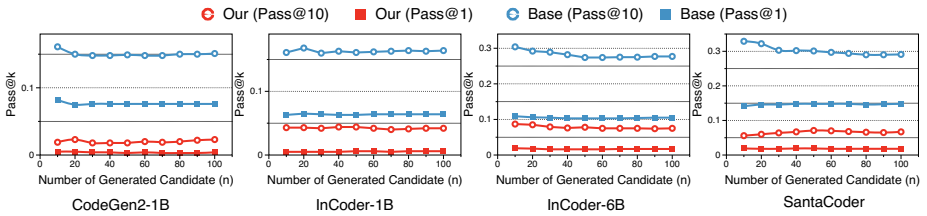

Fig. 5. The effectiveness of PPM with different numbers of generated candidates (n).

> Answers to **RQ2**: PPM can generate problems that reduce code generation model performance by modifying the problem semantics, a feature not found in existing methods.

6.3 RQ3: Naturalness

Results. The naturalness results are presented in Fig. 4. From left to right are the results of *perplexity*, *number of IDE warnings*, and *human scores*. The perplexity results demonstrate that our approaches outperform the baseline methods, as evidenced by their consistently lower perplexity values on both the *HumanEval* dataset and the *MBPP* dataset. Interestingly, for the *MBPP* dataset, our approaches even show an improvement in the naturalness of problem descriptions compared to the original descriptions. This can be attributed to the specific format of the problem descriptions in the *MBPP* dataset, which follows the pattern of "Write a function to execute some command." Our added descriptions conform to the style of these commands, resulting in a more cohesive and natural overall description. The original short problem descriptions, when combined with our added descriptions, yield lower perplexity values, indicating an increase in naturalness and coherence. For the IDE warning metric, it is observed that `Token Mutation`, `Character Mutation`, and `FuncName Mutation` introduce a significant number of IDE warnings. This can be attributed to the inherent nature of these methods, which naturally introduce typos into the prompt. These typos are easily identified by the IDE, rendering these methods impractical and unrealistic. We also observe three IDE warnings for `PPM-T` when applied to the *Human-Eval* dataset. Upon manual examination of these warnings, we determined that they are because of a specific operator within PPM. This operator is designed to transform one string into another based on ASCII values. However, during the transformation of the string in the demo part of the prompt, the resulting output string may not be an English word. Consequently, the IDE detects these transformations as typos and thus these warnings are false positives. For human score results. In line with the previous results, our approaches consistently achieved higher human scores compared to the baseline methods of token mutation and char mutation on both the *HumanEval* dataset and the *MBPP* dataset. This is because PPM focuses on generating natural and contextually relevant problem descriptions, resulting in higher human ratings for the quality and fluency of the generated descriptions.

> Answers to **RQ3**: Based on our quantitative evaluation and human study, we conclude that PPM can generate natural and realistic programming problems.

6.4 RQ4: Stability

Benchmarking Stability. The stability results from multiple randomized trials of PPM are showcased in Table 7. Each trial's *Pass@k* values are presented alongside their corresponding averages and standard variances. Notably, all average values surpass three times the variance, implying a high

Table 7. The stability results of multiple random trials.

Dataset	Methods	Trial ID	CodeGen-2b			CodeGen2-3.7b			Incoder-6b			Santacoder-1.1b		
			Pass@1	Pass@10	Pass@100	Pass@1	Pass@10	Pass@100	Pass@1	Pass@10	Pass@100	Pass@1	Pass@10	Pass@100
Human-Eval	PPM-V	1	0.03	0.10	0.21	0.01	0.06	0.12	0.02	0.08	0.17	0.02	0.06	0.12
		2	0.03	0.10	0.19	0.01	0.06	0.14	0.01	0.05	0.11	0.02	0.07	0.14
		3	0.02	0.10	0.18	0.01	0.05	0.11	0.02	0.07	0.13	0.02	0.07	0.14
		4	0.02	0.09	0.17	0.01	0.04	0.11	0.01	0.06	0.17	0.02	0.07	0.16
		5	0.02	0.10	0.25	0.00	0.03	0.09	0.02	0.08	0.16	0.02	0.07	0.16
		Avg ± Std	0.02 ± 0.003	0.10 ± 0.005	0.20 ± 0.032	0.01 ± 0.005	0.05 ± 0.011	0.11 ± 0.020	0.02 ± 0.005	0.07 ± 0.016	0.15 ± 0.027	0.02 ± 0.003	0.07 ± 0.006	0.15 ± 0.016
	PPM-T	1	0.01	0.07	0.16	0.01	0.03	0.08	0.01	0.04	0.11	0.01	0.04	0.12
		2	0.02	0.06	0.16	0.00	0.03	0.07	0.01	0.04	0.12	0.01	0.04	0.11
		3	0.02	0.07	0.20	0.01	0.04	0.10	0.01	0.03	0.09	0.02	0.06	0.16
		4	0.01	0.06	0.19	0.01	0.04	0.09	0.01	0.03	0.07	0.01	0.05	0.12
		5	0.02	0.07	0.18	0.01	0.03	0.06	0.01	0.04	0.12	0.01	0.05	0.14
		Avg ± Std	0.01 ± 0.003	0.07 ± 0.004	0.18 ± 0.018	0.01 ± 0.002	0.03 ± 0.007	0.08 ± 0.014	0.01 ± 0.002	0.04 ± 0.003	0.10 ± 0.020	0.01 ± 0.004	0.05 ± 0.009	0.13 ± 0.018
MBPP	PPM-V	1	0.03	0.17	0.36	0.01	0.07	0.19	0.02	0.10	0.28	0.03	0.15	0.34
		2	0.03	0.17	0.37	0.01	0.06	0.17	0.02	0.10	0.27	0.04	0.14	0.30
		3	0.04	0.18	0.37	0.01	0.07	0.20	0.02	0.10	0.28	0.03	0.15	0.34
		4	0.03	0.16	0.39	0.01	0.07	0.21	0.02	0.10	0.27	0.03	0.14	0.36
		5	0.04	0.18	0.37	0.01	0.07	0.18	0.02	0.11	0.27	0.03	0.14	0.32
		Avg ± Std	0.04 ± 0.003	0.17 ± 0.008	0.37 ± 0.011	0.01 ± 0.002	0.07 ± 0.007	0.19 ± 0.015	0.02 ± 0.001	0.10 ± 0.004	0.27 ± 0.006	0.03 ± 0.002	0.14 ± 0.006	0.33 ± 0.020
	PPM-T	1	0.06	0.22	0.39	0.03	0.12	0.24	0.02	0.10	0.25	0.04	0.14	0.31
		2	0.06	0.21	0.38	0.03	0.12	0.22	0.02	0.10	0.26	0.04	0.14	0.30
		3	0.07	0.23	0.41	0.03	0.11	0.21	0.02	0.09	0.24	0.04	0.15	0.29
		4	0.06	0.21	0.40	0.03	0.11	0.24	0.03	0.11	0.25	0.04	0.16	0.30
		5	0.06	0.22	0.40	0.03	0.11	0.25	0.02	0.10	0.27	0.03	0.14	0.29
		Avg ± Std	0.06 ± 0.004	0.22 ± 0.007	0.39 ± 0.011	0.03 ± 0.001	0.11 ± 0.003	0.23 ± 0.016	0.02 ± 0.002	0.10 ± 0.007	0.25 ± 0.011	0.04 ± 0.003	0.14 ± 0.007	0.30 ± 0.008

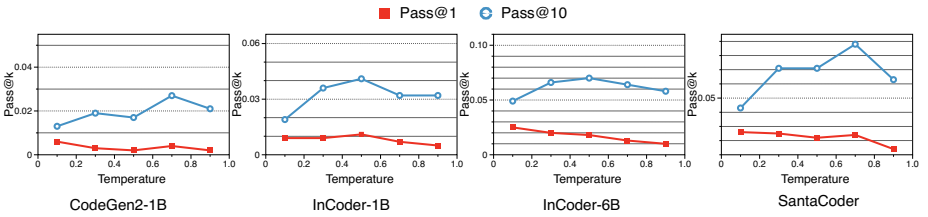

Fig. 6. The effectiveness of PPM under different temperatures.

degree of stability in the results. These results serve as confirmation that despite the introduction of randomness in the problem generation process, PPM consistently provides stable results.

Hyperparameter Stability. The results, obtained by varying hyperparameters, are illustrated in Fig. 5 and Fig. 6. Specifically, Fig. 5 demonstrates the *Pass@k* across different *n* values, while Fig. 6 displays the *Pass@k* for varying inference temperatures. In both result sets, it is evident that PPM consistently and significantly diminishes the performance of the code generation model across a diverse range of *n* settings and temperature configurations.

> Answers to **RQ4**: PPM consistently delivers stable benchmarking results across random trials and a wide array of hyperparameters.

7 THREATS TO VALIDITY

Internal Threat. Our primary internal concern revolves around the absence of a ground truth for assessing problem naturalness. To tackle this, we adopt a comprehensive approach, utilizing three evaluation metrics: *perplexity* to assess real-world relevance, *IDE warnings* to identify typos and grammar issues, and human evaluations. Additionally, we verify the real-world relevance of our generated problems by evaluating LCGMs' accuracy in solving the newly introduced lambda programming problems. These results confirm the natural and realistic nature of our lambda problems. Then if the seed problem is a natural one, our newly generated problem would be natural. Another concern pertains to how randomness affects long-term data integrity. we alleviate this threat by the following efforts. First, as PPM consistently provides stable results across multiple trials, thus, developers may choose not to publish the specific dataset on the Internet, but rather focus on publishing the method. With the public availability of the method, we simulate the probability of

problem repetition, observing minimal recurrence. Moreover, after 100 PPM attempts, approximately 70% of the problems remain distinct. Additionally, PPM offers configurability through a private "offset" parameter, making it challenging for others to replicate problems by keeping this space publicly inaccessible.

External Threat. Our external concern pertains to the selection of experimental subjects, such as datasets, models, and baselines. We aim to mitigate this concern through the following measures: (1) The chosen datasets and models are highly popular and extensively utilized in related research. (2) These datasets and models encompass diverse model types, model training algorithms, and data complexities, offering a broad spectrum of variation. (3) The selected baseline methods include almost all existing type of perturbations. Hence, while specific data might vary for other subjects, our experimental conclusions should generally remain valid due to this comprehensive selection.

8 RELATED WORK

Code Models for Software Engineering. In §2, we discussed code generation models, and now we introduce other code models for software engineering applications. Code representation models like code2vec [3] and code2seq [2] leverage syntax and structure, performing well in downstream tasks [5, 12]. Recently, pre-trained NLP models like BERT [15] and GPT-3 [7] demonstrated strong transferability to Programming Languages (PL), outperforming code2vec and code2seq [16]. This success has popularized pre-trained code models, benefiting diverse tasks [8, 17, 19, 21, 31, 34]. CuBERT [21] and C-BERT [8] use BERT architecture, specifically trained on Python and C source code, respectively. However, their single-language training limits applicability in diverse scenarios.

Robustness Evaluation for Code Model. Recently, several attack algorithms [13, 14, 20, 25, 25, 32, 38, 38] have been proposed to assess the robustness of code models. However, most of these existing methods [6, 9, 22, 35, 36, 39] primarily target classification code models. For instance, Yefet et al. presented DAMP [37], a white-box attack technique that adversarially alters variables in code using gradient information from the victim model. However, the evaluation of robustness in pre-trained code generation models has received limited research attention.

9 CONCLUSION

This paper introduces PPM, a novel approach for benchmarking LCGMs by merging programming problems to create diverse datasets automatically. PPM enhances dataset diversity and ensures long-term data integrity by accepting random input values for each trial. Additionally, it employs lambda programming tasks to maintain coherent and linguistically accurate descriptions. Extensive experiments demonstrate PPM's superiority in challenging code generation models, showcasing exceptional diversity and naturalness in the generated problems.

10 REPLICATION PACKAGE

Our code and data are available at https://github.com/SeekingDream/PPM

ACKNOWLEDGMENTS

This work was partially supported by NSF grants CCF-2146443, CCF-2008905, CNS-2135625, CPS-2038727, CNS Career 1750263, and a Darpa Shell grant.

REFERENCES

[1] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Muñoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy-Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu,

Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. 2023. SantaCoder: don't reach for the stars! *CoRR* abs/2301.03988 (2023). https://doi.org/10.48550/ARXIV.2301.03988 arXiv:2301.03988

[2] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=H1gKYo09tX

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.* 3, POPL (2019), 40:1–40:29. https://doi.org/10.1145/3290353

[4] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, and Ramesh Nallapati. 2023. Multi-lingual Evaluation of Code Generation Models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/pdf?id=Bo7eeXm6An8

[5] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR* abs/2108.07732 (2021). arXiv:2108.07732 https://arxiv.org/abs/2108.07732

[6] Hezekiah J. Branch, Jonathan Rodriguez Cefalu, Jeremy McHugh, Leyla Hujer, Aditya Bahl, Daniel del Castillo Iglesias, Ron Heichman, and Ramesh Darwishi. 2022. Evaluating the Susceptibility of Pre-Trained Language Models via Handcrafted Adversarial Examples. *CoRR* abs/2209.02128 (2022). https://doi.org/10.48550/ARXIV.2209.02128 arXiv:2209.02128

[7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. 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*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[8] Luca Buratti, Saurabh Pujar, Mihaela A. Bornea, J. Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, and Giacomo Domeniconi. 2020. Exploring Software Naturalness through Neural Language Models. *CoRR* abs/2006.12641 (2020). arXiv:2006.12641 https://arxiv.org/abs/2006.12641

[9] Earl T Campbell. 2019. A theory of single-shot error correction for adversarial noise. *Quantum Science and Technology* 4, 2 (2019), 025006.

[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[11] Simin Chen, Hanlin Chen, Mirazul Haque, Cong Liu, and Wei Yang. 2023. The Dark Side of Dynamic Routing Neural Networks: Towards Efficiency Backdoor Injection. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2023, Vancouver, BC, Canada, June 17-24, 2023*. IEEE, 24585–24594. https://doi.org/10.1109/CVPR52729.2023.02355

[12] Simin Chen, Hamed Khanpour, Cong Liu, and Wei Yang. 2022. Learning to Reverse DNNs from AI Programs Automatically. *CoRR* abs/2205.10364 (2022). https://doi.org/10.48550/ARXIV.2205.10364 arXiv:2205.10364

[13] Simin Chen, Cong Liu, Mirazul Haque, Zihe Song, and Wei Yang. 2022. NMTSloth: understanding and testing efficiency degradation of neural machine translation systems. 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*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 1148–1160. https://doi.org/10.1145/3540250.3549102

[14] Simin Chen, Zihe Song, Mirazul Haque, Cong Liu, and Wei Yang. 2022. NICGSlowDown: Evaluating the Efficiency Robustness of Neural Image Caption Generation Models. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2022, New Orleans, LA, USA, June 18-24, 2022*. IEEE, 15344–15353. https://doi.org/10.1109/CVPR52688.

2022.01493

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/V1/N19-1423

[16] Zishuo Ding, Heng Li, Weiyi Shang, and Tse-Hsun Peter Chen. 2022. Can pre-trained code embeddings improve model performance? Revisiting the use of code embeddings in software engineering tasks. *Empir. Softw. Eng.* 27, 3 (2022), 63. https://doi.org/10.1007/S10664-022-10118-5

[17] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. 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 (Findings of ACL, Vol. EMNLP 2020)*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, 1536–1547. https://doi.org/10.18653/V1/2020.FINDINGS-EMNLP.139

[18] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/pdf?id=hQwb-lbM6EL

[19] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=jLoC4ez43PZ

[20] Jordan Henkel, Goutham Ramakrishnan, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas W. Reps. 2022. Semantic Robustness of Models of Source Code. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*. IEEE, 526–537. https://doi.org/10.1109/SANER53432.2022.00070

[21] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and Evaluating Contextual Embedding of Source Code. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event (Proceedings of Machine Learning Research, Vol. 119)*. PMLR, 5110–5121. http://proceedings.mlr.press/v119/kanade20a.html

[22] Jia Li, Zhuo Li, Huangzhao Zhang, Ge Li, Zhi Jin, Xing Hu, and Xin Xia. 2022. Poison Attack and Defense on Deep Source Code Processing Models. *CoRR* abs/2210.17029 (2022). https://doi.org/10.48550/ARXIV.2210.17029 arXiv:2210.17029

[23] Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-Level Code Generation with AlphaCode. *CoRR* abs/2203.07814 (2022). https://doi.org/10.48550/ARXIV.2203.07814 arXiv:2203.07814

[24] Antonio Mastropaolo, Luca Pascarella, Emanuela Guglielmi, Matteo Ciniselli, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota. 2023. On the Robustness of Code Generation Techniques: An Empirical Study on GitHub Copilot. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2149–2160. https://doi.org/10.1109/ICSE48619.2023.00181

[25] Yixin Nie, Adina Williams, Emily Dinan, Mohit Bansal, Jason Weston, and Douwe Kiela. 2020. Adversarial NLI: A New Benchmark for Natural Language Understanding. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault (Eds.). Association for Computational Linguistics, 4885–4901. https://doi.org/10.18653/V1/2020.ACL-MAIN.441

[26] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. CodeGen2: Lessons for Training LLMs on Programming and Natural Languages. *CoRR* abs/2305.02309 (2023). https://doi.org/10.48550/ARXIV.2305.02309 arXiv:2305.02309

[27] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/pdf?id=iaYcJKpY2B_

[28] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023). https://doi.org/10.48550/ARXIV.2303.08774 arXiv:2303.08774

[29] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[30] Rylan Schaeffer. 2023. Pretraining on the Test Set Is All You Need. *CoRR* abs/2309.08632 (2023). https://doi.org/10.48550/ARXIV.2309.08632 arXiv:2309.08632

[31] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. IntelliCode compose: code generation using transformer. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1433–1443. https://doi.org/10.1145/3368089.3417058

[32] Boxin Wang, Chejian Xu, Shuohang Wang, Zhe Gan, Yu Cheng, Jianfeng Gao, Ahmed Hassan Awadallah, and Bo Li. 2021. Adversarial GLUE: A Multi-Task Benchmark for Robustness Evaluation of Language Models. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, Joaquin Vanschoren and Sai-Kit Yeung (Eds.). https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/335f5352088d7d9bf74191e006d8e24c-Abstract-round2.html

[33] Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, Dan Roth, and Bing Xiang. 2023. ReCode: Robustness Evaluation of Code Generation Models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, 13818–13843. https://doi.org/10.18653/V1/2023.ACL-LONG.773

[34] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. 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*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, 8696–8708. https://doi.org/10.18653/V1/2021.EMNLP-MAIN.685

[35] Zhou Yang, Jieke Shi, Junda He, and David Lo. 2022. Natural Attack for Pre-trained Models of Code. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1482–1493. https://doi.org/10.1145/3510003.3510146

[36] Zhou Yang, Bowen Xu, Jie M. Zhang, Hong Jin Kang, Jieke Shi, Junda He, and David Lo. 2023. Stealthy Backdoor Attack for Code Models. *CoRR* abs/2301.02496 (2023). https://doi.org/10.48550/ARXIV.2301.02496 arXiv:2301.02496

[37] Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial examples for models of code. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 162:1–162:30. https://doi.org/10.1145/3428230

[38] Yu Zhou, Xiaoqing Zhang, Juanjuan Shen, Tingting Han, Taolue Chen, and Harald C. Gall. 2022. Adversarial Robustness of Deep Code Comment Generation. *ACM Trans. Softw. Eng. Methodol.* 31, 4 (2022), 60:1–60:30. https://doi.org/10.1145/3501256

[39] Terry Yue Zhuo, Zhuang Li, Yujin Huang, Fatemeh Shiri, Weiqing Wang, Gholamreza Haffari, and Yuan-Fang Li. 2023. On Robustness of Prompt-based Semantic Parsing with Large Pre-trained Language Model: An Empirical Study on Codex. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2023, Dubrovnik, Croatia, May 2-6, 2023*, Andreas Vlachos and Isabelle Augenstein (Eds.). Association for Computational Linguistics, 1090–1102. https://doi.org/10.18653/V1/2023.EACL-MAIN.77

Received 2023-09-29; accepted 2024-01-23