COMPCODEVET: A COMPILER-GUIDED VALIDATION AND ENHANCEMENT APPROACH FOR CODE DATASET

Le Chen ¹ Arijit Bhattacharjee ¹ Nesreen K. Ahmed ² Niranjan Hasabnis ² Gal Oren ³ Bin Lei ⁴ Ali Jannesari ¹

ABSTRACT

Large language models (LLMs) have become increasingly prominent in academia and industry due to their remarkable performance in diverse applications. As these models evolve with increasing parameters, they excel in tasks like sentiment analysis and machine translation. However, even models with billions of parameters face challenges in tasks demanding multi-step reasoning. Code generation and comprehension, especially in C and C++, emerge as significant challenges. While LLMs trained on code datasets demonstrate competence in many tasks, they struggle with rectifying non-compilable C and C++ code. Our investigation attributes this subpar performance to two primary factors: the quality of the training dataset and the inherent complexity of the problem, which demands intricate reasoning. Existing "Chain of Thought" (CoT) prompting techniques aim to enhance multi-step reasoning. This approach, however, retains the limitations associated with the latent drawbacks of LLMs. In this work, we propose CompCodeVet, a compiler-guided CoT approach to produce compilable code from non-compilable ones. Diverging from the conventional approach of utilizing larger LLMs, we employ compilers as a teacher to establish a more robust zero-shot thought process. The evaluation of CompCodeVet on two open-source code datasets shows that CompCodeVet can improve the training dataset quality for LLMs.

1 Introduction

In recent years, the field of artificial intelligence has witnessed the meteoric rise of large language models (LLMs)(Brown et al., 2020)(Chen et al., 2021a)(Chowdhery et al., 2022)(OpenAI, 2023). These models, built on colossal amounts of data and high computational capacities, have shown impressive results in a multitude of natural language processing tasks. Their capacity to understand, generate, and even exhibit creativity in language processing tasks has made them a focal point in machine learning research and applications.

The journey of machine learning in the realm of code analysis is not nascent. Researchers and developers have long worked on automating tasks such as code analysis, bug fixing, and code optimization. With the emergence and subsequent dominance of LLMs in the AI landscape, there has been a renewed and increasing interest in harnessing their power specifically for code-based tasks. This includes code analysis, code completion, and even automated bug detection.

For effective training, LLMs demand voluminous amounts of data. While there exists a plethora of resources for natural language texts, the same cannot be said for code data. Presently, data for training LLMs on code-based tasks is

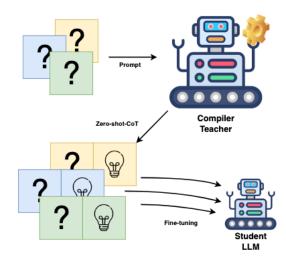


Figure 1. CompCodeVet: Diverging from the conventional approach of utilizing larger LLMs, we employ compilers (represented by the gear in the robot's hand) as teachers to establish a more robust zero-shot thought process. A smaller LLM is fine-tuned for compilable code generation.

primarily gathered from two major sources. One being, open-source repositories such as GitHub which offer a treasure trove of code from diverse projects and languages. The

other are code snippets embedded within website content, often seen in platforms like StackOverflow where users share and discuss code snippets for problem-solving. However, relying on these sources presents the followin challenges:

- Incomplete code data: A significant portion of the code from these sources, particularly from online discussions, might be fragments rather than complete code. This incomplete nature can hinder the efficacy of LLMs when tasked with code auto-completions.
- Inaccurate file extensions and mislabeling: Code snippets extracted from inline website content frequently lack associated file extensions. This, combined with occasional mislabeling of the programming language used, can impede tasks like code classification.
- 3. Low compilability: A non-trivial amount of code from these resources may not be directly compilable or might have inherent errors. This could compromise the quality of code generated by LLMs, as they might learn from flawed or erroneous code patterns.

Recognizing these challenges, this paper sets forth a robust framework called CodeCompVet. The primary aim is to meticulously examine the current code datasets employed for training LLMs and subsequently enhance their quality and completeness. By addressing the inherent flaws and improving the quality of the training datasets, we aspire to push the boundaries of what LLMs can achieve in the domain of code analysis and generation while using a smaller number of parameters which in turn reduce GPU compute times and curb the environmental costs of training LLMs (Bender et al., 2021).

2 BACKGROUND

Code compilation plays a pivotal role in code analysis, particularly in the domain of high-performance computing (HPC). It stands to reason that the proportion of compilable code within a training dataset would influence the ability of Large Language Models to generate valid, compilable code. In this section, we explore the significance of compilable code and provide an overview of the existing code datasets utilized for Large Language Model training.

2.1 Source Code Analysis

Source code analysis, especially static code analysis, has a rich history that traces its roots back to the early days of programming. Its primary objective has always been to analyze the source code of a program without executing it, enabling developers to detect issues, vulnerabilities, or even adherence to coding standards. Historically, manual code reviews served as the primary means of analyzing code. However, as software projects grew in complexity and size, manual reviews became increasingly challenging. This precipitated the need for automated code analysis tools.

Over the years, various tools have been developed to cater to different programming languages and purposes. For instance, tools like *Lint* for C, *SonarQube* for multiple languages, and *FindBugs* for Java have become mainstays in the developer's toolbox. These tools automatically inspect the codebase for potential issues, ranging from style violations and potential bugs to more severe security vulnerabilities

Compilable code plays a crucial role in this process. Only when the code can be successfully compiled can certain types of analyses, especially those that rely on understanding the interplay between different parts of a program, be conducted effectively. Compilable code, essentially, serves as a foundational requirement for many code analysis tools to function optimally. It also enables possibilities like dynamic analysis, where the program's behavior is observed during its execution, and facilitates a more comprehensive understanding of the software's potential behavior in realworld scenarios. For example, Intermediate Representations (IR) of code can only be generated from compilable code by LLVM (Lattner & Adve, 2004).

In the context of Large Language Models and their training on code datasets, the presence of compilable code is indispensable. It ensures that the generated code snippets are not just syntactically correct but also semantically meaningful, allowing for a richer and more accurate analysis.

2.2 Large Language Models and Code Generation

Large Language Models (LLMs) have emerged as a groundbreaking innovation in the realm of natural language processing. Rooted in deep learning architectures, especially transformer-based (Vaswani et al., 2017) models like BERT (Devlin et al., 2019), GPT (OpenAI, 2023), PaLM (Chowdhery et al., 2022), and their successors, LLMs have the capability to comprehend and generate human-like text across a myriad of tasks. They have performed exceptionally well on code completion benchmarks like HumanEval (Chen et al., 2021a) which look at the functional correctness of the code and MBPP (Austin et al., 2021).

The sheer size and scale of LLMs, often trained on vast amounts of diverse textual data, enable them to encapsulate nuanced patterns and intricacies of language. While they were initially oriented towards natural language tasks, their prowess was soon recognized in adjacent domains, most notably in code generation and comprehension.

2.2.1 LLMs for Code Generation

The idea of automating code writing isn't novel. However, the introduction of LLMs into this space has revolutionized what is possible. By training LLMs on vast code repositories, these models have demonstrated an ability to understand programming constructs, idioms, and even best practices across multiple languages. For instance, GitHub CoPilot¹, an AI based pair programming system can generate code directly from natural language problem description. CoPilot is powered by Codex (Chen et al., 2021b), which is developed by OpenAI and obtained by finetuning 12B parameter GPT models billions of lines of source code.

Platforms like GitHub, StackOverflow, and various code documentation sites have been invaluable resources for training data. When presented with a coding problem or a partial code snippet, LLMs can suggest completions, fix bugs, translate languages (Roziere et al., 2020), or even generate entire routines (Chen et al., 2021b). Such capabilities have not only expedited the coding process but have also served as educational tools, assisting novice developers in understanding coding best practices. Within the research community and industry, several efforts have emerged aiming to curate, refine, and utilize high-quality datasets for training LLMs for code generation tasks. A few notable examples include:

StarCoder: StarCoder (Li et al., 2023) is a 15B parameter model trained for code generation or completion. The training dataset, the Stack (Kocetkov et al., 2022b), has 1 trillion tokens sourced from a large collection of permissively licensed GitHub repositories.

Code Llama: Code Llama (Rozière et al., 2023) is an LLM capable of generating code from natural language prompts. It is a code specialized version of Llama2 (Touvron et al., 2023) developed by Meta. It was derived from Llama 2 by training it with code-specific datasets. Code Llama was released in sizes of 7B, 13B and 34B parameters. Each was trained with 500B tokens of code data.

WizardCoder: Most code LLMs like StarCoder have been trained on raw code data without instruction fine tuning. WizardCoder(Luo et al., 2023) with its complex instruction fine tuning by adapting the Evol-Instruct methods for coding tasks has shown to have improved performance for code generation.

These examples underscore the diversity of approaches and priorities when curating datasets for LLM training in code generation. Each dataset, with its unique strengths and focus areas, contributes to the broader goal of creating LLMs that are proficient, reliable, and versatile in gener-

ating code.

However, as highlighted previously, the quality of the training data is paramount. For LLMs to generate effective, efficient, and, most importantly, compilable code, they must be trained on high-quality, error-free codebases. This underscores the importance of curating and refining code datasets for LLM training, ensuring that the resulting models are both accurate and useful in real-world coding scenarios.

2.2.2 Instruction Fine Tuning

Instruction fine-tuning on code-specific LLMs typically requires training the model with code-related data, such as code snippets, documentation, or domain-specific programming languages. The goal is to enhance the model's understanding of programming concepts, idioms, and best practices. This approach has the potential to revolutionize software development by automating coding tasks, offering code recommendations, and aiding developers in writing efficient and accurate code. By refining the model's parameters and enhancing its knowledge, instruction fine-tuning enhances its performance in specialized contexts, making it a valuable tool for tailoring state-of-the-art language models to address specific research objectives and real-world challenges. Models like FLAN-T5 (Chung et al., 2022) have shown the improved performance of incorporating fine tuning. Even models like Code Llama have instruct variants which are instruction fined tuned for different objectives.

2.2.3 Quality of training dataset for LLMs

The efficacy of LLMs in the realm of natural language processing has been widely acknowledged. Their adeptness at understanding context, retaining long-term dependencies, and generating coherent text makes them exceptionally well-suited for various NLP tasks. As a result of this success, there has been a growing interest in harnessing the capabilities of LLMs for specialized tasks, particularly in the domain of code generation.

For LLMs to excel in code generation, the choice and quality of the training dataset (Gunasekar et al., 2023) become paramount. Unlike traditional NLP tasks that rely on textual data from books, articles, and websites, code generation requires a different kind of linguistic understanding—one rooted in the logic, structure, and semantics of programming languages.

Consequently, platforms that host vast repositories of code, such as GitHub and StackOverflow, have become pivotal in curating datasets for training LLMs in this domain. The diversity and richness of code available on these platforms, spanning multiple programming languages and tackling myriad computational problems, provide an ideal founda-

https://github.com/features/copilot

tion.

However, merely having access to vast amounts of code is not sufficient. The code needs to be of high quality, devoid of errors, and representative of good programming practices. Additionally, as we have emphasized earlier, the compilability of the code in the dataset is a significant factor. Training on non-compilable or poorly written code can inadvertently teach the model to reproduce such mistakes (Hasabnis, 2022a), detracting from the model's utility in real-world code generation scenarios causing performance degradation and thus adding noise (Sun et al., 2022). Providing high quality code datasets (Gunasekar et al., 2023) to models with lower number of parameters has shown to have comparative performance against models with higher number of parameters with lower quality code datasets.

Thus, meticulous curation, preprocessing, and validation of code datasets are essential to ensure that LLMs trained for code generation are not only proficient but also reliable in generating viable, efficient, and compilable code snippets.

3 CODE DATA VALIDATION

In this section, we delve into the validation phase of CompCodeVet. Our exploration commences with an evaluation of open-source code datasets for LLM training. Subsequently, we detail our methodology for collecting data tailored for the fine-tuning of an LLM, aimed at effective programming language classification.

3.1 Code Data Source

The dataset used in this work mostly comes from two opensource datasets: Stack and HPCorpus (Kadosh et al., 2023). The Stack (Kocetkov et al., 2022b), a 6.4 TB dataset of permissively licensed source code in 384 programming languages, included 54 GB of GitHub issues and repositorylevel metadata in the v1.2 version of the dataset.

3.2 Compilability Evaluation

To assess the compilability of each dataset, we selected random samples comprising 50k C code snippets and 50k C++ code snippets. For the compilation process, we employed the GNU Compiler Collection 12.3 (GCC-12.3). As per Definition 3.1, we opted not to account for linker errors in our analysis.

Definition 3.1. Compilation entails the conversion of source code into machine code or an intermediary representation. This process involves scrutinizing the code for various errors and subsequently generating the pertinent object files. A C or C++ code snippet is deemed "compilable" if the GCC compilation process concludes without returning

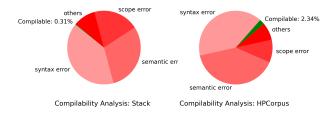


Figure 2. GCC compilation analysis results. The potion of non-compilation slides indicates the top categories.

any errors.

The results derived from our GCC compilation analysis are detailed in Figure 2. The compilability rates for the Stack and HPCorpus datasets are 0.31% and 2.34%, respectively. Additionally, a manual investigation of the compiler's report highlighted the predominant causes hindering the compilation of C/C++ code, as follows:

- syntax error: These are mistakes in the code structure, such as missing semicolons, parentheses, braces, or incorrect variable names. Syntax errors prevent the compiler from understanding the code.
- semantic error: errors such as undefined symbols and type errors.
- scope error: using variables outside their scope.

3.3 Programming Label Classification

The programming language (PL) of a code snippet not only dictates the compiler choice for compilation tests but also influences downstream tasks where the programming language is a significant factor. Through a meticulous inspection of compiler outputs, we discerned that the Stack dataset contains 34.7% of C code which is mislabeled. Specifically, out of the 50k samples tested from the Stack dataset, 17,350 samples of Objective-C are incorrectly labeled as C code.

Although the programming language of most code snippets is typically inferred from the file extension, there are scenarios in which this approach can lead to mislabeling:

• Shared File Extensions: Certain programming languages use identical file extensions. For instance, both C and Objective-C use the "*.h" extension for header files. However, their syntax and grammatical structures are vastly different. Incorporating mislabeled Objective-C code in the training dataset can adversely impact the model's performance.

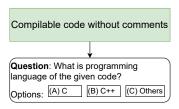


Figure 3. Example of processed data for fine-tuning the model for programming language classification.

 Code Snippets from Websites: Websites, such as Stack Overflow, often present code snippets without explicitly mentioning the file extension, making it challenging to automatically identify the correct programming language.

3.4 LLM Fine-tuning for Programming Language Classification

For the successful invocation of appropriate compilers in CompCodeVet, it's pivotal to accurately classify the programming language of given code snippets. To this end, we fine-tuned the Llama2-7b model specifically for the task of language classification.

Dataset Construction. We selected 50k compilable code snippets from the HPCorpus (Kadosh et al., 2023) dataset, using their associated PL labels as ground truth for the fine-tuning process. In our preprocessing steps, depicted in Figure 3, comments from the code snippets were systematically removed. Furthermore, for the creation of instructive data, we explicitly tagged each snippet with its respective PL label.

4 CODE DATA ENHANCEMENT

This section outlines the specifics of the dataset enhancement phase of CompCodeVet. In this phase, CompCodeVet uses a compiler-driven chain-of-thought strategy to turn non-compilable code into its compilable counterpart. As depicted in Figure 1, CompCodeVet harnesses the capabilities of compilers to steer the reasoning of LLMs, moving away from the conventional approach of relying on a larger-sized LLM.

4.1 Compiler-guided Chain-of-Thought Prompting

Wei et al. (2022) introduced the concept of CoT prompting for multi-step reasoning tasks. In contrast to directly prompting an LLM to rectify a non-compilable code snippet in one go, CompCodeVet systematically addresses each error reported by the compiler, one at a time. Once all errors from the initial compiler output have been addressed, CompCodeVet then compiles the modified code to ensure

its validity. If no new errors arise, the code is deemed compilable. If new errors are detected, the system repeats the iterative error-fixing process.

Although the steps in CompCodeVet's CoT framework can vary based on the specifics of the code and errors encountered, the principle remains consistent: address one error at a time. Rather than requesting the LLM to generate a universally compilable code, the focus is on rectifying individual, identified errors as shown in Figure 4. The prompts are structured as:

 P_0 [Given code C_0], please rectify [error e_0] identified in the compiler output.

 P_0 represents the initial prompt. This step can be mathematically represented as $C_1 = \operatorname{argmax} p(e_0|C_0)$, wherein C_1 is the updated code with error e_0 addressed. As long as there remain errors to be fixed, subsequent prompts are generated:

 P_1 [Given code C_1], please rectify [error e_1] indicated by the compiler.

Once the current error list is exhausted, the newly generated code undergoes compilation to verify its integrity. If additional errors are discovered, the described iterative process continues. However, to ensure that the process doesn't run indefinitely, we've imposed a maximum iteration limit of K. This constrains the number of iterations within CompCodeVet to not exceed K.

Once the current error list is exhausted, the newly generated code undergoes compilation to verify its integrity. If additional errors are discovered, the described iterative process continues until the code is fully rectified and compiled without errors.

To assure the process does not run infinitely, we set a maximum iteration limit K to limit the iteration within Comp-CodeVet does not go beyond K.

4.2 The BrokenComp-instruct Dataset

As previously illustrated, we leverage the combined strengths of LLMs and compilers to correct incomplete code, ensuring its compilability. To refine the LLM's reasoning capabilities at every juncture, it is essential to finetune the model, targeting the most prevalent compilation errors as identified in Section 3.2. To this end, we introduce an instructive dataset, termed BrokenComp-instruct. This dataset is meticulously curated to introduce a single error in each instruction, and it is paired with the corresponding compiler error output for guidance.

Dataset collection. Figure 5 illustrates the construction process of the BrokenComp-instruct dataset. To ascertain

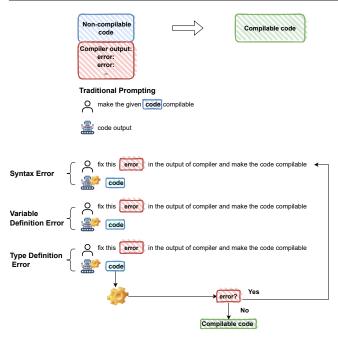


Figure 4. An illustration of compiler-guided chain-of-thought prompting in CompCodeVet.

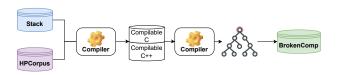


Figure 5. Construction Process of the BrokenComp Dataset.

the accuracy of the target-generated code in the instructions, we initiated our process by collecting compilable C/C++ code. This approach also addresses the concerns highlighted in Section 3.2 regarding the potential mislabeling of code data by datasets. We ended by collecting 50k compilable C code and 50k compilable C++ code. We also filtered out code with length beyond the common input token limits for popular LLMs.

Creating broken code. Starting with the 100k compilable code snippets, we introduced alterations as shown in Algorithm 1 based on the following operations using the Abstract Syntax Trees (AST):

- Constructed the code's AST and randomly selected variable leaf nodes, subsequently deleting the corresponding variable initialization.
- Identified a user-defined type through the AST and eliminated its definition.
- Introduced syntax errors by randomly eliminating operators and parentheses.

```
Algorithm 1 Introduce Only One Compilation error using AST
```

```
AST

1: AST \leftarrow \text{generateAST}(code) {Parse code to generate AST}

2: varNodes \leftarrow \text{findAllVariableLeafNodes}(AST)

3: if \text{length}(varNodes) > 0 then

4: randomNode \leftarrow \text{selectRandom}(varNodes)

5: if \text{hasInitialization}(randomNode) then

6: \text{deleteInitialization}(randomNode)

7: end if

8: end if

9: typeNodes \leftarrow \text{findAllUserDefinedTypes}(AST)

10: if \text{length}(typeNodes) > 0 then
```

11: randomTypeNode \leftarrow selectRandom(typeNodes)

12: deleteTypeDefinition(randomTypeNode)

13: **end if**

14: $modifiedCode \leftarrow traceASTtoCode(AST)$ {Convert modified AST back to code}

15: **return** modifiedCode

These steps ensured the creation of a dataset that would challenge and thus refine the LLM's capacity to rectify and compile broken code.

Creating fine-tuning instructions. To optimize the performance of large language models (LLMs) in handling broken code scenarios, we crafted specific instructions paired with the broken code and the associated compiler output. This approach ensures that the LLMs gain a nuanced understanding of how to rectify common errors present in the code.

```
# Instruction:
{"Fix the compiler error of the given PL code: compiler error"}
# Input:
{broken code}
# Response:
{original compilable code}
```

5 EVALUATION

In this section, we present and analyze the outcomes derived from our proposed approach, shedding light on its efficacy and implications.

5.1 Experimental Platform

Throughout both the code validation and data enhancement stages, we adopted parameter-efficient fine-tuning utiliz-

ing the Low-Rank Adaptation (LoRA) (Hu et al., 2021) method, sidestepping the need for comprehensive model fine-tuning. For the task of programming language classification, we selected the official release of Llama2-7b. Conversely, for generating compilable code, our choice was the official iteration of CodeLlama2-instruct-7b. All model interactions were facilitated using the Huggingface Transformers toolkit (Wolf et al., 2020). During the training phase, the models were fine-tuned to align their outputs with the reference responses, leveraging the crossentropy loss. We followed the Alpaca LoRA project for hyper-parameter settings. All training experiments share the same software environment and were carried out on 2 Nyidia A100 40GB GPUs.

5.2 Code Validation Results

During the code validation phase, our focus was on assessing the efficacy of programming language classification. The prompts for the test mirrored the format used during fine-tuning, as illustrated in Figure 3. We gauged accuracy by comparing the model's output against the ground truth labels.

Test dataset. Our test dataset was curated using Google BigQuery to crawl code from GitHub, encompassing ten widely-used programming languages: C, C++, Python, Objective-C, Assembly, Java, Go, C#, Ruby, and R. Each language category comprises of 1,000 samples.

Test results. Our model was benchmarked against the out-of-the-box Llama2-7b, GPT3.5-turbo, and GPT-4. We employed OpenAI's API for inference on the last two mentioned models. Across the board shown in Table 1, all models showcased impressive metrics in terms of precision, recall, and F1 score. This underscores the adeptness of LLMs in discerning patterns from their training data. Notably, CompCodeVet's fine-tuned Llama2-7b out-performed its pre-trained counterpart. This improvement can be attributed to the model's capability to emphasize language-specific keywords at distinct tokens and its fine-tuning on high-quality data. While GPT-4 delivered top-tier performance, CompCodeVet achieved comparably impressive results with a significantly more compact model (1.76 trillion vs. 7 billion).

Upon examining the misclassified cases, we observed that most of these cases occur between C and C++. These samples posed challenges even for human developers. Such ambiguity can be ascribed to the intricate relationship and shared syntax between the two programming languages.

Ablation Study. Compared to the pre-trained Llama2-7b model, the variant of CompCodeVet was fine-tuned with keywords in programming languages as special tokens. The results in Table 2 suggest the importance of compiler

Table 1. Programming Language Labeling Results

Model	Precision	Recall	F1
CompCodeVet	0.96	0.94	0.95
Llama2-7b	0.93	0.92	0.92
GPT3.5-turbo	0.95	0.95	0.95
GPT4	0.97	0.96	0.96

guidance when applying LLMs in code analysis tasks and the importance of high-quality data. We followed the same process to fine-tune the pre-trained Llama2-7b model with and without the special tokens for an ablation study. We also control the training data size in this ablation study.

Table 2 presents a comparison between Llama2-7b models, with and without the addition of special tokens (ST), across different dataset sizes. Our analysis indicates that incorporating special tokens positively influences the precision score, though no discernible impact is observed on the recall score. This might be attributed to the fact that the special tokens don't encompass all the programming language keywords present in the test set, particularly benefiting the identification of C and C++ codes.

As seen from Table 2, the size of the fine-tuning dataset has a noticeable impact on the model's performance. The results demonstrate a clear correlation between the volume of training data and the model's effectiveness across both precision and F1 scores.

During our manual evaluation, we observed that pre-trained models, when fine-tuned with keywords as special tokens, demonstrated enhanced proficiency in differentiating between C and C++ code with keywords as special tokens.

Table 2. Ablation study results comparing Llama2-7b (Llama2) with and without special tokens(ST), fine-tuned with various dataset sizes.

Dataset	Precision	Recall	F1
1K	0.93	0.92	0.92
1K	0.91	0.92	0.91
10k	0.92	0.91	0.91
10k	0.91	0.91	0.91
30k	0.95	0.91	0.93
30k	0.93	0.91	0.92
50k	0.95	0.93	0.94
50k	0.92	0.93	0.92
	1K 1K 10k 10k 30k 30k 50k	1K 0.93 1K 0.91 10k 0.92 10k 0.91 30k 0.95 30k 0.93 50k 0.95	1K 0.93 0.92 1K 0.91 0.92 10k 0.92 0.91 10k 0.91 0.91 30k 0.95 0.91 30k 0.93 0.91 50k 0.95 0.93

5.3 Code Enhancement Results

The primary objective of the code enhancement phase within CompCodeVet is to transform non-compilable code into a compilable state. Listing 2 shows an output example of CompCodeVet with Listing 1 as input.

```
Listing 1. Instance of non-compilable code in HPCorpus

void comm_clean()

{
    comm_close();
    if (port_name)
        free(port_name);
    port_name = NULL;
}
```

Listing 2. Instance of compilable code generated by CompCode-Vet

```
#include <stdlib.h>
char *port_name = NULL;
void comm_clean()
{
    comm_close();
    if (port_name)
        free(port_name);
    port_name = NULL;
}
```

Test dataset. Our test dataset was curated by amassing 5k C and 5k C++ code snippets during the development of the BrokenComp Dataset, as depicted in Figure 5. For efficient inference with CompCodeVet, we capped the iteration limit, K, at three.

Test results. We compared the performance of Comp-CodeVet with the out-of-the-box CodeLlama-7b-instruct, starchat-alpha, and WizardCoder-15b. All the models are open-source LLMs specifically pre-trained with code data. The tests were performed with the same prompts in Figure 4.

Table 3. Comparison of Model Compilability(Comp) between CompCodeVet, CodeLlama-7b-instruct(CodeLlama), StarChat Alpha(StarChat), and WizardCoder-15B(WizardCoder)

Model	Comp-C (%)	Comp-C++ (%)
CompCodeVet	10.6	8.4
CodeLlama	2.1	1.5
StarChat	0.2	0.1
WizardCoder	0.1	0.1

Ablation Study. CompCodeVet refines the code in an iterative manner, drawing insights from the compiler's feedback. This iterative approach, however, poses a risk of infinite loops if the LLM within the CoT framework either fails to rectify existing errors or inadvertently introduces new ones. This scenario resembles signal amplification, albeit with escalating errors instead of signals. In this context, setting an appropriate iteration limit of K becomes

paramount. Table 4 reveals that K=4 yields the most favorable results on a compact test set consisting of 1k samples. However, a setting of K=3 provides outcomes that are nearly as optimal. Given considerations of computational efficiency, we opted for K=3 for the experiments presented in Table 3.

Table 4. Compilability (Comp) of Code at Different Iterations (K)

K	Comp-C (%)	Comp-C++ (%)
1	3.4	2.7
2	6.8	3.1
3	9.4	7.9
4	9.4	8.1
5	9.4	8.1

6 RELATED WORKS

Standard code datasets (Kocetkov et al., 2022a; Li et al., 2022) suffer from several drawbacks, such as incompleteness, inaccurate information, and ambiguity (e.g., incomplete snippets, incorrect labels, non-compilable), which would impact the quality of trained machine learning models for code. The field of code cleaning and data preprocessing has seen significant research and development efforts over the years. The research on using LLMs for code cleaning includes efforts to transform natural language descriptions into data cleaning code. These endeavors have shown promising results in simplifying code cleaning tasks. CodeBERT (Feng et al., 2020) is a pretrained model that can understand and generate code from natural language comments, thus streamlining data preprocessing through human-readable descriptions. In recent years, a successful approach to improve model performance has been to scale up the model parameters and training data. (Gunasekar et al., 2023) has clearly shown us that improving the quality of the training dataset while using lesser number of parameters and fewer tokens can have on par performance with state-of-the-art LLMs which use trillions of parameters and tokens. Alternately, the field of software engineering has extensive set of tools to measure code quality, complexity, among other metrics (Ludwig et al., 2017; McCabe, 1976), and Hasabnis et al. (Hasabnis, 2022a) has recently employed these tools to evaluate quality of open-source code repositories on GitHub (Hasabnis, 2022b).

7 CONCLUSION

In this work, we presented CompCodeVet, a novel compiler-guided approach to leverage Large Language Models (LLMs) for the tasks of code validation and enhancement. Our main contribution lies in employing the

power of compilers as teachers, steering away from the conventional reliance on larger LLMs, and emphasizing the importance of multi-step reasoning in generating and validating code.

Our findings underscore the inefficiencies in current LLMs when faced with complex reasoning tasks, particularly in the context of code generation and comprehension. CompCodeVet's unique approach to employing a chain of thought (CoT) with the compiler's feedback loop showed promise in addressing such challenges, emphasizing the benefit of stepwise error resolution.

REFERENCES

- Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., and Sutton, C. Program synthesis with large language models, 2021.
- Bender, E. M., Gebru, T., McMillan-Major, A., and Shmitchell, S. On the dangers of stochastic parrots: Can language models be too big? FAccT '21, pp. 610–623, New York, NY, USA, 2021. Association for Computing Machinery. doi: 10.1145/3442188.3445922.
- Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901, 2020.
- Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code, 2021a.
- Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N.,

- Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021b. URL https://arxiv.org/abs/2107.03374.
- Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N., Prabhakaran, V., Reif, E., Du, N., Hutchinson, B., Pope, R., Bradbury, J., Austin, J., Isard, M., Gur-Ari, G., Yin, P., Duke, T., Levskaya, A., Ghemawat, S., Dev, S., Michalewski, H., Garcia, X., Misra, V., Robinson, K., Fedus, L., Zhou, D., Ippolito, D., Luan, D., Lim, H., Zoph, B., Spiridonov, A., Sepassi, R., Dohan, D., Agrawal, S., Omernick, M., Dai, A. M., Pillai, T. S., Pellat, M., Lewkowycz, A., Moreira, E., Child, R., Polozov, O., Lee, K., Zhou, Z., Wang, X., Saeta, B., Diaz, M., Firat, O., Catasta, M., Wei, J., Meier-Hellstern, K., Eck, D., Dean, J., Petrov, S., and Fiedel, N. Palm: Scaling language modeling with pathways, 2022.
- Chung, H. W., Hou, L., Longpre, S., Zoph, B., Tay, Y., Fedus, W., Li, Y., Wang, X., Dehghani, M., Brahma, S., Webson, A., Gu, S. S., Dai, Z., Suzgun, M., Chen, X., Chowdhery, A., Castro-Ros, A., Pellat, M., Robinson, K., Valter, D., Narang, S., Mishra, G., Yu, A., Zhao, V., Huang, Y., Dai, A., Yu, H., Petrov, S., Chi, E. H., Dean, J., Devlin, J., Roberts, A., Zhou, D., Le, Q. V., and Wei, J. Scaling instruction-finetuned language models, 2022.
- Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.
- Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. Codebert: A pre-trained model for programming and natural languages, 2020.
- Gunasekar, S., Zhang, Y., Aneja, J., Mendes, C. C. T., Giorno, A. D., Gopi, S., Javaheripi, M., Kauffmann, P., de Rosa, G., Saarikivi, O., Salim, A., Shah, S., Behl, H. S., Wang, X., Bubeck, S., Eldan, R., Kalai, A. T., Lee, Y. T., and Li, Y. Textbooks are all you need, 2023.

- Hasabnis, N. Are machine programming systems using right source-code measures to select code repositories? In *Proceedings of the 6th International Workshop on Machine Learning Techniques for Software Quality Evaluation*, MaLTeSQuE 2022, pp. 11–16, New York, NY, USA, 2022a. Association for Computing Machinery. ISBN 9781450394567. doi: 10.1145/3549034.3561176. URL https://doi.org/10.1145/3549034.3561176.
- Hasabnis, N. Gitrank: A framework to rank github repositories. In 2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR), pp. 729–731, 2022b. doi: 10.1145/3524842.3528519.
- Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. Lora: Low-rank adaptation of large language models, 2021.
- Kadosh, T., Hasabnis, N., Mattson, T., Pinter, Y., and Oren, G. Quantifying openmp: Statistical insights into usage and adoption, 2023.
- Kocetkov, D., Li, R., Allal, L. B., Li, J., Mou, C., Ferrandis, C. M., Jernite, Y., Mitchell, M., Hughes, S., Wolf, T., et al. The stack: 3 TB of permissively licensed source code. arXiv preprint arXiv:2211.15533, 2022a.
- Kocetkov, D., Li, R., Ben Allal, L., Li, J., Mou, C., Muñoz Ferrandis, C., Jernite, Y., Mitchell, M., Hughes, S., Wolf, T., Bahdanau, D., von Werra, L., and de Vries, H. The stack: 3 tb of permissively licensed source code. *Preprint*, 2022b.
- Lattner, C. and Adve, V. Llvm: A compilation framework for lifelong program analysis & transformation. In Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, CGO '04, pp. 75, USA, 2004.
- Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.
- Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.
- Ludwig, J., Xu, S., and Webber, F. Compiling static software metrics for reliability and maintainability from github repositories. In 2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC), pp. 5–9, 2017. doi: 10.1109/SMC.2017.8122569.

- Luo, Z., Xu, C., Zhao, P., Sun, Q., Geng, X., Hu, W., Tao, C., Ma, J., Lin, Q., and Jiang, D. Wizardcoder: Empowering code large language models with evol-instruct. arXiv preprint arXiv:2306.08568, 2023.
- McCabe, T. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976. doi: 10.1109/TSE.1976.233837.
- OpenAI. Gpt-4 technical report, 2023.
- Roziere, B., Lachaux, M.-A., Chanussot, L., and Lample, G. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems*, 33, 2020.
- Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., and Synnaeve, G. Code llama: Open foundation models for code, 2023.
- Sun, Z., Li, L., Liu, Y., Du, X., and Li, L. On the importance of building high-quality training datasets for neural code search. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pp. 1609–1620, New York, NY, USA, 2022. Association for Computing Machinery. doi: 10.1145/3510003.3510160.
- Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Ferrer, C. C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P. S., Lachaux, M.-A., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E. M., Subramanian, R., Tan, X. E., Tang, B., Taylor, R., Williams, A., Kuan, J. X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., and Scialom, T. Llama 2: Open foundation and fine-tuned chat models, 2023.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. Attention is all you need. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R. (eds.), Advances in Neural Information Processing Systems, volume 30, 2017.

- Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35: 24824–24837, 2022.
- Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Scao, T. L., Gugger, S., Drame, M., Lhoest, Q., and Rush, A. M. Huggingface's transformers: State-of-the-art natural language processing, 2020.