

A Chain of AI-based Solutions for Resolving FQNs and Fixing Syntax Errors in Partial Code

Qing Huang, Jiahui Zhu, Zhenchang Xing, Huan Jin, Changjing Wang, Xiwei Xu

Abstract—API documentation, technical blogs and programming Q&A sites contain numerous partial code that can be reused in programming tasks, but often these code are uncompileable due to unresolved names and syntax errors. To facilitate partial code reuse, we propose the Partial Code Reuse Chain (PCR-Chain) for resolving fully-qualified names (FQNs) and fixing last-mile syntax errors in partial code based on a giant large language model (LLM) like ChatGPT. Methodologically, PCR-Chain is backed up by the underlying global-level prompt architecture (which combines three design ideas: hierarchical task breakdown, prompt composition, and a mix of prompt-based AI and non-AI units) and the local-level prompt design. Technically, we propose PCR-Chain, which employs in-context learning rather than symbolic, costly training methods. Experimental results demonstrate that in dynamically-typed languages (Python), PCR-Chain outperforms current state-of-the-art (SOTA) 5% accuracy like RING. For statically-type languages (Java), our approach achieves high accuracy of 80.5% in resolving both non-FQNs and last-mile syntax errors, surpassing SOTA methods (RING) that can only address last-mile syntax errors. The correct execution of the unit, module, and PCR-Chain demonstrates the effectiveness of the prompt design, composition, and architecture and opens up possibilities for building software engineering tools based on LLMs, replacing traditional program analysis methods.



1 INTRODUCTION

PARTIAL code from sources like API documentation and Q&A site is commonly reused in programming tasks [1]–[6]. However, a recent study analyzing 491,906 posts on Stack Overflow found that over 90% of the partial code is uncompileable [7] due to non-fully qualified names (non-FQNs) [8]–[12], and last-mile syntax errors [13] such as unbalanced parentheses, missing commas. This presents a significant challenge for developers, limiting the usefulness of partial code and wasting time and effort.

The Java code in Fig. 1-a contains non-fully qualified names (non-FQNs) and last-mile syntax errors that prevent it from compiling correctly. For example, the use of “StringUtils” on line 6 is a non-FQN that cannot be resolved by the compiler, resulting in a “cannot find symbol” error. In addition, the comma error on line 4 is a last-mile syntax error that causes an “unexpected token” error during compilation. These errors collectively prevent the code from running as intended.

When working with partial code, developers typically perform two manual steps to repair it: FQN inference (to infer missing fully qualified names) and syntax error fix (to check and correct last-mile syntax errors). However, this process can be time-consuming and error-prone, resulting in inefficiencies during the development process. To address this issue, many partial program analysis techniques are proposed, including dictionary lookup strategies based on

a symbolic knowledge base [14], [15] to infer FQNs of non-FQN types, and symbolic-based approaches [16]–[19] to fix syntax errors. However, these methods may encounter out-of-vocabulary (OOV) failures, requiring extensive domain-specific knowledge.

Unlike symbolic-based approaches, which have finite knowledge, recent studies [20]–[23] propose treating code as text and training large language models (LLMs) on near-infinite code text. These LLM-based approaches, such as CodeBERT [24], CodeT5 [25], Copilot [26], can effectively reduce out-of-vocabulary (OOV) failures. For example, some researchers have used LLM-based approaches to infer FQNs of non-FQN types [8] and fix last-mile syntax errors [27]. However, these approaches are typically heavyweight supervised learning, requiring large amounts of data and computing resources for efficient model training and gradient updating. In contrast, in-context learning (ICL) is a lightweight unsupervised approach proposed by Brown et al. [28] that prompts LLMs to learn from a few examples of specific downstream tasks in context and complete those tasks by mimicking those examples [29]–[32], without gradient updates [33]–[35]. Inspired by this, we consider leveraging LLMs’ ICL capability to make partial code compileable.

However, directly asking LLMs to make partial code compileable is challenging, as LLMs may only fix some syntax errors but not infer FQNs of non-FQN types. Since LLMs are language models, both FQNs and non-FQNs represent correct type names from the point of view of language features, so LLMs may still consider code with non-FQNs to have no compilation errors in the absence of importing packages. However, this is incorrect. Even if code has no syntax errors, it still may not compile because it contains non-FQNs. As shown in Fig. 1-A, when directly querying the LLM with the prompt “make this code compileable”, the LLM still responds with uncompileable code.

Instead of directly querying LLMs in one step to make

- Q. Huang, J. Zhu, C. Wang are with School of Computer Information Engineering, Jiangxi Normal University, China.
- H. Jin is with Jiangxi University of Technology’s School of Information Engineering, China.
- Q. Huang and J. Zhu are co-first authors, C. Wang is the corresponding author (wcj@jxnu.edu.cn)
- Z. Xing is with Australian National University, Australia.
- X. Xu is with Data61, CSIRO.

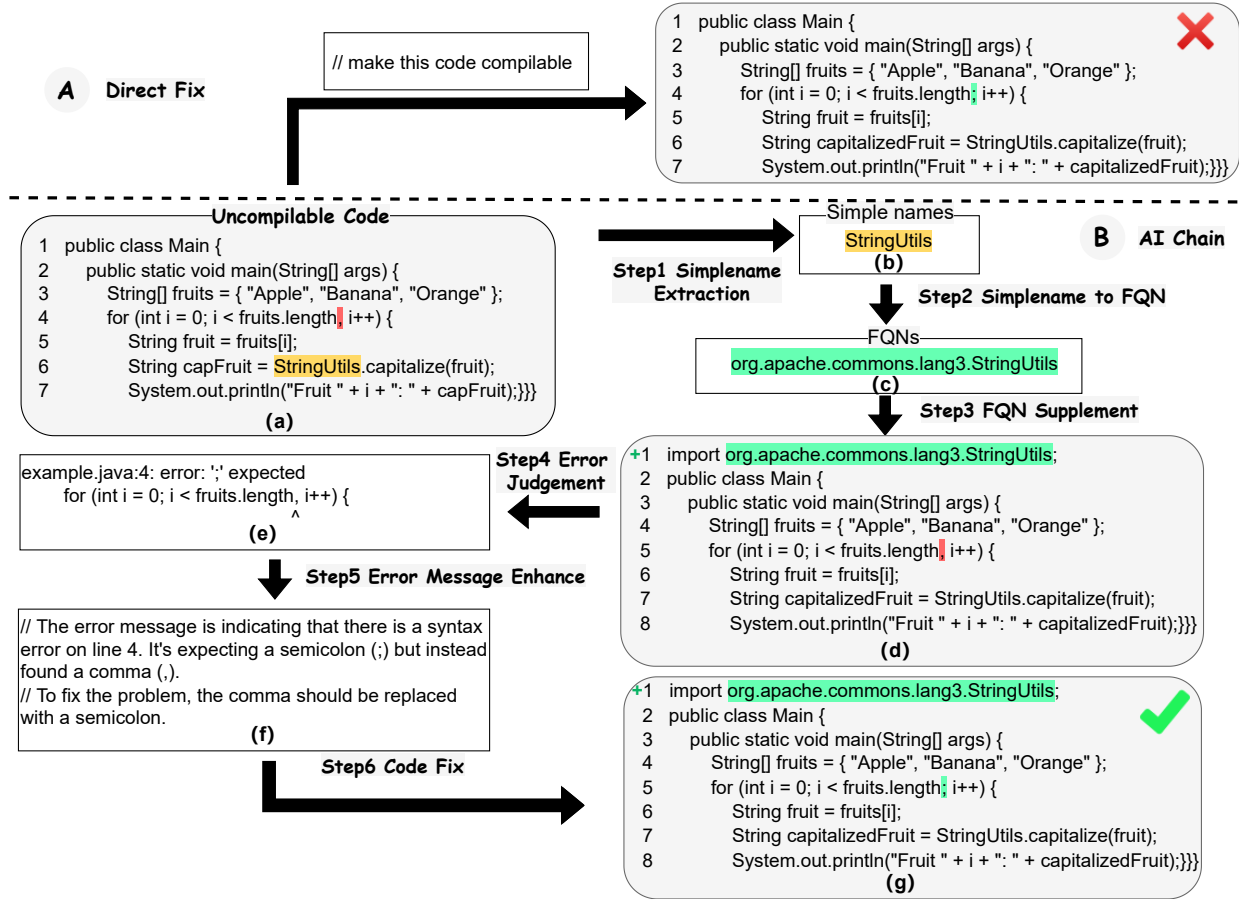


Fig. 1: Fix errors directly vs. fix errors with AI Chain

partial code compilable, a more effective approach is to design a chain of thought (CoT) with multiple steps [36]–[38]. As shown in Fig. 1-B, this CoT consists of two main parts: FQN Inference and Syntax Error Fix. The FQN Inference is further divided into three steps: Simplename Extraction, Simplename to FQN, and FQN Supplement. These steps comprehensively infer FQNs for simple names in the code. The Syntax Error Fix is decomposed into three steps: Error Judgement, Error Message Enhance, and Code Fix. Together, these steps enhance the error messages and provide solutions for fixing the code, thus resolving syntax errors. Overall, the CoT includes six key steps that make it an effective solution for making partial code compilable.

However, this CoT still has limitations due to its single prompt implementation, which can lead to error accumulation and an “epic” prompt with too many responsibilities. This can lead to errors and difficulties in controlling and improving the prompts. To overcome these limitations, we adopt the principle of single responsibility in software engineering and decompose the CoT into an AI chain, with each step corresponding to a separate AI unit. We develop an effective prompt for each AI unit, which performs a separate LLM call. As shown in Fig. 1-B, this AI chain can interact with LLM step by step, thus solving both non-FQN and last-mile syntax error.

We evaluate the effectiveness of our approach, Partial Code Reuse Chain (PCR-Chain), through a series of experiments. We first test the accuracy of each Unit and Module, with units achieving accuracy rates ranging from 92.1% to

95.7%, indicating the effectiveness of prompt design. We also verify the accuracy of our key modules, FQN Inference and Syntax Error Fix, achieving rates of 80.5% and 98%, respectively.

We then compare PCR-Chain to state-of-the-art (SOTA) methods such as BIFI [39], CURE [27] and RING [40], in both dynamically-typed (Python) and statically-typed (Java) programming languages. In Python, PCR-Chain outperforms SOTA LLM-based methods such as BIFI by 13.7% accuracy and ICL-based methods such as RING by 5%. In Java, our approach accurately solves both non-FQNs and last-mile syntax errors with an accuracy of 80.5%, while SOTA methods such as CURE and RING could only solve the latter. We also conduct an ablation experiment to investigate the effectiveness of our AI Chain design principles and find that our design was reasonable. Besides, we explore the sensitivity of our approach to prompt forms and find that our method’s accuracy remains stable under different prompt types.

From our study, we derive three AI chain design principles, including hierarchical task decomposition, unit composition, and a combination of AI and non-AI units, that can serve as guidelines for future software engineering projects.

This paper makes the following contributions:

- To the best of our knowledge, we are the first to propose the prompt architecture, which combines three global design ideas: hierarchical task breakdown, prompt composition, and a mix of prompt-based AI and non-AI units, rather than a simple AI chain.

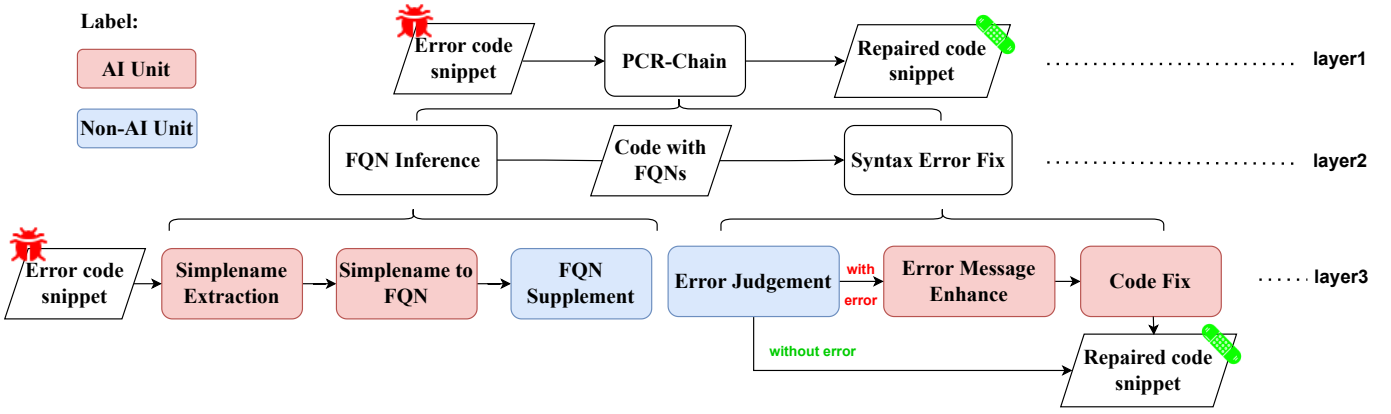


Fig. 2: Overall Framework of PCR-Chain

- Instead of creating an “epic” prompt to implement CoT, we split the CoT into an AI chain, with each step corresponding to a separate AI unit, making the design of the AI chain more reasonable and easier to optimize and control.
- We stand on the giant LLM’s shoulder and use a lightweight in-context learning method to resolve FQNs and fix syntax errors in partial code.
- The successful completion of the unit, module, and PCR-Chain demonstrates the efficacy of the prompt design, composition, and architecture, to resolve FQNs and fix last-mile syntax errors.

2 APPROACH

Partial code reuse can be a difficult task due to non-FQNs and last-mile syntax errors. Non-FQNs arise when program entities lack declarations, while last-mile syntax errors occur due to minor mistakes like missing commas or unbalanced parentheses. To address these challenges, we propose a new approach called the Partial Code Reuse Chain (PCR-Chain). Our approach leverages the vast amount of code knowledge obtained during the training of large language models (LLMs) like ChatGPT¹. PCR-Chain is designed to simulate the human thought process by breaking down the task of making partial code compilable into single-responsibility sub-problems and designing functional units. These units are then connected in a serial or conditional structure to interact with the underlying LLM. Unlike fine-tuning LLM, which requires significant effort in data gathering, preprocessing, annotation, and model training, PCR-Chain only needs to consider task characteristics, data properties, and information flow, utilizing the capabilities of ChatGPT.

2.1 Hierarchical Task Breakdown

As the code contains multiple errors (i.e., non-FQNs and last-mile syntax errors), it is not easy for LLM to resolve all errors in single LLM call. As shown in Fig. 1-A, when using a single instruction “make this code compilable” to call a single LLM, it only fixes the last-mile syntax error, but not fixes the non-FQN error. To address this issue, we need to make the instruction more informative and break

it down into several sub-instructions, each executed by a separate LLM call, to solve the code error step by step, as shown in Fig. 1-B. To facilitate a reasonable decomposition, we re-analyze the code in Fig. 1(a) and identify that the issue of partial code reuse can be divided into two sub-modules: the first sub-module is **FQN Inference**, and the second sub-module is **Syntax Error Fix** (see Fig. 2-layer 2).

2.2 Hierarchical Module Decomposition

In the overall framework of PCR-Chain, as shown in Fig. 2-layer 3, the functional units are classified into two categories: AI units that utilize fuzzy reasoning based on LLM, and non-AI units that follow pre-defined rules or logic. The design of these modules/units adheres to two important software engineering principles, namely separation of concerns and single responsibility, and employs a modular design structure.

Two key modules in the PCR-Chain approach are the **FQN Inference** module and the **Syntax Error Fix** module.

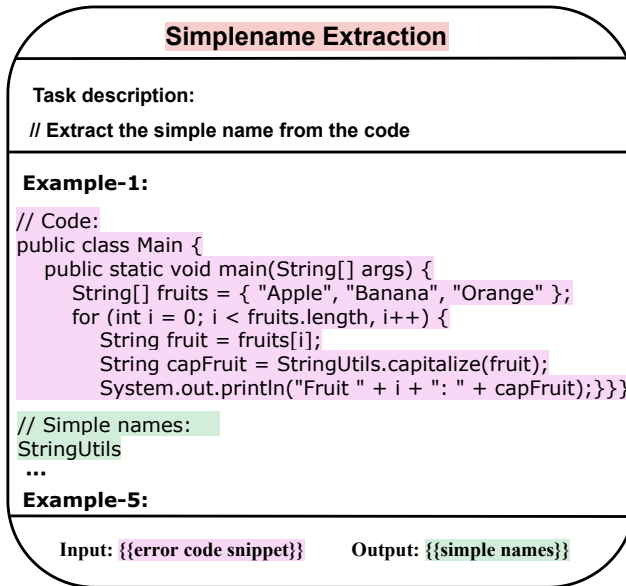
The **FQN Inference** module is responsible for identifying simple names in the code and inferring them as fully qualified names (FQNs), then completing the FQNs in the code. This module consists of two AI units, namely *Simple Extraction* and *Simple to FQN*, and a non-AI unit called **FQN Supplement**. To accomplish this task, the **FQN Inference** module first extracts simple names from the code using the *Simple Extraction* unit. It then infers the FQNs of the simple names using the *Simple to FQN* unit, then combining the inferred FQNs with the original code using the **FQN Supplement** unit.

On the other hand, the **Syntax Error Fix** module aims to enhance error messages and fix syntax errors in code. This module comprises two AI units, *Error Message Enhance* and *Code Fix*, as well as a non-AI unit called **Error Judgement**. The **Error Judgement** unit employs a compiler to assess whether there are any errors in the code, and if so, retrieves the error messages for the code. The *Error Message Enhance* unit enhances the error messages and offers solutions for fixing them. Finally, the *Code Fix* unit leverages the enhanced error messages to rectify the syntax errors in the code.

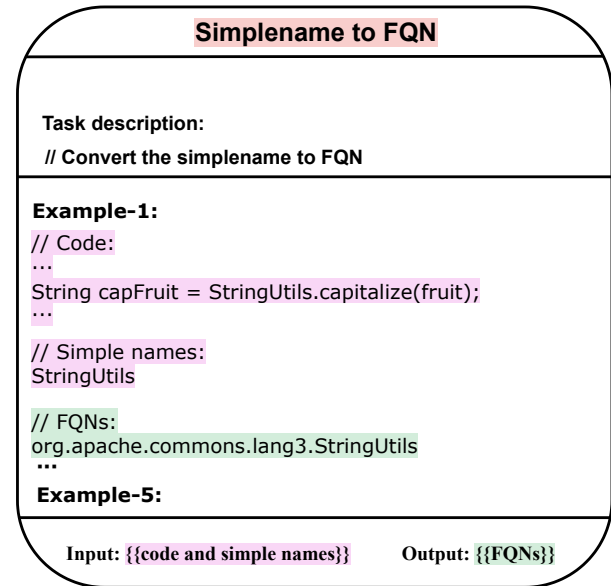
2.3 Prompt Design for AI-Units

The AI units in PCR-Chain are designed to activate LLMs for downstream tasks. There are two methods for imple-

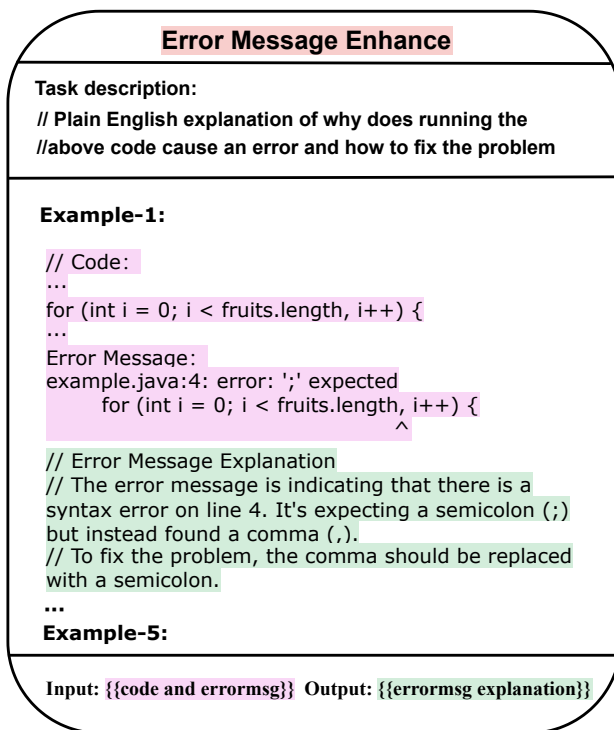
1. <https://openai.com/blog/introducing-chatgpt-and-whisper-apis>



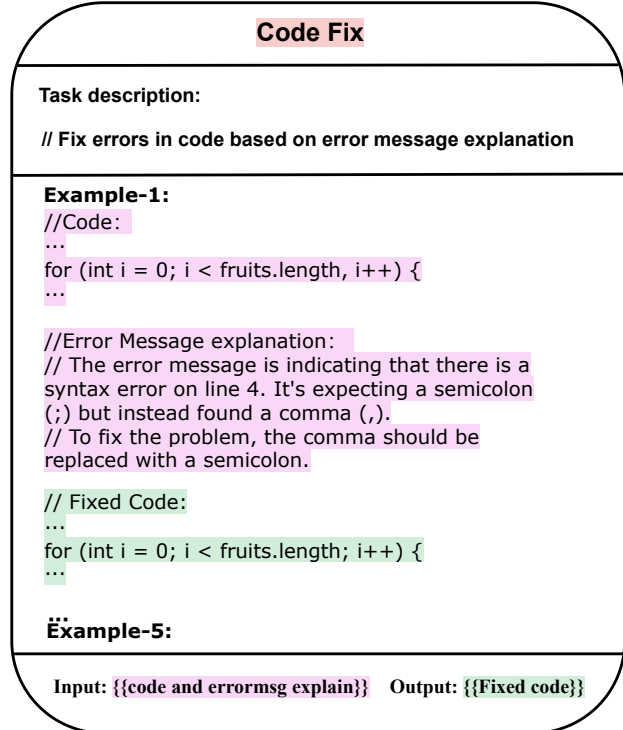
(a)



(b)



(c)



(d)

Fig. 3: All AI units design in approach

menting this: supervised fine-tuning and in-context learning. Supervised fine-tuning involves adjusting the weights of the LLMs using a labeled dataset that is specific to the task. On the other hand, in-context learning involves conditioning the LLMs on a task description along with some examples of the task, even if it is an unseen one. While both methods have their own benefits, in-context learning is more straightforward to adopt as it only requires a task instruction along with zero or several examples. In contrast, fine-tuning requires collecting data and making

updates to the model, which can be more time-consuming and resource-intensive.

Taking all of these factors into account, our AI units are developed using in-context learning. This approach allows for simpler implementation while still providing effective and efficient performance on a wide range of tasks.

An empirical study [11] found that descriptions and examples are critical for in-context learning. To standardize our prompt design, we develop a generic template that includes a task description and input-output examples. The

Simplename Extraction unit serves as an example of the template’s structure, as shown in Fig. 3(a). The template includes a task description (e.g., “Extract the simple names in the code”), followed by five input-output examples (e.g., Input: “...String capFruit = StringUtils.capitalize(fruit)...”, Output: “StringUtils”). After being provided with a code, the LLM extracts the simple names in the code.

Noted that in this work, we pre-select five examples that are used for all AI units. While the model adaptability generally increases with more examples [11], Min et al. [41] have shown that additional examples beyond four results in limited increase in accuracy.

In the following sections, we describe the prompt designs for each of the four units in the PCR-Chain approach.

2.3.1 *Simplename Extraction Unit*

This AI unit is responsible for extracting the simple names from the given code. To prompt the LLM to perform this task, a generic template is used, as shown in Fig. 3(a), with a task description of “Extract the simple names in the code”, five input-output examples, and a placeholder to input the code to be extracted simple names.

2.3.2 *Simplename to FQN Unit*

This AI unit is responsible for inferring the simple names to FQNs. To prompt the LLM to perform this task, a generic template is used, as shown in Fig. 3(b), with a task description of “Convert the simplename to FQN”, five input-output examples, and a placeholder to input the code and simplenames to convert to FQNs.

2.3.3 *Error Message Enhance Unit*

This AI unit is responsible for enhancing the error message from compiler. To prompt the LLM to perform this task, a generic template is used, as shown in Fig. 3(c), with a task description of “Plain English explanation of why does running the above code cause an error and how to fix the problem”, five input-output examples, and a placeholder to input the code and corresponding error message.

2.3.4 *Code Fix Unit*

This AI unit is responsible for fixing errors based on error message explanation. To prompt the LLM to perform this task, a generic template is used, as shown in Fig. 3(d), with a task description of “Fix errors in code based on error message explanation”, five input-output examples, and a placeholder to input the code and error message explanation.

2.4 Running Example

To illustrate how the different units work together and how the data is transformed among them, we present an example using a Java code that contains a non-FQN error and a last-mile syntax error, as shown in Fig. 1(a). To start, the Java code is input into the *Simplename Extraction* unit, which identifies the simple names and extracts them. The output of this unit is shown in Fig. 1(b). Next, the code and simple names are fed into the *Simplename to FQN* unit, which infers the FQNs for the identified simple names. The output of this unit is shown in Fig. 1(c). Subsequently, the code and FQNs

will be input into the FQN Supplement unit, it will combine code and FQNs. The output of this unit is shown in fig 1(d). Then, the code with FQNs is input into the *Error Judgement* unit, which utilizes a compiler to detect errors and provide error messages. The output of this unit is shown in Fig. 1(e). After that, the code and corresponding error message are input into the *Error Message Enhance* unit, which provides plain English explanations of why the code produces the error and how to fix the problem. The output of this unit is shown in Fig. 1(f). Finally, the code and the explanation of the error message are input into the *Code Fix* unit, which fixes the errors and outputs the error-free code, as shown in Fig. 1(g).

3 EXPERIMENTS SETUP

In this section, we present our research questions (RQs) that evaluate the effectiveness of our approach. Additionally, we describe our experimental setup, including data preparation, baselines, and evaluation metrics.

3.1 Research Questions

We conducted the following research questions to evaluate PCR-Chain’s performance in partial code reuse.

- RQ1: What is the quality of each unit or module in PCR-Chain?
- RQ2: How well does PCR-Chain perform in partial code reuse?
- RQ3: How effective are the AI Chain and error message enhance strategies employed in PCR-Chain?
- RQ4: How sensitive are the prompts in PCR-Chain to different forms?

3.2 Data Preparation

To evaluate the performance of our approach, we collect two distinct datasets: one for Python and one for Java. For the Python dataset, we randomly select 200 syntactically invalid code snippets from the dataset used by the state-of-the-art syntax repair tool for Python, BIFI [39]. These code snippets were collected from real GitHub repositories.

For the Java dataset, we first crawl 30,000 posts from Stack Overflow that are tagged with “java”, and collect the highest up-voted answer from each post. Then we extract Java code by identifying the text between the code block HTML tags, i.e., `<pre><code>`, and identify code with errors using a compiler. We manually filter out 200 error codes that have non-Fully Qualified Names (FQNs) and last-mile syntax errors.

In summary, we prepare two distinct datasets:

- Python dataset: 200 error Python code that all contain last-mile syntax issues.
- Java dataset: 200 error Java code that all contain non-FQNs and last-mile syntax issues.

3.3 Baselines

To evaluate the effectiveness of PCR-Chain’s overall design and module designs, we compare it with the state-of-the-art methods that reuse partial code. These methods fall into two main approaches: LLM-based and ICL-based.

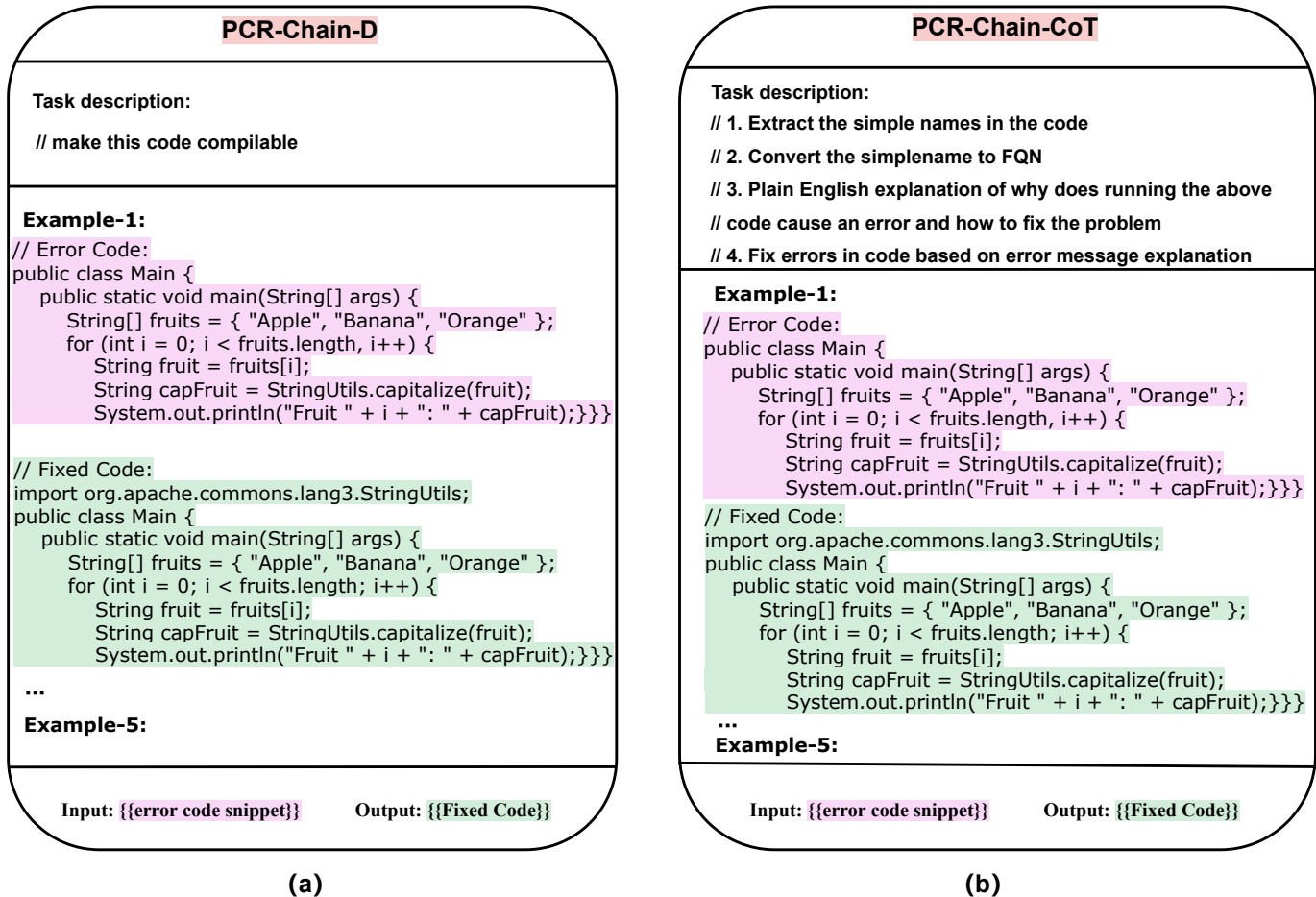


Fig. 4: Consult LLM directly (PCR-D) and consult LLM based on CoT (PCR-CoT)

For Python, we compare PCR-Chain with BIFI [39], an LLM-based SOTA method, and RING [40], an ICL-based SOTA method. In Java, we compare PCR-Chain with CURE, an LLM-based SOTA method, and RING. We modified the examples in RING's prompt to be compatible with Java, making it possible to apply it to Java code.

To conduct the evaluation, we obtain the code for BIFI² and CURE³ from their respective GitHub repositories, as they are both state-of-the-art methods for reusing partial code in Python and Java, respectively. BIFI is implemented using an LLM-based approach in Python, while CURE is implemented in Java using the same approach. However, RING does not release its code, so we reproduce it as accurately as possible.

To gain a better understanding of the mechanism behind the PCR-Chain, we conduct an ablation study by designing three variants:

- PCR-D (see Fig. 4(a)), which directly calls the LLM to generate the PCR of the Java code.
- PCR-CoT (see Fig. 4(b)), a single-prompting approach that describes all steps in one chunk of prompt text and completes a single generative pass.
- PCR-Chain_{w/oEME}, a multiple-prompting approach that does not enhance the error message from compiler, that is

2. <https://github.com/michiyasunaga/BIFI>

3. <https://github.com/lin-tan/CURE>

without error message enhance (EME).

We conduct several experiments to evaluate different aspects of our approach. First, we compare the effectiveness of two different designs: PCR-D and PCR-CoT. Next, we compare PCR-CoT with PCR-Chain to evaluate the effectiveness of our AI chain design. Finally, we evaluate the effectiveness of our error message enhancement (EME) strategy by comparing PCR-Chain_{w/oEME} with PCR-Chain.

3.4 Evaluation Metrics

In our study, we use different metrics to evaluate the effectiveness of our approach for RQs.

For RQ1, we measure the quality of units and modules using the accuracy metric.

For RQ2, RQ3, and RQ4, we use three metrics to evaluate the effectiveness of our approach. These metrics are:

- Number of resolved non-FQNs: This metric counts the code snippets that no longer contain non-FQNs after applying our approach.
- Number of resolved last-mile syntax errors: This metric counts the code snippets that no longer contain last-mile syntax errors after applying our approach.
- Total number of all resolved: This metric counts all the resolved non-FQNs and last-mile syntax errors combined.

4 EXPERIMENTAL RESULTS

In this section, we explore four research questions to evaluate and discuss the performance of our method.

4.1 RQ1: What is the quality of each unit or module in PCR-Chain?

4.1.1 Motivation

The CoT approach is a widely adopted method that advocates for breaking down complex tasks into smaller and more manageable steps. However, relying solely on a single “epic” prompt in CoT-based methods can limit their effectiveness and lead to errors accumulating. To address this limitation, we develop an AI chain that consists of explicit sub-steps, with each step corresponding to a separate AI unit or non-AI unit. In this RQ, we investigate whether each AI unit and module in our approach can effectively ensure the accuracy of partial code reuse.

4.1.2 Methodology

To evaluate the effectiveness of each AI unit and module in PCR-Chain on Java dataset, we first need to obtain ground-truth data. To do so, we enlist the help of two computer science postgraduate students to manually fix unresolved symbols in the Java dataset based on information present in the posts, such as mentioned APIs and API links. One annotator perform the fixing, while the other validate the fixed code. Any disagreements are discussed and resolved.

Once we had the FQNs for non-FQN types in the Java dataset, we could evaluate the *Simplename Extraction* unit and *Simplename to FQN* unit by comparing their outputs with the ground-truth. We consider an exact match to be correct. For the *Error Message Enhance* unit, we do not evaluate the accuracy, as our approach allows this unit to be noisy. As long as some relevant and correct error message explanation is inferred, the subsequent *Code Fix* unit could make repairs.

Regarding the *Code Fix* unit, we consider an output as correct if the code snippet is free of any last-mile syntax errors. The results are presented in Table 1, and more detailed information can be found in Section 3.4.

4.1.3 Result Analysis

The second column of Table 1 presents the accuracy of each unit. Specifically, the *Simplename Extraction* unit of the **FQN Inference Module** correctly predicts 95.7% (815 out of 851) of the simple names in the 200 code snippets, while the *Simplename to FQN* unit infers 92.1% (781 out of 851) of the corresponding FQNs. In contrast, for the **Syntax Error Fix Module**, the *Code Fix* unit accurately fixes 98% (196 out of 200) of the code snippets that contain last-mile syntax errors.

The accuracy of each module is shown in the fourth column of Table 1. For the **FQN Inference** module, it takes in 200 code snippets and outputs 161 code snippets without any non-FQNs, resulting in an accuracy of 80.5%. On the other hand, for the **Syntax Error Fix** module, it takes in 200 code snippets and outputs 196 code snippets that are free of last-mile syntax errors, achieving an accuracy of 98%.

TABLE 1: The Quality of AI Units and Modules

AI unit	Acc	Module	Acc
Simplename Extraction	0.957	FQN Inference	0.805
Simplename to FQN	0.921		
Code Fix	0.980	Syntax Error	0.980

The high accuracy of the units confirms the usefulness of the prompt design and lays the foundation for high-quality modules. The successful execution of modules shows the value of prompt composition in linking units to achieve superior outcomes for higher-level tasks.

4.2 RQ2: How well does PCR-Chain perform in partial code reuse?

4.2.1 Motivation

Our objective is to evaluate the effectiveness of our method in terms of partial code reuse and compare it with the SOTA methods such as BIFI [39], CURE [27] and RING [40] in both dynamically-typed (Python) and statically-typed (Java) programming languages.

For Python, we will compare our method with two SOTA methods, RING (ICL-based) and BIFI (LLM-based). Similarly, for Java, we will compare our method with two SOTA methods, RING (ICL-based) and CURE (LLM-based).

To evaluate the performance of these methods, we will use three metrics: the number of resolved non-FQNs, the number of resolved last-mile syntax errors, and the total number of all resolved, which refers to all non-FQNs and last-mile syntax errors that are resolved.

4.2.2 Methodology

We evaluate the performance of PCR-Chain, BIFI, and RING on the Python dataset, and PCR-Chain, CURE, and RING on the Java dataset. The results are presented in Table 2 and more detailed information can be found in Section 3.4.

4.2.3 Result Analysis

Our evaluation demonstrates that our method is effective in addressing partial code reuse in both dynamically-typed (Python) and statically-typed (Java) programming languages.

For Python, we fix the last-mile syntax errors included in the Code. As shown in Table 2, our method outperforms all baselines, with an accuracy of 99% in resolving 198 last-mile syntax errors. Notably, our accuracy is about 5% and 13.7% higher than RING and BIFI, respectively. This demonstrates that there is sufficient syntax knowledge in LLM to efficiently address last-mile syntax errors.

For Java, we address both non-FQNs and last-mile syntax errors simultaneously, achieving an accuracy of 80.5%. As shown in Table 2, compared with RING and CURE, our method can effectively solve both non-FQNs and last-mile syntax errors, while RING and CURE only address the latter. This difference in performance can be attributed to the fact that RING and CURE directly ask LLMs to make partial code compilable, which can be a challenging task. However, LLMs may only fix some syntax errors and struggle to infer FQNs for non-FQN types.

Since LLMs are language models, both FQNs and non-FQNs represent correct type names from the point of view of language features. This can lead LLMs to mistakenly regard code with non-FQNs as compilable, even if it contains errors due to non-imported packages. However, in reality, such code may fail to compile despite having no syntax errors, precisely because it includes non-FQNs.

As shown in Fig. 1-A, when directly querying the LLM with the prompt “make this code compilable”, the LLM still responds with uncompileable code. In contrast, our method interacts with LLMs step-by-step using the AI chain, thus allowing us to effectively address both non-FQNs and last-mile syntax errors.

Standing on the shoulder of LLM for partial code reuse, PCR-Chain effectively resolves FQNs and fixes last-mile syntax errors. Each AI unit in the AI Chain follows the principle of single responsibility and can interact with LLMs separately to effectively reuse partial code.

4.3 RQ3: How effective are the AI Chain and error message enhance strategies employed in PCR-Chain?

4.3.1 Motivation

CoT can alleviate the challenge posed by directly relying on LLMs, such as the challenge of LLMs mistakenly regarding code with non-FQNs as compilable. However, CoT-based approaches face difficulties in control and optimization due to the “epic” cues with excessive accountability. To solve this problem, we design an AI chain that facilitates step-by-step interaction with the LLM to effectively tackle both non-FQN and last-mile syntax errors.

In this RQ, we aim to investigate two aspects of our approach. Firstly, we would like to explore whether our AI chain design can effectively interact with LLMs, thus enhancing the robustness of our approach. Secondly, we would like to investigate whether the error message enhance strategy could enhance the effectiveness of our AI chain.

4.3.2 Methodology

We set up three approach variants (PCR-D, PCR-CoT, and PCR-Chain_{w/oEME}). Two scenarios (PCR-D vs. PCR-CoT, PCR-CoT vs. PCR-Chain) are used to evaluate the effectiveness of the AI chain. The last one scenario (PCR-Chain_{w/oEME} vs. PCR-Chain) is used to evaluate the effectiveness of error message enhance strategy. The results are presented in Table 3, and more detailed information can be found in Section 3.4.

4.3.3 Result Analysis

The experimental results are presented in Table 3. For PCR-D, both the number of resolved non-FQNs and resolved last-mile syntax errors are fewer than PCR-CoT. This is because when directly asking LLMs to make code compilable, LLMs may mistakenly regard code with non-FQNs as compilable, even if it has missing imported packages. As shown in Fig. 1-A, when directly querying the LLM with the prompt “make this code compilable”, the LLM still responds with uncompileable code. However, the CoT-based prompt is more informative than PCR-D’s prompt.

PCR-CoT shows fewer number of resolved non-FQNs and resolved last-mile syntax errors compared to PCR-Chain. This highlights the superiority of our AI chain design over CoT’s single-prompting approach, which uses an “epic” prompt with hard-to-control behavior and error accumulation.

In contrast, PCR-Chain breaks down the CoT into separate AI units, allowing step-by-step interaction with LLMs for partial code reuse. The last two rows of the Table 3 demonstrate that the error message enhancement strategy improves PCR-Chain’s effectiveness in reusing partial code.

Compared with directly asking the LLM to achieve code reuse, our AI chain design that interacts with the LLM can effectively improve the response reliability of the LLM. Our error message enhancement strategy can reuse partial code more effectively.

4.4 RQ4: How sensitive are the prompts in PCR-Chain to different forms?

4.4.1 Motivation

To further explore the impact of prompt forms on results, we investigate the task description, demonstration examples, and content representation of the prompt.

4.4.2 Methodology

To investigate the impact of task description, we explore prompts with and without task descriptions. Additionally, we examine the impact of the order of demonstration examples, considering different orders such as order, reverse order, and fixed order based on cosine similarity between the prompt’s input and the examples.

For content representation, we focus on the format of representation, whether it is in natural language or a semi-structured form using specific tags. For example, the prompt can be expressed as either “*Task Description: Extract the simple name from the code.*” in natural language form or as “*<Task Description> Extract the simple name from the code</Task Description>.*” in a semi-structured form. We use the Java dataset in this RQ.

Before conducting our experiments, we define a basic configuration based on our intuition of what options would be most effective for PCR-Chain. The basic configuration includes a task description, example prompts in a fixed order, and the use of natural language form in the prompt. We then create variant configurations by changing one factor at a time (task description, order of demonstration examples, content representation), while keeping the other two factors the same as in the basic configuration.

4.4.3 Result Analysis

The results reveal that including a task description improves the LLM’s understanding of tasks, as there were two fewer effective code reuses observed without a task description compared to with a task description.

The order of demonstration examples significantly impacts the LLM’s effectiveness. Presenting the most similar example first (i.e., with the most similar example farthest from the prompt’s input) resulted in the worst accuracy, with 153 out of 200 codes reused, while presenting the most dissimilar example first (i.e., with the most similar example closest to the prompt’s input) led to the best

TABLE 2: Evaluations Results of Multiple Languages (“-” means this error do not contain in this dataset)

Dataset	Approach	Resolved Non-FQNs	Resolved Last-mile Syntax Errors	All Resolved
Python	RING	-	188	188
	BIFI	-	174	174
	PCR	-	198	198
Java	RING	0	188	0
	CURE	0	137	0
	PCR	161	196	161

TABLE 3: Ablation Results of PCR-Chain Variants (“-” means this error do not contain in this dataset)

Strategies	Dataset	Resolved Non-FQNs	Resolved Last-mile Syntax Errors	All Resolved
PCR-Chain	Java	161	196	161
	Python	-	198	198
PCR-D	Java	101	188	101
	Python	-	188	188
PCR-CoT	Java	141	190	141
	Python	-	190	190
PCR-Chain _{w/oEME}	Java	146	190	146
	Python	-	180	180

TABLE 4: Results of Sensitivity of Prompt (+/-Value Against The Basic Config)

Factor	Variant	Resolved Non-FQNs	Resolved Last-mile Syntax Errors	All Resolved
	Basic Config	161	196	161
Task Description	Not Provided	-2	-3	-2
Order of Demonstration Examples	Similar First	-8	-4	-8
	Dissimilar First	+1	+1	+1
Content Representation	Semi-Structured	-8	-2	-8

accuracy, with 162 out of 200 codes reused. This indicates that the LLM is highly influenced by the example closest to the prompt’s input. The results of content representation experiment shows that using natural language prompts reuses 8 more code snippets than using semi-structured forms. This highlights the importance of natural language over semi-structured forms in promoting effective code reuse. While semi-structured forms may improve prompt organization and clarity, they weaken the LLM’s learning ability. In contrast, natural language prompts facilitate better learning and enhance the LLM’s effectiveness.

Although studies [42]–[44] show LLMs are sensitive to prompt factors, our approach remains overall stable in reusing partial code in face of variant prompt factors. Our intuition of the effectiveness of factor variants largely holds, except for prompt order of demonstration example. This indicates the necessity to combine intuition and empirical evidences in prompt design.

5 DISCUSSION

In this section, we summarize the principles of AI chain and prompt design patterns, and also discuss potential threats to validity.

5.1 Prompt Engineering Principles

Our experiments highlight the importance of improving the reliability of LLM responses through the design of an

informative CoT and the creation of an effective AI chain with multiple single-responsibility, composable steps.

In designing an AI chain, we propose three principles:

- **Hierarchical Task Breakdown:** Breaking down a problem into modules, submodules, and further breaking them down into functional units, facilitating a structured problem-solving approach.
- **Unit Composition:** Connecting functional units in a specific structure enables cohesive functioning of the AI chain and achieves optimal results.
- **Mixing of AI Units and non-AI Units:** Designing logic functional units as non-AI units, and utilizing the LLM for fuzzy logic functional units through prompt design.

We believe that prompt engineering will play a crucial role in the future of problem-solving. The principles offer valuable guidance for designing AI chains and maximizing the potential of LLM-based paradigms for effective problem-solving.

5.2 Threats to Validity

Our method faces both internal and external threats. An internal threat is the potential inconsistency in the manually labeled ground-truth data in the Java dataset. To mitigate this, we employ two annotators to label simultaneously and measure the consistency of the results using the Kappa coefficient. A high Kappa coefficient (all coefficients above 80%) indicates the reliability of the annotation results.

In terms of external threats, our research on partial code reuse has focused on the Java and Python. However, to expand the universality of our approach, we plan to investigate code reuse in niche languages such as smart contract code.

Compared to traditional code reuse methods that require expertise in program analysis and significant engineering and maintenance efforts for different languages and their versions, our method offers greater adaptability. Adapting our approach to other languages mainly involves substituting the language type in the prompt examples.

The emergence of new large-scale LLMs, such as GPT-4 [45], [46], may impact our method. Although we are currently on the waitlist for access to GPT-4, we eagerly anticipate using it in the future to validate the effectiveness and universality of our approach.

6 RELATED WORK

Partial code reuse is a common practice among developers, involving the copying and pasting of code snippets from online resources like Stack Overflow into Integrated Development Environments (IDEs). However, unresolved type and last-mile syntax errors often hinder the compilation of partial code. To address these issues, several related works have been conducted in recent years.

TABLE 5: Method Comparison

	Non-FQN	Last-mile Syntax Error	Approach-Based
SNR [14]	✓	✗	Symbolic-Based
Sumit [19]	✗	✓	Symbolic-Based
Huang [8]	✓	✗	LLM-Based
CURE [27]	✗	✓	LLM-Based
RING [40]	✗	✓	ICL-Based
PCR-Chain	✓	✓	CoT-Based

Table 5 presents a comparison of different methods for resolving non-fully qualified name (non-FQN) errors and fixing last-mile syntax errors.

Symbolic-based methods like SNR [14] and Sumit [19] are commonly used to resolve non-fully qualified name (non-FQN) errors or fix last-mile syntax errors. However, these methods may encounter out-of-vocabulary (OOV) failures and require significant engineering effort to develop and program analysis/fix experience.

Built on source code naturalness, recent approaches have overcome these by training or fine-tuning a LLM. For example, Huang et al. [8] utilize the prompt-tuned LLM to resolve types, and CURE [27] use a LLM fine-tuned by an automated program repair task to fix syntax errors. However, LLM-based methods typically require a large amount of data and computing resources to train efficient models. Instead, Brown et al. [28] propose a more lightweight approach called in-context learning (ICL).

ICL is a novel paradigm that allows foundation models to adapt to new tasks without extensive training or updates. Instead of relying on gradient updates [33]–[35], ICL utilizes zero- or few-shot prompts for task adaptation. This paradigm has been successfully applied in range of software engineering tasks, such as testing [47], code generation [48],

and GUI automation [49]. RING [40] used an ICL-based method to fix last-mile syntax errors, avoiding the need to train or fine-tune a LLM. However, this approach does not address non-FQN errors.

Our method, PCR-Chain, differs from RING by not directly querying the LLM with simple prompts like “make this code compilable.” Instead, we leverage the idea of CoT to address both non-FQN and last-mile syntax errors. However, existing CoT approaches only provide simple instructions like “let’s do something step by step.” which cannot handle complex tasks. In contrast, our approach is based on an AI chain [36], [50], [51] that interacts with the model to reuse partial code. While the idea of an AI chain has been explored for writing assistants [36], our AI chain involves complex task analysis and data flow for domain-specific partial code reuse.

7 CONCLUSION AND FUTURE WORK

In this paper, we propose a novel approach for partial code reuse that utilizes the vast amount of code knowledge stored in LLMs. Our approach involves a CoT consisting of six steps, including Simplename Extraction, Simplename to FQN, FQN Supplement, Error Judgement, Error Message Enhance, and Code Fix.

To adhere to the single responsibility principle, we decompose the CoT into an AI chain and supplement it with effective prompt instructions. Our approach outperforms traditional LLM-based methods and the original CoT method in terms of code reuse rates. With the lower cost of building LLM-based partial code reuse tools compared to traditional symbolic-based approaches, our method offers a new LLM-based alternative for software engineering tool development.

Our approach provides a practical and cost-effective solution for software engineering tools, reducing the need for extensive engineering and maintenance work. We also introduce practical principles for using just-in-time engineering and AI chains in SE tasks, demonstrating the potential of LLM4SE. By leveraging the underlying model, we can focus on identifying the problems that AI needs to solve, rather than investing in data collection, labeling, model training, or program analysis. Our code and data package can be found here.⁴

8 ACKNOWLEDGEMENTS

The work is supported by National Nature Science Foundation of China (Nos. 62262031), and Science and Technology Key Project of Education Department of Jiangxi Province GJJ2200302, GJJ2200303, GJJ2200304, GJJ210307).

REFERENCES

- [1] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 513–522, 2010. 1

4. <https://github.com/SE-qinghuang/A-Chain-of-AI-based-Solutions-for-Resolving-FQNs-and-Fixing-Syntax-Errors-in-Partial-Code>

- [2] Margaret-Anne Storey, Leif Singer, Brendan Cleary, Fernando Figueira Filho, and Alexey Zagalsky. The (r) evolution of social media in software engineering. *Future of software engineering proceedings*, pages 100–116, 2014. 1
- [3] Ke Mao, Licia Capra, Mark Harman, and Yue Jia. A survey of the use of crowdsourcing in software engineering. *Journal of Systems and Software*, 126:57–84, 2017. 1
- [4] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Mining stackoverflow to turn the ide into a self-confident programming prompter. In *Proceedings of the 11th working conference on mining software repositories*, pages 102–111, 2014. 1
- [5] Le An, Ons Mlouki, Foutse Khomh, and Giuliano Antoniol. Stack overflow: A code laundering platform? In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 283–293. IEEE, 2017. 1
- [6] Di Yang, Pedro Martins, Vaibhav Saini, and Cristina Lopes. Stack overflow in github: any snippets there? In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 280–290. IEEE, 2017. 1
- [7] Valerio Terragni, Yepang Liu, and Shing-Chi Cheung. Csnippex: automated synthesis of compilable code snippets from q&a sites. In *Proceedings of the 25th international symposium on software testing and analysis*, pages 118–129, 2016. 1
- [8] Qing Huang, Zhiqiang Yuan, Zhenchang Xing, Xiwei Xu, Liming Zhu, and Qinghua Lu. Prompt-tuned code language model as a neural knowledge base for type inference in statically-typed partial code. *arXiv preprint arXiv:2208.05361*, 2022. 1, 5, 6
- [9] Hong Jin Kang, Ferdian Thung, Julia Lawall, Gilles Muller, Lingxiao Jiang, and David Lo. Semantic patches for java program transformation (experience report). In *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019. 1
- [10] CM Khaled Saifullah, Muhammad Asaduzzaman, and Chanchal K Roy. Learning from examples to find fully qualified names of api elements in code snippets. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 243–254. IEEE, 2019. 1
- [11] Qing Huang, Dianshu Liao, Zhenchang Xing, Zhiqiang Yuan, Qinghua Lu, Xiwei Xu, and Jiaxing Lu. Se factual knowledge in frozen giant code model: A study on fqcn and its retrieval, 2022. 1, 2, 3
- [12] Qing Huang, Zhiqiang Yuan, Zhenchang Xing, Zhengkang Zuo, Changjing Wang, and Xin Xia. 1+1>2: Programming know-what and know-how knowledge fusion, semantic enrichment and coherent application. *ArXiv*, abs/2207.05560, 2022. 1
- [13] Rohan Bavishi, Harshit Joshi, José Cambroner, Anna Fariha, Sumit Gulwani, Vu Le, Ivan Radiček, and Ashish Tiwari. Neurosymbolic repair for low-code formula languages. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):1093–1122, 2022. 1
- [14] Yiwen Dong, Tianxiao Gu, Yongqiang Tian, and Chengnian Sun. Snr: constraint-based type inference for incomplete java code snippets. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1982–1993, 2022. 1, 5, 6
- [15] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. Live api documentation. In *Proceedings of the 36th international conference on software engineering*, pages 643–652, 2014. 1
- [16] Vidroha Debroy and W Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 65–74. IEEE, 2010. 1
- [17] Reudismam Rolim, Gustavo Soares, Loris D’Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 404–415. IEEE, 2017. 1
- [18] Ke Wang, Rishabh Singh, and Zhendong Su. Search, align, and repair: data-driven feedback generation for introductory programming exercises. In *Proceedings of the 39th ACM SIGPLAN conference on programming language design and implementation*, pages 481–495, 2018. 1
- [19] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. Automated clustering and program repair for introductory programming assignments. *ACM SIGPLAN Notices*, 53(4):465–480, 2018. 1, 5, 6
- [20] Premkumar T. Devanbu. On the naturalness of software. *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847, 2012. 1
- [21] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51:1–37, 2018. 1
- [22] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the “naturalness” of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 428–439, 2016. 1
- [23] Ahmed Khanfir, Matthieu Jimenez, Mike Papadakis, and Yves Le Traon. Codebert-nt: code naturalness via codebert. *arXiv preprint arXiv:2208.06042*, 2022. 1
- [24] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. *ArXiv*, abs/2002.08155, 2020. 1
- [25] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021. 1
- [26] “github copilot. your ai pair programmer” [online]. available: <https://copilot.github.com/>. 1
- [27] Nan Jiang, Thibaud Lutellier, and Lin Tan. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1161–1173. IEEE, 2021. 1, 4.2.1, 5, 6
- [28] Tom Brown, Benjamin Mann, Amanda Ryder, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020. 1, 6
- [29] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. What makes good in-context examples for gpt-3? *arXiv preprint arXiv:2101.06804*, 2021. 1
- [30] Sewon Min, Mike Lewis, Luke Zettlemoyer, and Hannaneh Hajishirzi. Metaicl: Learning to learn in context. *arXiv preprint arXiv:2110.15943*, 2021. 1
- [31] Stephanie CY Chan, Adam Santoro, Andrew K Lampinen, Jane X Wang, Aaditya Singh, Pierre H Richemond, Jay McClelland, and Felix Hill. Data distributional properties drive emergent in-context learning in transformers. *arXiv preprint arXiv:2205.05055*, 2022. 1
- [32] Sewon Min, Xinxu Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. Rethinking the role of demonstrations: What makes in-context learning work? *arXiv preprint arXiv:2202.12837*, 2022. 1
- [33] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J Liu, et al. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21(140):1–67, 2020. 1, 6
- [34] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019. 1, 6
- [35] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, T. J. Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeff Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *ArXiv*, abs/2005.14165, 2020. 1, 6
- [36] Tongshuang Wu, Michael Terry, and Carrie Jun Cai. Ai chains: Transparent and controllable human-ai interaction by chaining large language model prompts. In *CHI Conference on Human Factors in Computing Systems*, pages 1–22, 2022. 1, 6
- [37] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, 2023. 1
- [38] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022. 1
- [39] Michihiro Yasunaga and Percy Liang. Break-it-fix-it: Unsupervised learning for program repair. In *International Conference on Machine Learning*, pages 11941–11952. PMLR, 2021. 1, 3.2, 3.3, 4.2.1

- [40] Harshit Joshi, José Cambronero, Sumit Gulwani, Vu Le, Ivan Radicek, and Gust Verbruggen. Repair is nearly generation: Multilingual program repair with llms. *arXiv preprint arXiv:2208.11640*, 2022. 1, 3.3, 4.2.1, 5, 6
- [41] Sewon Min, Xinxu Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. Rethinking the role of demonstrations: What makes in-context learning work? In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 11048–11064, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. 2.3
- [42] Tianyu Gao, Adam Fisch, and Danqi Chen. Making pre-trained language models better few-shot learners. *arXiv preprint arXiv:2012.15723*, 2020. 4.4.3
- [43] Ning Ding, Yulin Chen, Xu Han, Guangwei Xu, Pengjun Xie, Hai-Tao Zheng, Zhiyuan Liu, Juanzi Li, and Hong-Gee Kim. Prompt-learning for fine-grained entity typing. *arXiv preprint arXiv:2108.10604*, 2021. 4.4.3
- [44] Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. Calibrate before use: Improving few-shot performance of language models. In *International Conference on Machine Learning*, pages 12697–12706. PMLR, 2021. 4.4.3
- [45] Harsha Nori, Nicholas King, Scott Mayer McKinney, Dean Carignan, and Eric Horvitz. Capabilities of gpt-4 on medical challenge problems. *arXiv preprint arXiv:2303.13375*, 2023. 5.2
- [46] Qing Lyu, Josh Tan, Mike E Zpadka, Janardhana Ponnatapuram, Chuang Niu, Ge Wang, and Christopher T Whitlow. Translating radiology reports into plain language using chatgpt and gpt-4 with prompt learning: Promising results, limitations, and potential. *arXiv preprint arXiv:2303.09038*, 2023. 5.2
- [47] Bei Chen, Fengji Zhang, A. Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. *ICLR*, 2023. 6
- [48] Antonio Mastropaolo, Luca Pascarella, Emanuela Guglielmi, Matteo Ciniselli, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota. On the robustness of code generation techniques: An empirical study on github copilot. 2023. 6
- [49] Zhe Liu, Chunyang Chen, Junjie Wang, Xing Che, Yuekai Huang, Jun Hu, and Qing Wang. Fill in the blank: Context-aware automated text input generation for mobile gui testing. *ArXiv*, abs/2212.04732, 2022. 6
- [50] Tongshuang Wu, Ellen Jiang, Aaron Donsbach, Jeff Gray, Alejandra Molina, Michael Terry, and Carrie J Cai. Promptchainer: Chaining large language model prompts through visual programming. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts*, pages 1–10, 2022. 6
- [51] Hai Dang, Lukas Mecke, Florian Lehmann, Sven Goller, and Daniel Buschek. How to prompt? opportunities and challenges of zero-and few-shot learning for human-ai interaction in creative applications of generative models. *arXiv preprint arXiv:2209.01390*, 2022. 6



QING HUANG is an Associate Professor in the School of Computer and Information Engineering, Jiangxi Normal University. His research interests are software engineering and knowledge graph.



Jiahui Zhu is a first-year graduate student in the School of Computer and Information Engineering, Jiangxi Normal University. His research interests are software engineering and program repair.



Zhenchang Xing is an Associate Professor in the Research School of Computer Science, Australian National University. His research areas are software engineering, applied data analytics, and human-computer interaction.



Huan Jin is currently an associate professor at Jiangxi University of Technology's School of Information Engineering. Her research interests are in data mining and service-oriented software engineering.



Changjing Wang is a Professor in the School of Computer and Information Engineering, Jiangxi Normal University, China. His research interests are Web service and formal method.



Xiwei Xu is a Senior Research Scientist with Architecture & Analytics Platforms Team, Data61, CSIRO. She is also a Conjoint Lecturer with UNSW. She started working on blockchain since 2015. Her main research interest is software architecture. She also does research in the areas of service computing, business process, and cloud computing and dependability.