DataRecipe — How to Cook the Data for CodeLLM?

Kisub Kim
Independent Researcher
Hong Kong
falconlk00@gmail.com

Jounghoon Kim
HKUST
Hong Kong
jkimcv@connect.ust.hk

Byeongjo Park
Chungbuk National University
South Korea
byeongjo06@gmail.com

Dongsun Kim*
Korea University
South Korea
darkrsw@korea.ac.kr

Chun Yong Chong
Monash University
Malaysia
chong.chunyong@monash.edu

Yuan Wang
Independent Researcher
Hong Kong
zhongykau@gmail.com

Tiezhu Sun
SnT, University of Luxembourg
Luxembourg
tiezhu.sun@uni.lu

Daniel Tang
SnT, University of Luxembourg
Luxembourg
xunzhu.tang@uni.lu

Jacques Klein
SnT, University of Luxembourg
Luxembourg
jacques.klein@uni.lu

Tegawendé F. Bissyandé
SnT, University of Luxembourg
Luxembourg
tegawende.bissyande@uni.lu

ABSTRACT

Despite the proliferation of language models, a lack of transparency persists regarding the training datasets used. Security concerns are often cited, but identifying high-quality training data is crucial for optimal model performance. Yet, while significant efforts have been made to improve model performance, dataset quality remains an under-explored area. Our study addresses this gap by comprehensively investigating data-quality properties and processing strategies used to train code generation models. We focus on identifying dataset features that impact model performance and leverage these insights to optimize datasets and enhance model efficacy. Our approach involves a multifaceted analysis encompassing metadata, statistics, data quality issues, semantic correlations between intent and code, and design choices. By manipulating these features, we explore their influence on model performance. Our findings reveal that dataset design choices significantly impact the performance of code generation models. Additionally, semantic correlations between intent and code can also affect performance, although to varying degrees.

*Corresponding author.

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10…$15.00
https://doi.org/10.1145/3691620.3695593

CCS CONCEPTS

• **Software and its engineering** → **Automatic programming**; • **Computing methodologies** → **Information extraction**.

1 INTRODUCTION

The recent evolution of Code Large Language Models (CodeLLMs) has significantly advanced the capabilities of automated code-related tasks [15, 20, 81]. They now play a pivotal role not only in code generation but also in many software engineering tasks. For example, CodeLLMs help software development processes [86] and debugging techniques [74].

The effectiveness of these models tends to be highly dependent upon the quality of the datasets used for training [6]. Poor-quality datasets, characterized by inconsistencies in code formatting, incomplete or erroneous annotations, and lack of diversity in coding patterns, can lead to models that are less robust and more prone to generating inaccurate or less-optimized code. As these models are increasingly integrated into critical software engineering workflows, the need for *high-quality*, reliable datasets becomes paramount.

However, it is not clearly revealed how to process the datasets for CodeLLMs. In particular, the dataset processing methods for closed-sourced models such as the series of ChatGPT [52] and Claude [8], are never identified to gain a competitive edge in the market. Unfortunately, without stringent standards for dataset compilation and preprocessing, the transformative potential of CodeLLMs in revolutionizing software practices remains only partially realized.

The current benchmarking studies focus only on evaluating and comparing CodeLLMs rather than analyzing the impact of the datasets used for the models. Several researchers have studied and established benchmarks for various code-related tasks, such as code search [71], code review [41, 76], and code understanding [46]. Primarily, these datasets are carefully curated from *high-quality*

GitHub repositories, characterized by a significant number of stars and robust developer engagement, or from active coding communities such as StackOverflow. Although the existing studies are comprehensive and concrete, they did not explore how the properties of the datasets affect the performance of the models.

This study has been meticulously designed to assess the impact of dataset properties used in training CodeLLMs, with a specific focus on text-to-code generation tasks. Initially, we establish baseline performances through zero-shot experiments. Subsequently, our study undertakes a fine-tuning process using multiple code generation datasets to determine which dataset most effectively enhances performance.

After we demonstrate which features potentially affect the model performance by showing the characteristics (i.e., metadata, statistics, static analysis results, semantic correlations, and design choice), we further try boosting the performance through dataset revisions. The revision process is structured at three levels of granularity: Code-level, which refines the source code using automated tools; Pair-level, which enhances the semantic correlations between textual prompts and code; and Dataset-level, which standardizes the design of datasets. This granular approach allows for targeted improvements that are anticipated to bridge the gap between dataset quality and practical model utility.

To achieve the objectives, we formulate three research questions as follows:

- **RQ1**: Can automating code-level refactoring on dataset enhance model performance?
- **RQ2**: How does refining pair-level semantic correlations on dataset impact model performance?
- **RQ3**: Does aligning dataset-level design on dataset amplify model performance?

Our experimental results unveil several significant insights:

- The impact of datasets varies based on the size of the models, resulting in diverse performances across different benchmarks.
- Each dataset presents unique statistics, including token count, line count, readability, and code-related issues, alongside distinct semantic correlations and data design choices.
- Automated refactoring has the potential to enhance code-side quality, influencing model performance contingent upon the characteristics of both the model and the benchmark, but the impact highly varies.
- Correcting semantic correlation generally enhances performance on Functionality-focused benchmarks, although the impact is not significant while exhibiting neutral effects on Alignment-focused ones.
- The dataset design may be a primary consideration for performance enhancement, as it consistently and significantly improves performance across all model-benchmark combinations.

2 BACKGROUND AND MOTIVATION

Effective code generation models rely fundamentally on the quality and structuring of their training datasets. This section outlines the critical factors and evaluation metrics of datasets specifically designed for these models.

2.1 Datasets for Code Generation

Source code datasets are utilized to train language models, providing a rich and diverse array of examples that help these models grasp both context and coding nuances. Datasets typically comprise of following aspects: intent, code snippets, test cases, and metadata. An intent is a concise description or specific question in a natural language that guides the LLMs in understanding the required type of code generation. It covers various domains, such as data manipulation and algorithmic problem-solving. A code snippet should be syntactically and functionally correct, representing a practical implementation of the given text. These snippets may vary in abstraction, from methods, classes, and projects. Test cases play a crucial role in assessing the efficacy of LLMs in code generation. They ensure the generated code not only compiles but also executes correctly and meets the requirements established by the prompts. Metadata within the code generation dataset provides supplementary information that, while not directly relevant to training or evaluation, aids in managing and understanding the dataset.

2.2 Quality of Code Generation Dataset

Ensuring data quality is of paramount importance in the training of language models, particularly those tailored for code generation tasks. Despite its critical role, a universally accepted definition of data quality remains elusive, as does a standardized set of criteria for identifying *high-quality* datasets suitable for training such models [60]. Nevertheless, established methodologies, such as leveraging human annotations and heuristics, are commonly employed for evaluating the quality of individual components, including natural language intent and source code.

Despite advancements in automated quality assessment techniques, the overall validation of data quality still heavily relies on the expertise and intuition of analysts, particularly in large technology companies like Google [60]. In the field of natural language processing, researchers commonly employ various metrics and techniques to gauge the quality of text. These may include measures of coherence [50], fluency [55], and semantic relevance [89], among texts. Conversely, assessing the quality of source code involves distinct methodologies, such as simplicity measure [70], static code analysis [54], code review [32], and unit testing [25]. Empirical studies [24, 28, 53] have demonstrated that factors such as the presence of bugs, code smells, and software complexity significantly influence program stability and behavior.

3 STUDY DESIGN AND PRELIMINARIES

In this section, we outline the rationale behind our target task with a formal definition, discuss the selection of training and test datasets (Section 3.1), and detail our choice of language models (Section 3.2). We then introduce how we measure the performance of the models (Section 3.3), providing implementation details (Section 3.4). Additionally, we conduct full-parameter fine-tuning to explore which dataset can optimize the performance of the existing models most effectively (Section 3.5). Finally, we conclude the section by outlining the research questions addressed in this paper (Section 3.7).

3.1 Target Task, Training, and Test Datasets

We center our attention on the task of code generation, a process wherein source code artifacts are created automatically, drawing from given input specifications or contexts. We selected such a task as our target as follows:

(1) Code generation has gained notable interest due to advancements in LLMs [15, 16, 52, 68].
(2) It produces executable code, enabling straightforward evaluation through metrics like correctness [15, 34], syntactic accuracy [62], and semantic relevance [55, 62].
(3) Code generation models need diverse input pairs of code snippets and natural language, highlighting the importance of data quality across domains and styles [15].
(4) As a key part of software development, code generation enhances practices like synthesis and documentation, linking data quality studies to practical improvements [20].

Specifically, for an LLM M_Θ with parameters Θ and a corpus $X = x_1, ..., x_n$, the language modeling loss for optimization is:

$$L(X) = \sum_i \log P(x_i \mid x_{i-k}, ..., x_{i-1}; \Theta) \quad (1)$$

The model takes a concatenation of the prompt and the ground truth as input and predicts each token x_i in an autoregressive manner, given the preceding tokens $x_{i-k}, ..., x_{i-1}$.

We have chosen our target datasets based on their widespread use in the field of code generation. Each dataset consists of intent-code pairs, which are pivotal for training language models to understand and generate programming code. Specifically, we utilize the following datasets:

- **MBPP**$_{train}$ [9]: This dataset is designed for training and evaluating code generation models. It comprises 374 training and 500 test instances of diverse intent-code pairs of fundamental programming challenges that primarily focus on foundational programming concepts. It is widespread use and recognition stem from its structured collection of problems that systematically cover essential programming skills.
- **CoNaLa**$_{train}$ [87]: This dataset represents a valuable resource for researchers and practitioners because of its unique composition and origins. The dataset consists of 2,880 curated pairs of instructions and code snippets, sourced from Stack Overflow. It is divided into 2,380 instances for training and 500 for testing, and it encapsulates real-world programming challenges and solutions encountered by developers across diverse domains.
- **CodeAlpaca**$_{train}$ [14]: Comprising a curated collection of 2,190 programming problems and corresponding solutions for training. It offers a unique opportunity to explore the intersection of natural language understanding and software engineering. The dataset comprises longer and more intricate examples compared to CoNaLa, enabling a more thorough evaluation of models on code generation.
- **DS-1000**$_{train}$ [35]: This is a specialized code generation benchmark that consists of 897 data science problems, addressing diverse and realistic use cases across seven Python libraries. It is designed to avoid memorization by implementing perturbations in the problem statements. We include this dataset as one of our target datasets despite its primary purpose being test. Our objective is to maximize diversity for a comprehensive study.
- **ODEX**$_{train}$ [78]: This is an open-domain, multilingual, execution-based dataset designed for text-to-code generation. We only leverage 439 English text and source code pairs. It covers a broad range of 45 unique libraries that are harvested from StackOverflow to reflect practical coding queries. While this dataset is designed to test, the same amount of training set also exists.

Particularly, MBPP, CoNaLa, CodeAlpaca, and ODEX datasets have distinct training and test datasets, respectively. We only target the training parts of each dataset but the entire set of DS-1000.

To evaluate the dataset quality upon fine-tuning, we design our experiments into two categories: 'Functionality-focused' and 'Alignment-focused' in terms of evaluation with existing well-known benchmarks. In the 'Functionality-focused' category, we utilize the HumanEval$_{test}$ [15], HumanEval+$_{test}$ [43], MBPP$_{test}$ [9], and MBPP+$_{test}$ [43] datasets, to assess the functional effectiveness of our models. These are the most well-known benchmarks and we only use their test datasets for benchmarking. In the 'Alignment-focused' category, we use the ODEX$_{test}$ [78] and CoNaLa$_{test}$ [87] datasets to check the semantic alignment of the generated code with the intended specifications. To avoid data leakage, we clearly distinguish the training datasets from the benchmarks above.

3.2 Target Language Models

To conduct a comprehensive analysis, our selection of Language Models (LMs) was guided by several criteria. First, we exclusively focused on open-source models, excluding closed-sourced models such as ChatGPT and Codex due to the inaccessibility of their parameters. Second, we chose models released within the past few years. Last, to investigate the impact of scaling, we selected models with a diverse range of parameters. Models with less than 2B parameters were categorized as **normal-sized LMs**, while those surpassing this threshold were classified as **LLMs**. In total, our experimental setup comprised seven models sourced from diverse families of models, ensuring a comprehensive and representative exploration of model performance.

Normal-sized LMs: We utilize the CodeGen-350M-Mono [51], StarCoder-1B [39], and DeepSeek-Coder-1.3B [27] models as normal-sized language models. CodeGen-350M-Mono is an autoregressive language model, serving as a scaled-down version of the CodeGen. StarCoder-1B combines the capabilities of natural language processing with machine learning to interpret and write code effectively, making it ideal for applications in software development. DeepSeek-Coder-1.3B, on the other hand, is tailored for deep code analysis and search functionalities. This is the smallest version of the DeepSeek family.

LLMs: We use DeepSeek-Coder-6.7B [27], MagiCoder-DS-6.7B [79], CodeLlama-7B [63], and Llama-2-7B [75]. DeepSeek-Coder-6.7B is built upon the enhancements of its predecessor, DeepSeek. This model excels at understanding and evolving user-specific coding patterns and preferences. MagiCoder-DS-6.7B distinguishes itself from other CodeLLMs by focusing on seamless integration and adaptability across a wider range of programming environments and languages. CodeLlama-7B constitutes a series of LLMs built upon the foundation of Llama-2-7B.

3.3 Metrics

We measure the *Pass@1* [15, 34] and *BLEU* [55] metrics, which provide quantitative insights into the accuracy and linguistic quality of the generated code, respectively. *Pass@1* measures the percentage of instances where the model's first attempt at generating code is correct, reflecting its precision in understanding and implementing user requirements. *BLEU*, on the other hand, originally developed for assessing the quality of machine-translated text, is adapted here to evaluate the syntactical and grammatical coherence of code comments and documentation generated by the models. While *Pass@1* is valuable for evaluating the functionality of generated code, it may not capture nuances in the quality of code generation from natural language descriptions as effectively as *BLEU* does.

3.4 Implementation Detail

For all training and inference activities associated with our seven target models, we utilize four NVIDIA A800 SXM4 80GB GPUs. Our focus is solely on full fine-tuning to precisely ascertain the impacts attributable to different datasets. Utilizing other techniques such as in-context learning [11], few-shot learning [33], or parameter-efficient fine-tuning [29] could introduce potential biases, which we aim to avoid in this study.

Our study uses the following hyperparameters. We configured the learning rate at 5×10^{-5}. The Adafactor optimizer [66] was utilized with 32-bit floating-point precision for all models. The weight decay parameter was set to 0.01. For training configurations, the maximum token length was constrained to 512, the batch size was established at 4, gradient accumulation steps at 8, and the number of warm-up steps was 500. Additionally, we set the evaluation interval such that 0.2% of the samples were seen between each model evaluation step. We select the checkpoint with the lowest evaluation loss for inference and set the number of epochs to 50 to optimize accuracy and ensure maximum performance.

We have made our code and data publicly available [1], to facilitate further research and the reproduction of our results.

3.5 Baseline Performance Exploration

We aim to discover the impact of diverse datasets on model performance by fine-tuning them in this Section. To initiate this assessment, we first establish each model's Zero-shot performance as a baseline. Then, we fine-tune the models with each dataset, keeping all hyper-parameters and the experimental environment constant. This ensures that any observed performance differences are attributable to the datasets. Table 1 presents the comprehensive code generation performance of all models across the target datasets. As outlined in Section 3.3, we utilize two distinct performance metrics: *Functionality-focused (Pass@1)* and *Alignment-focused (BLEU)*.

We fine-tune models on the target datasets under consistent hyper-parameters and experimental conditions, followed by performance evaluation to compare against the Zero-shot. This rigorous approach ensures variations in performance are solely attributable to the datasets, reflecting their impact on model performance.

Fine-tuning models with different datasets has definitely an impact on the performance of code generation. We spotlight the best Zero-shot performance with gray-colored cells in the table. Overall, DeepSeek-Coder-1.3B emerges as the top performer among normal-sized LMs, while MagiCoder-DS-6.7B leads in the LLM category. Improved fine-tuning performance of each dataset is highlighted with colored cells: green cells indicate enhancements, and red cells showcase degraded performance. Clear differences in performance between these two metrics suggest that the optimal fine-tuning dataset varies for each metric and is also influenced by model size. Specifically, MBPP_{train} is the most effective dataset for fine-tuning normal-sized LMs on the *Pass@1* metric, while DS-1000_{train} provides the greatest performance enhancement for LLMs. Additionally, the ODEX_{train} dataset is the most influential tuning dataset for both the CoNaLa_{test} and ODEX_{test} benchmarks.

Table 1: Code generation performance of target normal-sized (upper) and large language models (lower) with tuning datasets. HumanEval+ includes additional code snippets and test cases compared to HumanEval, while MBPP+ is inspired by MBPP but contains entirely different data instances.

Tuning Data	Model & Size	Functionality-focused (Pass@1)				Alignment-focused (BLEU)	
		HumanEval	HumanEval+	MBPP	MBPP+	CoNaLa	ODEX
Zero-shot	CodeGen-350M-Mono	13.41	12.80	13.80	12.80	20.69	20.32
	StarCoder-1B	12.80	12.80	23.00	12.80	26.16	31.51
	DeepSeek-Coder-1.3B	**33.53**	**28.04**	**45.20**	**28.04**	**29.95**	**33.80**
MBPP	CodeGen-350M-Mono	**14.63**	**12.20**	**15.00**	**30.08**	**19.77**	**21.44**
	StarCoder-1B	**14.63**	**13.41**	**22.60**	**36.84**	**15.85**	**17.26**
	DeepSeek-Coder-1.3B	**40.24**	**32.32**	**38.40**	**55.14**	**23.74**	**25.35**
CoNaLa	CodeGen-350M-Mono	**5.49**	**4.88**	**7.20**	**8.52**	**19.06**	**20.99**
	StarCoder-1B	7.32	**6.10**	**1.00**	**2.01**	**19.50**	**22.55**
	DeepSeek-Coder-1.3B	**18.29**	**14.63**	**29.80**	**37.84**	**30.15**	**29.21**
CodeAlpaca	CodeGen-350M-Mono	**9.76**	**9.15**	**11.40**	**14.04**	**14.88**	**13.44**
	StarCoder-1B	**14.02**	**12.20**	**12.40**	**16.79**	**13.17**	**13.90**
	DeepSeek-Coder-1.3B	**28.66**	**23.78**	**33.60**	**47.62**	**18.62**	**20.58**
DS-1000	CodeGen-350M-Mono	**9.15**	**7.93**	**14.20**	**21.55**	**17.30**	**17.86**
	StarCoder-1B	**14.02**	**12.20**	**20.00**	**27.32**	**22.25**	**25.79**
	DeepSeek-Coder-1.3B	**41.46**	**34.15**	**44.80**	**55.39**	**30.63**	**33.13**
ODEX	CodeGen-350M-Mono	**12.20**	**10.98**	**14.00**	**24.81**	**62.08**	**74.85**
	StarCoder-1B	**15.85**	12.80	**11.00**	**15.54**	**66.51**	**75.59**
	DeepSeek-Coder-1.3B	**35.98**	**29.88**	**43.40**	**56.39**	**80.06**	**95.35**
Zero-shot	DeepSeek-Coder-6.7B	47.56	36.58	57.60	36.58	**36.73**	**40.11**
	MagiCoder-DS-6.7B	**59.75**	**53.04**	**60.20**	**53.04**	32.91	37.86
	CodeLlama-7B	30.48	26.82	37.80	26.82	35.28	38.66
	Llama-2-7B	12.80	10.97	19.20	10.97	27.01	29.64
MBPP	DeepSeek-Coder-6.7B	**53.66**	**44.51**	**55.00**	**70.43**	**31.27**	**34.25**
	MagiCoder-DS-6.7B	**60.98**	**52.44**	**57.80**	**71.43**	**36.23**	**39.18**
	CodeLlama-7B	**32.32**	**27.44**	**38.40**	**52.38**	**33.59**	**36.75**
	Llama-2-7B	**11.58**	**9.76**	**18.00**	**32.33**	**21.47**	**21.72**
CoNaLa	DeepSeek-Coder-6.7B	**34.15**	**27.44**	**46.00**	**56.89**	**29.88**	**31.75**
	MagiCoder-DS-6.7B	**23.17**	**28.05**	**33.60**	**45.11**	**29.12**	**33.10**
	CodeLlama-7B	**28.66**	**23.78**	**20.60**	**28.82**	**18.65**	**19.15**
	Llama-2-7B	**0.00**	**0.00**	**0.00**	**0.25**	**16.32**	**16.48**
CodeAlpaca	DeepSeek-Coder-6.7B	47.56	**40.85**	**47.40**	**60.65**	**25.00**	**28.14**
	MagiCoder-DS-6.7B	**45.73**	**40.24**	**47.20**	**60.40**	**26.69**	**29.24**
	CodeLlama-7B	**28.05**	**23.78**	**29.00**	**38.35**	**17.79**	**21.29**
	Llama-2-7B	**6.71**	**6.10**	**3.00**	**4.51**	**10.29**	**11.81**
DS-1000	DeepSeek-Coder-6.7B	**51.83**	**43.29**	**56.20**	**72.93**	**33.87**	**37.22**
	MagiCoder-DS-6.7B	**61.59**	**54.88**	**58.60**	**72.18**	**30.41**	**34.38**
	CodeLlama-7B	**33.84**	**29.12**	**38.40**	**53.13**	**31.93**	**35.22**
	Llama-2-7B	**2.44**	**2.44**	**3.40**	**4.76**	**23.97**	**25.32**
ODEX	DeepSeek-Coder-6.7B	47.56	**40.24**	57.60	**70.93**	**80.56**	**95.76**
	MagiCoder-DS-6.7B	**64.02**	**56.71**	**59.00**	**73.93**	**78.64**	**93.74**
	CodeLlama-7B	**32.32**	26.82	**36.20**	**48.12**	**63.99**	**74.92**
	Llama-2-7B	**6.71**	**4.88**	**8.00**	9.27	**58.58**	**67.87**

The first column represents all training datasets ($*_{train}$) used in this study. The second row lists up all test datasets ($*_{test}$) used in this study. The values in gray (nn.nn) denote the best Zero-shot performance while green (nn.nn) and red (nn.nn) indicate the improved and degraded performance scores, respectively. Color saturation represents the magnitude of the differences between the best Zero-shot and the improved performance.

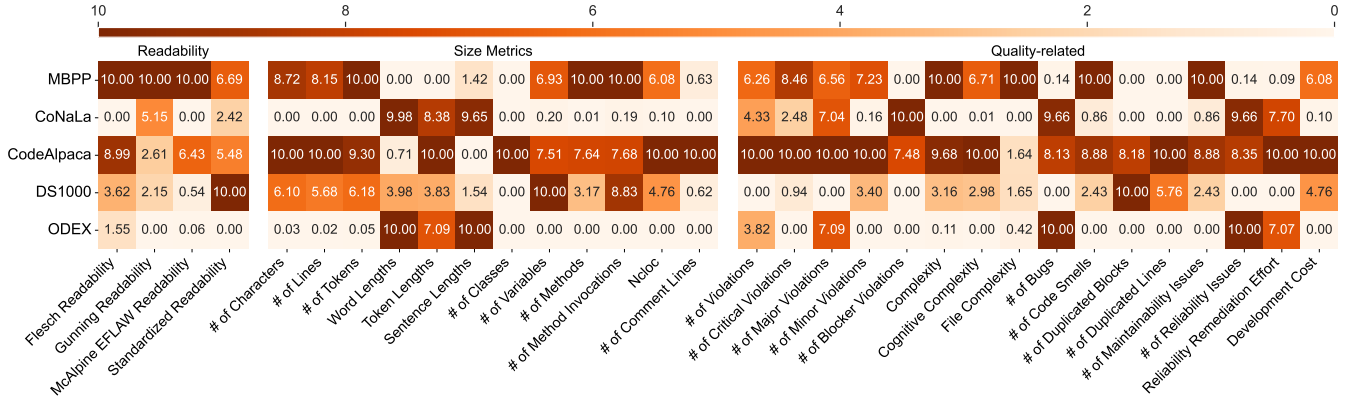

Figure 1: Characteristic exploration of the target training datasets. There are three groups of dataset characteristics: readability, size metrics, and quality-related. The initial values are extracted by using SonarQube, a well-known static analyzer. Such values are calculated as frequencies such as $(1+numberOfConditionalBranches)/numberofrecords$ which is theoretically indefinite. Consequently, we normalize these values to fit into the range of [0.00, 10.00].

The varying impact of fine-tuning with diverse datasets on model performance suggests the potential benefits of delving deeper into dataset quality assessment. First, while most datasets contribute to improved performance, these enhancements are specifically observed within the MBPP+$_{test}$ benchmark, suggesting that their efficacy may be limited to certain types of code generation tasks. A notable exception is the CoNaLa$_{train}$ dataset, which does not enhance code generation performance in most cases, suggesting a potential misalignment with the tasks or model architectures used. Second, an unexpected outcome is that our only foundational model, Llama-2-7B, does not outperform our smallest model, CodeGen-350M-Mono, highlighting that larger model size does not necessarily equate to better performance and may depend heavily on specific model-dataset combinations.

Performance Exploration: Depending on the model size, the impact of each dataset varies. The MBPP$_{train}$ and DS-1000$_{train}$ datasets are the most effective in enhancing code generation performance for the category of normal-sized LMs and LLMs, respectively. The ODEX$_{train}$ dataset is the most influential tuning dataset for both the CoNaLa$_{test}$ and ODEX$_{test}$ (i.e., Alignment-focused metrics).

3.6 Characteristic Exploration

This section aims to explore features of datasets that might have an impact on a model's code generation performance. Specifically, we examine the three groups of dataset characteristics as follows: 1) statistical attributes, including features from static analysis, 2) semantic correlation, and 3) dataset design choices.

3.6.1 Statistical Attributes. Statistical attributes provide quantitative insights into the dataset [47], highlighting its diversity and scope. These statistics encompass basic features such as the number of samples, as well as more detailed attributes detected by a static analyzer. Basic features include the # of Characters, Lines (Ncloc), Tokens, Classes, Methods, Method Invocations, Variables, and Comment Lines. These features are calculated as average-based for each dataset. We also measure the various readability scores on the data points as it could ensure that the content is accessible and understandable to the intended LMs [31]. Additionally, features identified by a static analyzer provide deeper insights into the source code quality essential for evaluating the practicability of the generated code. To conduct static analysis, we utilize SonarQube [2] to detect issues from Bugs to Reliability Issues. The values of statistical attributes are listed in Figure 1. The values are normalized, average-based scores ranging from 0 to 10.

Table 2: Semantic correlation scores (range: [0.00,100.00]) of the target training datasets.

Dataset	Semantic Correlation
MBPP$_{train}$	83.94
CoNaLa$_{train}$	91.71
CodeAlpaca$_{train}$	81.74
DS-1000$_{train}$	79.92
ODEX$_{train}$	91.08

Dataset characteristics may correlate with model performance. The high readability of the MBPP$_{train}$ dataset, as shown in Figure 1, may contribute to achieving the best performance in *Functionality-focused* tasks when models are fine-tuned with MBPP$_{train}$ (See Table 1). Attributes relevant to violations and smells have an impact on the performance for the *Alignment-focused* tasks. The ODEX$_{train}$ dataset has a relatively lower number of violations and smells, while its number of bugs and reliability issues are higher than other datasets though, and it shows better performance when the models are fine-tuned with the datasets.

3.6.2 Semantic Correlation. Semantic correlation is the relationship between the intent written in natural language and the corresponding code within the dataset. To quantify these semantic correlations, we constructed a classifier by fine-tuning the DeepSeek-Coder-6.7B-Instruct model [27], a variant of the DeepSeek-Coder series, specifically responsive to instruction-based prompts. First, we fine-tune the model with a specifically curated dataset for intent-code semantic correlation. To build such a dataset, we leveraged GPT-4 [3], following a previous study [26]. We obtained 742 paired data points. The first four authors cross-checked the GPT-generated fine-tuning data to ensure the semantic correlation quality and we conducted de-duplication for data leakage against the benchmarks. Since labeled data was not available, we created negative samples

by randomly pairing code snippets with unrelated intents. Table 2 presents the semantic correlation scores for each dataset as measured by our fine-tuned model. The scores range from 0 to 100.

Our exploration cannot identify a significant relationship with semantic correlation with respect to model performance. The CoNaLa$_{train}$ and ODEX$_{train}$ datasets exhibit relatively high scores of 91.74 and 91.08, respectively, compared to the scores of 83.94 for MBPP$_{train}$, 81.74 for CodeAlpaca$_{train}$, and 79.92 for DS-1000$_{train}$. However, the higher numbers directly have an impact on the model performance; ODEX$_{train}$ can improve the performance after fine-tuning but CoNaLa$_{train}$ cannot.

3.6.3 Dataset Design Choice. Dataset design refers to the structured relationship and alignment between NL intents and their corresponding code within datasets [87], which varies across different datasets. Each dataset adopts a unique design based on the specific requirements of the coding tasks it aims to support and different organizations have distinct design preferences [4]. It may influence how effectively a model can learn and generalize from the training data to benchmarks or even to real-world applications.

We manually check the design differences of our target datasets. MBPP$_{train}$ features specific one-line natural language intent like, *Write a Python program that does ...*, followed by complete Python methods, enhancing the model's ability to parse and execute detailed programming instructions. CodeAlpaca$_{train}$ introduces a wide variety of designs. For example, comprehensive API usage, simple tasks like finding ASCII values, or constructing dictionaries. DS-1000$_{train}$ features detailed intents involving complex scenarios, paired with relatively simpler, method-level Python solutions. CoNaLa$_{train}$ offers succinct one-line natural language descriptions paired with equivalent one-line code snippets, sharpening the model's capacity for direct translation of brief queries into executable code. ODEX$_{train}$ also has simple descriptive intents and corresponding snippet-based Python solutions, such as adding specific arguments to a parser while providing multiple options. Figure 2 presents actual examples of different designs across all the target datasets.

Characteristic Exploration: The datasets selected for this study have wide ranges of attribute values. While some statistical attributes are slightly correlated with the model performance, semantic correlations have no significant contributions to the performance. Dataset design choices can also have an impact on the performance. Our exploration results motivate the research questions.

3.7 Research Questions

In this study, we focus on the following research questions:

- RQ1: **Can automating code-level refactoring on dataset enhance model performance?** Source code refactoring, as highlighted in studies [49, 57, 58], reduces errors, complexity, and dataset bias, enhancing code quality and maintainability for more efficient machine learning [42, 65]. We aim to leverage automated refactoring techniques at the code level to improve language model performance by investigating its impact on effectiveness.
- RQ2: **How does refining pair-level semantic correlations on dataset impact model performance?** Previous studies [61, 82, 83, 88] investigated correlations across various data types, including text-text, image-text, or video-text pairs, underscoring the significance of accuracy. We hypothesize that refining semantic correlations at the intent-code pair level could enrich contextual information and enhance understanding, thereby potentially improving code generation performance.
- RQ3: **Does aligning dataset-level design on dataset amplify model performance?** While considerable attention has been given to the importance of datasets' design [13, 37, 44, 69], the design alignment for training CodeLLM remains underexplored. We aim to explore the effect of aligning dataset design choices on language model performance. Specifically, we investigate whether harmonizing formatting conventions across diverse datasets can

Write a function to find the median of two sorted arrays of the same size.

```
def get_ludic(n):
    ludics = []
    for i in range(1, n + 1):
        ludics.append(i)
    ...
    return ludics
```

(a) A design example of MBPP[1].

get a list of items form nested list 'li' where third element of each item contains string 'ar'.

```
[x for x in li if 'ar' in x[2]]
```

(b) A design example of CoNaLa[2].

Create an API using Django Rest Framework to retrieve data from a PostgreSQL database table.

```
import viewsets, routers from myapp.models
import MyModel
class MyModelViewSet(viewsets.ModelViewSet):
    ...
    router = routers.DefaultRouter()
    router.register(r'mymodel', MyModelViewSet)
```

(c) A design example of CodeAlpaca[3].

Problem: How can I know the (row, column) index of the minimum of a numpy array/matrix? For example, if A = array([[1, 2], [3, 0]]), I want to get (1, 1) Thanks! A: <code> import numpy as np a = np.array([[1, 2], [3, 0]]) </code> result = ... # put solution in this variable BEGIN SOLUTION <code>.

```
import numpy as np
def generate_test_case(test_case_id):
    ...
def exec_test(result, ans):
    ...
def test_execution(solution: str):
    ...
    assert exec_test(test_env["result"], expected_result)
```

(d) A design example of DS-1000[4].

convert items in 'T1' to integers.

```
def f_642154(T1):
    return tuple(map(int, T1))
...
def f_642154(T1):
    return (T1 * 1).astype(int)
```

(e) A design example of ODEX[5].

Figure 2: Design examples taken from the target datasets. The gray boxes represent the intent of each corresponding code snippet.

[1]https://huggingface.co/datasets/mbpp/viewer/full/train?row=2
[2]https://huggingface.co/datasets/neulab/conala/viewer?row=61
[3]https://huggingface.co/datasets/antolin/codealpaca-filtered/viewer?row=61
[4]https://huggingface.co/datasets/xlangai/DS-1000/viewer/default/test?p=3&row=320
[5]https://huggingface.co/datasets/mbpp/viewer/full/train?row=2

enhance model efficacy and accuracy. This hypothesis is based on the idea that inconsistencies in dataset design may impede the model's generalization and pattern recognition during training.

4 ANSWERING RESEARCH QUESTIONS

This section describes the experiment settings to answer each research question and analyzes the results of each experiment.

4.1 RQ1: Automated Refactoring

Setup: Motivated by the results shown in Section 3.6.1, our study hypothesizes that code-related issues, such as bugs, code smells, and violations of coding standards, can be correlated with the performance of the code generation models.

To confirm our hypothesis, our study inspects overall code understandability scores before and after applying code refactoring tools to the datasets. We use the Pylint library, a well-known static code analysis tool. Such a library identifies programming errors and enforces coding standards to enhance readability and maintainability. We derived the dataset-level code understandability scores by averaging the scores of all code files within each dataset, where scores range from 0 to 100, with higher scores indicating better code quality. For code refactoring, we leverage six automated tools: Autopep8[6], Black[7], YAPF[8], Autoflake[9], Docformatter[10], and Unify[11].

Results and Analyses: First, applying the six automated refactoring tools can improve the understandability of the code snippets in the datasets selected in this study, while the amount of the improvements varies for each dataset as listed in Table 3. The 'Original Score' and 'Refactored Score' represent before and after applying the refactoring tools to the datasets. Notably, the MBPP_{train} dataset experiences the most significant improvement, with code understandability scores escalating from 42.22 to 90.81, marking an impressive increase of 115%. In contrast, the impact of refactoring on the CoNaLa_{train} dataset is minimal, showing a negligible increase of only 0.01%. This minimal impact can be attributed to the structure of the CoNaLa_{train} dataset, which consists of concise one-line intents paired with one-line code snippets, leaving limited scope for substantial refactoring.

Automated refactoring contributes to performance improvement for the most of the target models this study addresses. We fine-tune each model with original training datasets and refactored training datasets, respectively, and we compare the model performance between a pair of two fine-tuned models. Specifically, the testing results on MBPP_{test} are demonstrated in Figure 3. When training models with the refactored MBPP_{train}, all models, except for Llama-2-7B, consistently exhibited improved performance compared to their fine-tuned counterparts using the original datasets. While automated refactoring generally improved the performance on the MBPP_{test} benchmark, it also involved significant trade-offs.

Table 3: Code understandability scores of original and refactored datasets.

Dataset	Original Score	Refactored Score
MBPP_{train}	42.22	90.81 (115.09%)
CoNaLa_{train}	76.62	77.10 (0.01%)
$\text{CodeAlpaca}_{train}$	70.57	81.42 (15.37%)
DS1000_{train}	87.58	95.31 (8.83%)
ODEX_{train}	72.17	73.60 (1.98%)

However, the overall performance improvement is not statistically significant. We apply the Wilcoxon signed-rank test [18] to the paired test samples (before/after refactoring) as the distribution of the samples is non-parametric. As a result, the p-value of the statistical testing is 0.986, and thus we cannot reject the null hypothesis. Specifically, training with the refactored CoNaLa_{train} led to considerable performance declines in three models StarCoder-1B, CodeLlama-7B, and Llama-2-7B, with reductions nearly reaching zero. In contrast, MagiCoder-DS-6.7B experienced a substantial increase from 33.60 to 48.80, marking an improvement of 45.24%. Finally, other datasets also demonstrated that LLMs consistently benefited from training with automatically refactored datasets rather than normal-sized LMs.

Additionally, evaluating with the other testing datasets shows similar performance improvements. For the HumanEval_{test}, scores improved in 11 out of 35 (31.43%) instances, i.e., different training data and model combinations. A similar increase was observed in HumanEval+_{test}. In the MBPP_{test}, improvements were noted at 65.71%, whereas MBPP+_{test} showed a lesser improvement rate of 45.71%. The CoNaLa_{test} and ODEX_{test} benchmarks exhibited increases of 28.57% and 34.29%, respectively. The detailed comparison results on all testing datasets are available [1].

To further interpret, different datasets exhibit varying levels of initial code quality and structure. For instance, the MBPP dataset had a significant improvement due to its initially lower code understandability scores (i.e., many functions lacked proper formatting), allowing more room for enhancement via refactoring by automated refactoring tools like Black and YAPF. In contrast, datasets like CoNaLa, which contained concise one-like code snippets, showed minimal impact as there was limited scope for refactoring. Additionally, we can also interpret the results as the simple scripts might not benefit as much from refactoring compared to relatively complex code snippets such as multi-line functions that can be further optimized. Larger models such as MagiCoder-DS-6.7B are less sensitive to improvements in code quality because they are already known to be adept at handling diverse code structures. However, smaller models like CodeGen-350M-Mono showed more significant improvements, highlighting that model size and training complexity influence how much they benefit from cleaner, more standardized code.

> **Answer to RQ1**: Automated refactoring has an impact on the performance of LMs on specific benchmarks, although the impact varies according to the dataset. The MBPP_{train} dataset, in particular, was more sensitive to code refactoring than others, demonstrating the largest increase in overall code understandability. Such findings may indicate that refactoring the source code could impact the generation performance.

[6] https://github.com/hhatto/autopep8
[7] https://github.com/psf/black
[8] https://github.com/google/yapf
[9] https://github.com/PyCQA/autoflake
[10] https://github.com/PyCQA/docformatter
[11] https://github.com/myint/unify

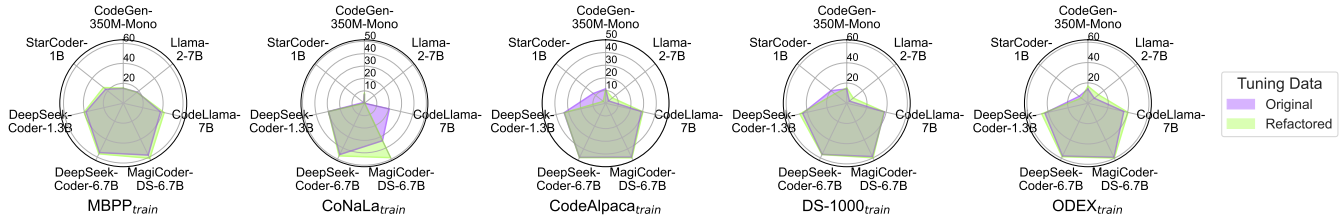

Figure 3: RQ1 — Effectiveness of the automated refactoring on the MBPP$_{test}$ benchmark with the names of training datasets listed beneath each plot. The greater the coverage of the green portion, the stronger the performance it signifies. The detailed results, including all the combinations of datasets and models, are provided in our Online Appendix [1].

4.2 RQ2: Semantic Correlational Correction

Setup: For this RQ, motivated by Section 3.6.2, we hypothesize that enhancing the semantic correlation of datasets can improve code generation performance compared to fine-tuning with the original dataset. Therefore, we compare the model performance between before and after correcting the semantic correlation of datasets. In particular, we select the MBPP$_{train}$ because of its high average semantic correlation score with method-level source code. For this task, we employ GPT-3.5-turbo to correct the semantic correlations between intent-code pairs in the dataset. Our prompt to the GPT was straightforward as follows: 'Please correct the semantic correlations between the following pair <NL intent> ... <Code> ...'. Upon obtaining the corrected pairs, we achieved an improved correlation score of 86.23%, surpassing the original 83.94% checking with our fine-tuned DeepSeek-Coder-6.7B-Instruct model (Section 3.6.2). Subsequently, we re-fine-tune the original models to compare their performances.

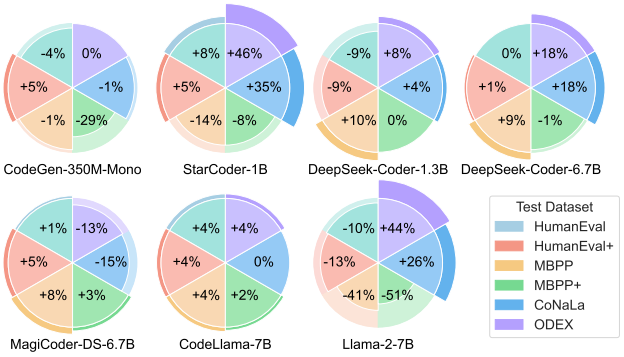

Figure 4: RQ2 — Effectiveness of the semantic correlation correction. `+nn%` and `-nn%` represent the performance increment and decrement, respectively, after correcting semantic correlation. The number details are provided as a table in our Online Appendix [1].

Results and Analyses: Correcting semantic correlations generally amplifies the fine-tuning performance, notwithstanding occasional declines as shown in Figure 4. We apply the Wilcoxon signed-rank test [18]

to the paired test samples (before/after semantic correlation correction). The p-value of the statistical test is 0.129, and thus we cannot reject the null hypothesis. We detail the results in the rest of this section.

On the Functionality-focused. Notably, the impact on LLMs tailored for code, such as DeepSeek-Coder-6.7B, MagiCoder-DS-6.7B, and CodeLlama-7B, predominantly yields positive outcomes on functionality-focused benchmarks. This suggests a trend where larger model sizes correlate with more consistent improvements. The HumanEval+$_{test}$ notably experiences the most frequent enhancements. Specifically, with the exception of DeepSeek-Coder-1.3B and Llama-2-7B, all models demonstrate improvements ranging from 4.00% to 5.00%. Similarly, the MBPP$_{test}$ experiences enhancements with all the LLMs tailored for code as well as DeepSeek-Coder-1.3B. Except for MagiCoder-DS-6.7B and CodeLlama-7B, the correlation correction negatively impacts the MBPP+$_{test}$. In particular, CodeGen-350M-Mono faces a significant decline of -29%. The foundation model, Llama-2-7B, consistently exhibits negative impacts with the correction on functionality-focused benchmarks.

The results suggest that the impact of semantic correlation correction varies across different benchmarks and model configurations. While larger model sizes tend to benefit from the correction, as evidenced by the consistent improvements observed in functionality-focused benchmarks such as HumanEval+$_{test}$, smaller models like DeepSeek-Coder-1.3B and Llama-2-7B appear less resilient to its effects. Moreover, the significant performance decline of CodeGen-350M-Mono on the MBPP$_{test}$ highlights the importance of considering the specific characteristics of each model when applying correlation correction techniques. Further investigation is warranted to elucidate the underlying factors contributing to these disparities and to inform more nuanced strategies for optimizing model performance through correlation correction.

On the Alignment-focused. It is interesting to observe such notable fluctuations in the performances of various models under semantic correlation correction. For instance, StarCoder-1B demonstrates significant improvements of 35.00% and 46.00% on the CoNaLa$_{test}$ and ODEX$_{test}$, respectively. Conversely, MagiCoder-DS-6.7B, another LLM for code, experiences a remarkable decline of -15.00% and -13.00% on the same benchmarks. Notably, the foundation model, Llama-2-7B, showcases exceptional positive experiences with performance improvements of 26.00% and 44.00%. These observations underscore the nuanced effects of correlation correction and suggest that certain models may respond more positively or negatively to the intervention depending on their specific characteristics and training data. Further analysis is warranted to discern the underlying factors driving such diverse outcomes and to inform tailored strategies for optimizing model performance.

To further interpret, from the dataset-model interaction view, models with strong contextual understanding, such as MagiCoder-DS-6.7B and DeepSeek-Coder-6.7B, showed substantial improvements with enhanced semantic correlations. For example, improving the alignment between intents and code in the MBPP dataset led to better performance on functionality-focused benchmarks like HumanEval. This suggests that models capable of leveraging context can better utilize improved semantic correlations to generate accurate code. From the task specificity view, functionality-focused benchmarks showed greater benefits from semantic corrections than alignment-focused ones. This is because functionality-focused tasks directly depend on the accuracy of the code generated from the given intents. In contrast, alignment-focused tasks may not require as precise a match between intent and code, explaining the lesser impact observed.

> **Answer to RQ2**: The impact of semantic correlation correction on the model performance varies for each dataset. It can enhance the model performance for *Functionality-focused* benchmarks, particularly for larger models. However, smaller models may not benefit as much, even decline. In *Alignment-focused* benchmarks, model performance varies under the correction, with models showing significant improvements while others decline.

4.3 RQ3: Data Design Alignment

Setup: Our preliminary results shown in Section 3.6.3 motivate this research question and we hypothesize that design choices have an impact on the performance of code generation models through fine-tuning. Particularly, we craft a dataset design based on our preliminary investigations and literature review as follows. First, the findings in Section 3.6.3 suggest that datasets organized at the **method-level**, such as MBPP, hold promise for enhancing language models' comprehension of context, consequently leading to improved code generation performance overall (See Section 3.) Additionally, prior studies [7, 36] have highlighted the benefits of method-level granularity in accurately capturing the intent and functionality of code. Notably, the widely recognized CodeBERT model [21], a benchmark for code-related tasks, has been trained using method-level data. Second, previous work [85] has underscored the significance of **method signatures**, suggesting that models may struggle to generate new methods from scratch without this crucial information [56]. Finally, several studies [30, 46, 84] emphasize the importance of **annotations** in streamlining the training process, as they provide explicit semantic links between code and natural language, thereby enhancing learning efficiency.

To test our hypothesis, we specifically apply our design to a dataset, CodeAlpaca$_{train}$, creating CodeAlpaca$_{designed}$, due to its suboptimal performance in code generation when used to fine-tune our target models. One-line-based datasets, such as CoNaLa$_{train}$ and ODEX$_{train}$, present challenges in conversion due to their inherent limitations on contextual information within both the intent and code segments. To expedite the process and ensure the creation of accurate datasets, we utilize GPT-4 to convert the 2.19K data points. This conversion process involves prompting the model with specific instructions, such as *'Here is a specific format of a data point: <NL intent> ... <Code> ... Please convert the following data point to match this design.'* It is worth noting that we have the flexibility to perform manual conversions, either through manual labor or any LLM-assisted approaches. Curated examples can be found in [1].

Results and Analyses: Improving dataset design choices has a significant impact on the model performance. We compare the model performance before, i.e., CodeAlpaca$_{train}$, and after applying new design choices, i.e., CodeAlpaca$_{designed}$ to each dataset. The results are represented in Figure 5. We also apply the Wilcoxon signed-rank test [18] to the paired test samples (before/after semantic correlation correction). The p-value of the statistical test is 0.000061; thus, we can reject the null hypothesis.

On the Functionality-focused. The first four charts in Figure 5 illustrate a consistent trend across all our target benchmarks: Each model with the re-designed CodeAlpaca$_{train}$ consistently outperforms those fine-tuned with the original dataset. On average, the normal-sized LMs exhibit superior performance improvements, with the exception of Llama-2-7B. Specifically, across all functionality-focused benchmarks, we observed average improvement rates of 63.13%, 51.78%, 35.22%, 14.21%, 32.93%, 35.56%, and 271.05% for CodeGen-350M-Mono, StarCoder-1B, DeepSeek-Coder-1.3B, DeepSeek-Coder-6.7B, MagiCoder-DS-6.7B, CodeLlama-7B, and Llama-2-7B, respectively. Notably, performance enhancement rates vary significantly, ranging from 7.70% (observed in the case of DeepSeek-Coder-6.7B on the HumanEval$_{test}$) to 422.39% (observed in the case of Llama-2-7B on the MBPP+$_{test}$).

Intriguingly, the observation that MagiCoder-DS-6.7B exhibits more pronounced and positive impacts on design alignment compared to DeepSeek-Coder-6.7B, despite their similar original performances, suggests that certain models may be more sensitive to dataset modifications. This finding underscores the importance of not only evaluating model performance but also understanding the underlying mechanisms driving performance improvements. Overall, these results provide valuable insights into the impact of dataset design on model performance in code generation tasks and highlight avenues for further research and optimization.

On the Alignment-focused. Similar to the functionality-focused benchmark results, the performance is consistently better than those with the original dataset. Specifically, across two Alignment-focused benchmarks, we obtained average enhancement rates of 21.60%, 108.95%, 44.71%, 37.41%, 19.38%, 93.03%, and 179.88% for CodeGen-350M-Mono, StarCoder-1B, DeepSeek-Coder-1.3B, DeepSeek-Coder-6.7B, MagiCoder-DS-6.7B, CodeLlama-7B, and Llama-2-7B, respectively.

Unlike the phenomenon observed in Functionality-focused benchmarks, where normal-sized LMs showed better improvement rates, we did not observe a similar trend in Alignment-focused benchmarks. This suggests that the impact of dataset design on model performance may vary depending on the nature of the benchmark and the specific characteristics of the tasks involved. Furthermore, the contrasting results between MagiCoder-DS-6.7B and DeepSeek-Coder-6.7B in Alignment-focused versus Functionality-focused benchmarks highlight the complexity of model-dataset interactions. Specifically, while MagiCoder-DS-6.7B exhibited more pronounced impacts on design alignment in the Functionality-focused benchmarks, DeepSeek-Coder-6.7B showed a better improvement rate for Alignment-focused benchmarks.

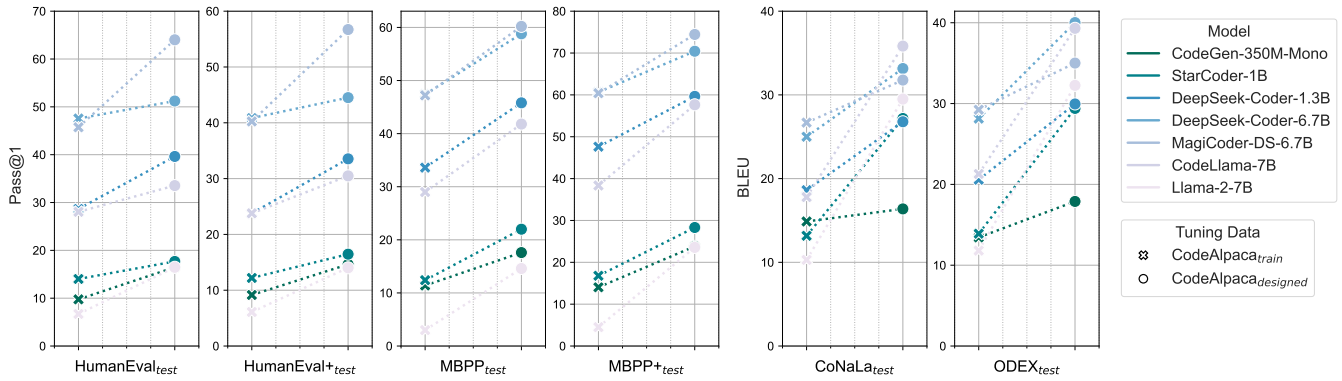

Figure 5: RQ3 - Effectiveness of design alignment. The Y-axis shows the values of the metrics (Pass@1 for the left four plots and BLEU for the right two plots). The X-axis of each subplot represents the datasets before (left-hand side: CodeAlpaca$_{train}$) and after (right-hand side: CodeAlpaca$_{designed}$) applying design alignment. Both datasets are applied to each target model to fine-tune it. Detailed values are available in [1].

Overall, these results highlight the effectiveness of our curated dataset design in improving model performance across a range of source code generation tasks. The consistently superior performance of models fine-tuned with the curated dataset suggests that specific features and characteristics within the dataset are conducive to better model training and performance. Furthermore, the varying improvement rates across different models and benchmarks underscore the importance of considering model-specific characteristics and task complexities when evaluating dataset effectiveness.

Answer to RQ3: Updating dataset design significantly enhances model performance. Notably, while normal-sized LMs demonstrated better improvement rates in *Functionality-focused* benchmarks, no such trend was observed in *Alignment-focused* benchmarks. These findings highlight the critical role of dataset design in enhancing model performance for code generation.

5 DISCUSSION

Performance dropping dataset. While fine-tuning is widely used to enhance performance in many language model tasks, it can also lead to performance declines in certain cases. Surprisingly, Table 1 reveals that fine-tuning with original datasets resulted in average increase/decrease rates of -13.37%, -15.69%, -24.59%, 49.96%, 11.17%, and 13.32% for HumanEval, HumanEval+, MBPP, MBPP+, CoNaLa, and ODEX benchmarks, respectively. The increase rates observed for each benchmark were 23.83% (combining StarCoder-1B with ODEX$_{train}$), 21.79% (combining DeepSeek-Coder-1.3B with DS-1000$_{train}$), 8.7% (combining CodeGen-350M-Mono with MBPP$_{train}$), 194.71% (combining Llama-2-7B with MBPP$_{train}$), and 200.05% and 268.36% for the last two benchmarks when combining CodeGen-350M-Mono with ODEX$_{train}$. However, the decrease rates were observed with a specific model, Llama-2-7B, when fine-tuned with the CoNaLa$_{train}$ for all the Functionality-focused benchmarks and the CodeAlpaca$_{train}$ for all the Alignment-focused benchmarks. The significant fluctuations in performance underscore the importance of the dataset quality and this observation highlights the critical need for rigorous dataset curation to ensure optimal model performance in code generation tasks. Further research into dataset refinement strategies is essential to address this limitation and unlock the full definition of the *high-quality* dataset for CodeLLMs.

Comparing against the Zero-shot. We intentionally excluded comparisons against zero-shot model performances, except for preliminary analysis, as our focus is on evaluating dataset quality and identifying potential enhancement techniques. While these comparisons are somewhat tangential to our primary aim, they may offer insights into techniques that are superior to the original dataset quality as well. In the case of our dataset design curation, most of the combinations show superior performances, when additionally considering zero-shot performances. For Functionality-focused benchmarks, the average increase rates against Zero-shot were 18.93%, 18.94%, 1.89%, and 97.51% for HumanEval, HumanEval+, MBPP, and MBPP+ while the results show a slight decrease on average with Alignment-focused benchmarks (i.e., -4.27% and -3.90% on both CoNaLa and ODEX datasets). This further emphasizes the impact of dataset design choices. By carefully selecting and refining datasets based on their characteristics and suitability for the intended application, researchers and practitioners can optimize model training and improve real-world performance outcomes.

Generalization of the Study. While we acknowledge the importance of generalizability in studies, our research specifically aimed to investigate the impact of fine-tuning dataset quality on the performance of CodeLLMs for natural language text-to-code generation. This focused approach allowed us to conduct a deep and thorough analysis of this particular application. We believe that our insights into fine-tuning dataset quality, and we also believe that our insights can easily extend to other CodeLLMs because other tasks are also trained by the same method, fine-tuning the pre-trained models. Furthermore, our current study focused on Python due to its widespread use in both software and machine learning communities, and the availability of rich and diverse datasets. Difficulties that should be considered for other programming languages like Java and Go would be as follows: 1) collecting vast amounts of training datasets, 2) we would also need to establish new benchmarks and evaluation metrics tailored to these languages, and 3) the process of fine-tuning is computationally intensive and it requires careful evaluation to ensure performance parity across languages.

6 THREATS TO VALIDITY

External validity. There may be generalizability issues and selection bias regarding the applicability of the training datasets used in the study, as there are more datasets used for code generation tasks in model training. However, we mitigated such issues by carefully selecting a diverse range of datasets that represent different code generation scenarios and domains as described in Section 3.1. The controlled experimental environment may lack the complexity and diversity of real-world tasks, posing a threat to ecological validity. This threat has been automatically mitigated as the selected datasets and benchmarks are collected from real-world open-source projects and communities.

Internal validity. We acknowledge that there may be potential errors in assessing dataset features and model performance metrics, so we relied on standardized procedures and official sources from reputable AI communities like HuggingFace[12]. This ensured consistency and reliability in our experimental process. We also controlled for environmental factors such as GPU clusters and hyperparameters to maintain consistency across experiments. While there's a possibility of overlooking certain features that impact dataset quality, we mitigated this by leveraging well-known detection tools and refactoring libraries tailored for dataset assessment. These tools enabled systematic evaluation of dataset quality aspects, enhancing the reliability of our findings and the internal validity of the study.

7 RELATED WORK

Data Characteristic Analysis. Researchers have conducted numerous studies analyzing data characteristics for software-related tasks. Frantzeskou et al. [22, 23] specifically focused on software developer copyrights. They categorized key features as variable, method, class, package naming, layout, and comments to identify authorship, aiming to aid in triage processes or protect intellectual property rights. Calleja et al. [12] compiled a dataset of malware and conducted an investigation to reveal correlations between code reuse and malware creation. Their analysis focused on the number of duplicates, length, and complexity of source code, discovering a strong correlation, with both the length and complexity of the code increasing linearly over time. Zhang et al. [90] introduced a dataset analysis framework for bug fixing, the framework incorporates error-related features such as error type, error line length, average line depth, and more. Subsequently, Pujar et al. [59] developed a bug detection model using this framework. Their findings suggested that characteristic analysis facilitated the model in detecting a higher number of bugs. Recently, several researchers [38, 72, 73] studied the characteristic analysis of the generated source code by the language models. The results indicated that characteristic analysis is mandatory for comprehensively understanding the quality and potential of the datasets for software-related tasks as well as training language models. In contrast, our study focuses specifically on dataset characteristics for training language models. We aim to uncover *high-quality* datasets suitable for training language models and to explore potential approaches to enhance code generation performance.

[12]https://huggingface.co/

Language Model Fine-tuning. In addition to examining data characteristics, researchers also focus on fine-tuning language models, a process that entails additional training of pre-trained models with smaller, task-specific datasets to enhance their performance on particular tasks [17, 19]. Fine-tuning techniques have been used for many tasks in the field of Software Engineering such as code generation [67, 80], code search [64, 77], and code review [40, 45]. Regarding the dataset quality, it may be crucial for the successful fine-tuning of language models. Ahmed et al. [5] highlight the importance of dataset quality, particularly the diversity and representativeness of different programming languages and coding styles. Additionally, Martin et al [48] stress the significance of using high-quality datasets for fine-tuning models, as cleaner data leads to better model performance and more accurate API sequence predictions. Berabi et al. [10] proposed TFix, which leverages a high-quality dataset for fine-tuning, encompassing a broad range of coding errors extracted from GitHub commits. These indicate the fine-tuning performance depends on what the language model learns such as diverse patterns and solutions. Furthermore, dataset quality impacts the generalization ability of the models. We delve deeper into the mechanisms by which features affect the dataset quality for fine-tuning CodeLLMs and explore the potential approaches to enhance the model performance.

8 CONCLUSION AND FUTURE WORK

The ascent of CodeLLMs is reshaping the landscape of software development, yet concerns linger regarding the handling of data, chiefly driven by apprehensions about data quality. We emphasize the paramount importance of *high-quality* datasets in training LLMs for specific tasks, exploring various features such as code-related issues, pair-wise semantic correlation, and design nuances. Our deep dive into dataset characteristics sheds light on factors critical to model performance, particularly in the realm of source code generation. Furthermore, we probe the effectiveness of potential solutions to enhance code generation performance. Our findings not only provide a blueprint for augmenting LLM efficacy through enhanced dataset quality but also furnish researchers and practitioners with invaluable insights for harnessing the full potential of LLMs in real-world software engineering applications. Given the potential opportunities for further enhancing model performance through sequence training and the combination of processing methods, we actively plan to conduct additional studies in this area. Our future work also includes the quality of pre-training datasets for CodeLLMs so that we can thoroughly cover pre-training as well as fine-tuning.

ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2021R1I1A3048013) and by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 949014). Additionally, this research was also funded in whole, or in part, by the Luxembourg National Research Fund (FNR), grant reference NCER22/IS/16570468/NCER-FT.

REFERENCES

[1] 2024. https://figshare.com/s/4c79642b98adc74f3234.

[2] 2024. https://www.sonarsource.com/.

[3] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[4] Areeg Ahmed, Shahira Azab, and Yasser Abdelhamid. 2023. Source-Code Generation Using Deep Learning: A Survey. In *Progress in Artificial Intelligence*, Nuno Moniz, Zita Vale, José Cascalho, Catarina Silva, and Raquel Sebastião (Eds.). Springer Nature Switzerland, Cham, 467–482.

[5] Toufique Ahmed and Premkumar Devanbu. 2022. Multilingual training for software engineering. In *Proceedings of the 44th International Conference on Software Engineering*. 1443–1455.

[6] Alon Albalak, Yanai Elazar, Sang Michael Xie, Shayne Longpre, Nathan Lambert, Xinyi Wang, Niklas Muennighoff, Bairu Hou, Liangming Pan, Haewon Jeong, et al. 2024. A Survey on Data Selection for Language Models. *arXiv preprint arXiv:2402.16827* (2024).

[7] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.

[8] AI Anthropic. 2024. The claude 3 model family: Opus, sonnet, haiku. *Claude-3 Model Card* (2024).

[9] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[10] Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. 2021. Tfix: Learning to fix coding errors with a text-to-text transformer. In *International Conference on Machine Learning*. PMLR, 780–791.

[11] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[12] Alejandro Calleja, Juan Tapiador, and Juan Caballero. 2018. The malsource dataset: Quantifying complexity and code reuse in malware development. *IEEE Transactions on Information Forensics and Security* 14, 12 (2018), 3175–3190.

[13] Yihan Cao, Yanbin Kang, Chi Wang, and Lichao Sun. 2023. Instruction Mining: When Data Mining Meets Large Language Model Finetuning. arXiv:2307.06290 [cs.CL]

[14] Sahil Chaudhary. 2023. Code Alpaca: An Instruction-following LLaMA model for code generation. https://github.com/sahil280114/codealpaca.

[15] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[16] YunSeok Choi and Jee-Hyong Lee. 2023. CodePrompt: Task-Agnostic Prefix Tuning for Program and Language Generation. In *Findings of the Association for Computational Linguistics: ACL 2023*. 5282–5297.

[17] Kenneth Ward Church, Zeyu Chen, and Yanjun Ma. 2021. Emerging trends: A gentle introduction to fine-tuning. *Natural Language Engineering* 27, 6 (2021), 763–778. https://doi.org/10.1017/S1351324921000322

[18] W. J. Conover. 1999. *Practical Nonparametric Statistics, 3rd* (3rd edition ed.). Wiley, New York.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[20] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. *arXiv preprint arXiv:2310.03533* (2023).

[21] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[22] Georgia Frantzeskou, Stephen MacDonell, Efstathios Stamatatos, and Stefanos Gritzalis. 2008. Examining the significance of high-level programming features in source code author classification. *Journal of Systems and Software* 81, 3 (2008), 447–460.

[23] Georgia Frantzeskou, Efstathios Stamatatos, Stefanos Gritzalis, and Sokratis Katsikas. 2006. Effective identification of source code authors using byte-level information. In *Proceedings of the 28th international conference on Software engineering*. 893–896.

[24] Hadi Ghanbari, Tero Vartiainen, and Mikko Siponen. 2018. Omission of quality software development practices: A systematic literature review. *ACM Computing Surveys (CSUR)* 51, 2 (2018), 1–27.

[25] Lucas Gren and Vard Antinyan. 2017. On the Relation Between Unit Testing and Code Quality. In *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 52–56. https://doi.org/10.1109/SEAA.2017.36

[26] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. 2023. Textbooks are all you need. *arXiv preprint arXiv:2306.11644* (2023).

[27] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).

[28] Md Shariful Haque, Jeff Carver, and Travis Atkison. 2018. Causes, impacts, and detection approaches of code smell: a survey. In *Proceedings of the ACMSE 2018 Conference*. 1–8.

[29] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. In *International conference on machine learning*. PMLR, 2790–2799.

[30] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[31] Joseph Marvin Imperial and Harish Tayyar Madabushi. 2023. Flesch or Fumble? Evaluating Readability Standard Alignment of Instruction-Tuned Language Models. *arXiv preprint arXiv:2309.05454* (2023).

[32] Barbara Kitchenham. 2004. Procedures for Performing Systematic Reviews. *Keele, UK, Keele Univ.* 33 (08 2004).

[33] Gregory Koch, Richard Zemel, Ruslan Salakhutdinov, et al. 2015. Siamese neural networks for one-shot image recognition. In *ICML deep learning workshop*, Vol. 2. Lille.

[34] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems* 32 (2019).

[35] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2022. DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation. *ArXiv* abs/2211.11501 (2022).

[36] Triet HM Le, Hao Chen, and Muhammad Ali Babar. 2020. Deep learning for source code modeling and generation: Models, applications, and challenges. *ACM Computing Surveys (CSUR)* 53, 3 (2020), 1–38.

[37] Mosh Levy, Alon Jacoby, and Yoav Goldberg. 2024. Same Task, More Tokens: the Impact of Input Length on the Reasoning Performance of Large Language Models. *arXiv preprint arXiv:2402.14848* (2024).

[38] Ke Li, Sheng Hong, Cai Fu, Yunhe Zhang, and Ming Liu. 2023. Discriminating Human-authored from ChatGPT-Generated Code Via Discernable Feature Analysis. In *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 120–127.

[39] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[40] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. 2022. Automating code review activities by large-scale pre-training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1035–1047.

[41] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. 2022. Codereviewer: Pre-training for automating code review activities. *arXiv preprint arXiv:2203.09095* (2022).

[42] Bin Lin, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2019. On the impact of refactoring operations on code naturalness. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 594–598.

[43] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2024).

[44] Yixin Liu, Avi Singh, C. Daniel Freeman, John D. Co-Reyes, and Peter J. Liu. 2023. Improving Large Language Model Fine-tuning for Solving Math Problems. arXiv:2310.10047 [cs.CL]

[45] Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. 2023. LLaMA-Reviewer: Advancing Code Review Automation with Large Language Models through Parameter-Efficient Fine-Tuning. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 647–658.

[46] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[47] Md Abdullah Al Mamun, Christian Berger, and Jörgen Hansson. 2017. Correlations of software code metrics: an empirical study. In *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement* (Gothenburg, Sweden)

(IWSM Mensura '17). Association for Computing Machinery, New York, NY, USA, 255–266. https://doi.org/10.1145/3143434.3143445

[48] James Martin and Jin LC Guo. 2022. Deep api learning revisited. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. 321–330.

[49] Tom Mens and Tom Tourwé. 2004. A survey of software refactoring. *IEEE Transactions on software engineering* 30, 2 (2004), 126–139.

[50] Ben Naismith, Phoebe Mulcaire, and Jill Burstein. 2023. Automated evaluation of written discourse coherence using GPT-4. In *Proceedings of the 18th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2023)*. 394–403.

[51] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[52] R OpenAI. 2023. GPT-4 technical report. *arXiv* (2023), 2303–08774.

[53] Fabio Palomba, Marco Zanoni, Francesca Arcelli Fontana, Andrea De Lucia, and Rocco Oliveto. 2017. Toward a smell-aware bug prediction model. *IEEE Transactions on Software Engineering* 45, 2 (2017), 194–218.

[54] Michail Papamichail, Themistoklis Diamantopoulos, and Andreas Symeonidis. 2016. User-Perceived Source Code Quality Estimation Based on Static Analysis Metrics. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 100–107. https://doi.org/10.1109/QRS.2016.22

[55] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[56] Md Rizwan Parvez. 2022. *Learning through Auxiliary Supervision for Multi-modal Low-resource Natural Language Processing*. University of California, Los Angeles.

[57] Anthony Peruma, Steven Simmons, Eman Abdullah AlOmar, Christian D Newman, Mohamed Wiem Mkaouer, and Ali Ouni. 2022. How do i refactor this? An empirical study on refactoring trends and topics in Stack Overflow. *Empirical Software Engineering* 27, 1 (2022), 11.

[58] Dorin Pomian, Abhiram Bellur, Malinda Dilhara, Zarina Kurbatova, Egor Bogomolov, Timofey Bryksin, and Danny Dig. 2024. Together We Go Further: LLMs and IDE Static Analysis for Extract Method Refactoring. *arXiv preprint arXiv:2401.15298* (2024).

[59] Saurabh Pujar, Yunhui Zheng, Luca Buratti, Burn Lewis, Yunchung Chen, Jim Laredo, Alessandro Morari, Edward Epstein, Tsungnan Lin, Bo Yang, et al. 2024. Analyzing source code vulnerabilities in the D2A dataset with ML ensembles and C-BERT. *Empirical Software Engineering* 29, 2 (2024), 48.

[60] Crystal Qian, Emily Reif, and Minsuk Kahng. 2024. Understanding the Dataset Practitioners Behind Large Language Model Development. *arXiv preprint arXiv:2402.16611* (2024).

[61] Scott Reed, Zeynep Akata, Honglak Lee, and Bernt Schiele. 2016. Learning deep representations of fine-grained visual descriptions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 49–58.

[62] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).

[63] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[64] Iman Saberi, Fatemeh Fard, and Fuxiang Chen. 2024. Utilization of Pre-trained Language Model for Adapter-based Knowledge Transfer in Software Engineering. arXiv:2307.08540 [cs.SE]

[65] Tushar Sharma, Maria Kechagia, Stefanos Georgiou, Rohit Tiwari, Indira Vats, Hadi Moazen, and Federica Sarro. 2021. A survey on machine learning techniques for source code analysis. *arXiv preprint arXiv:2110.09610* (2021).

[66] Noam Shazeer and Mitchell Stern. 2018. Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning*. PMLR, 4596–4604.

[67] Ensheng Shi, Yanlin Wang, Hongyu Zhang, Lun Du, Shi Han, Dongmei Zhang, and Hongbin Sun. 2023. Towards Efficient Fine-Tuning of Pre-trained Code Models: An Experimental Study and Beyond. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (<conf-loc>, <city>Seattle</city>, <state>WA</state>, <country>USA</country>, </conf-loc>) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 39–51. https://doi.org/10.1145/3597926.3598036

[68] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-level prompt generation for large language models of code. In *International Conference on Machine Learning*. PMLR, 31693–31715.

[69] Irene Solaiman and Christy Dennison. 2021. Process for adapting language models to society (palms) with values-targeted datasets. *Advances in Neural Information Processing Systems* 34 (2021), 5861–5873.

[70] Ioannis Stamelos, Lefteris Angelis, Apostolos Oikonomou, and Georgios L Bleris. 2002. Code quality analysis in open source software development. *Information systems journal* 12, 1 (2002), 43–60.

[71] Zhensu Sun, Li Li, Yan Liu, Xiaoning Du, and Li Li. 2022. On the importance of building high-quality training datasets for neural code search. In *Proceedings of the 44th International Conference on Software Engineering*. 1609–1620.

[72] Florian Tambon, Arghavan Moradi Dakhel, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Giuliano Antoniol. 2024. Bugs in large language models generated code. *arXiv preprint arXiv:2403.08937* (2024).

[73] Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. 2024. Chatgpt vs sbst: A comparative assessment of unit test suite generation. *IEEE Transactions on Software Engineering* (2024).

[74] Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Zhiyuan Liu, and Maosong Sun. 2024. Debugbench: Evaluating debugging capability of large language models. *arXiv preprint arXiv:2401.04621* (2024).

[75] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[76] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. 2022. Using pre-trained models to boost code review automation. In *Proceedings of the 44th international conference on software engineering*. 2291–2302.

[77] Deze Wang, Boxing Chen, Shanshan Li, Wei Luo, Shaoliang Peng, Wei Dong, and Xiangke Liao. 2023. One adapter for all programming languages? adapter tuning for code search and summarization. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 5–16.

[78] Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2022. Execution-Based Evaluation for Open-Domain Code Generation. *arXiv preprint arXiv:2212.10481* (2022).

[79] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120* (2023).

[80] Martin Weyssow, Xin Zhou, Kisub Kim, David Lo, and Houari Sahraoui. 2024. Exploring Parameter-Efficient Fine-Tuning Techniques for Code Generation with Large Language Models. arXiv:2308.10462 [cs.SE]

[81] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 1–10.

[82] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. 2015. Show, attend and tell: Neural image caption generation with visual attention. In *International conference on machine learning*. PMLR, 2048–2057.

[83] Ran Xu, Caiming Xiong, Wei Chen, and Jason Corso. 2015. Jointly modeling deep video and compositional text to bridge vision and language in a unified framework. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 29.

[84] Qiaomu Xue. 2023. Automating Code Generation for MDE using Machine Learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 221–223.

[85] Guang Yang, Yu Zhou, Wenhua Yang, Tao Yue, Xiang Chen, and Taolue Chen. 2024. How important are good method names in neural code generation? a model robustness perspective. *ACM Transactions on Software Engineering and Methodology* 33, 3 (2024), 1–35.

[86] Ke Yang, Jiateng Liu, John Wu, Chaoqi Yang, Yi R Fung, Sha Li, Zixuan Huang, Xu Cao, Xingyao Wang, Yiquan Wang, et al. 2024. If llm is the wizard, then code is the wand: A survey on how code empowers large language models to serve as intelligent agents. *arXiv preprint arXiv:2401.00812* (2024).

[87] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *2018 IEEE/ACM 15th international conference on mining software repositories (MSR)*. IEEE, 476–486.

[88] Litian Zhang, Xiaoming Zhang, and Junshu Pan. 2022. Hierarchical cross-modality semantic correlation learning model for multimodal summarization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 11676–11684.

[89] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. 2019. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675* (2019).

[90] Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. 2021. D2a: A dataset built for ai-based vulnerability detection methods using differential analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 111–120.