

Master's Programme in Computer, Communication and Information Sciences

ComplementaryRepair: Enhancing Small Open-Source Large Language Models for Repairing Introductory Student Code with Prompt Diversity

Jinglin Yang

© 2025

This work is licensed under a [Creative Commons](https://creativecommons.org/licenses/by-nc-sa/4.0/) “Attribution-NonCommercial-ShareAlike 4.0 International” license.



Author Jinglin Yang

Title ComplementaryRepair: Enhancing Small Open-Source Large Language Models for Repairing Introductory Student Code with Prompt Diversity

Degree programme Computer, Communication and Information Sciences

Major Computer Science

Supervisor and advisor Senior University Lecturer Arto Hellas

Date 28 April 2025

Number of pages 54

Language English

Abstract

Automated Program Repair (APR) could enhance introductory programming education by repairing errors in student code efficiently. Beyond simply providing solutions, APR can offer partial repairs as hints, aid instructors in identifying errors, and serve as a basis for generating automated feedback. Moreover, repaired student code can offer a more personalized and effective reference for post-submission learning. Recent advancements in Large Language Models (LLMs) have demonstrated impressive capabilities in various tasks, including code repair. However, the effectiveness of LLM-based repair is highly dependent on the design of prompts, which can significantly influence the quality of the generated solutions. Prior works have generally focused on improving prompt design, such as selecting and incorporating structurally similar reference code into the prompt, or application of advanced prompting strategies such as Chain-of-Thought (CoT) to increase repair accuracy. However, few studies have explored how to leverage the variability in prompt design to enhance overall repair accuracy for buggy codes.

To address this limitation, we propose ComplementaryRepair, a conversation-based APR framework that leverages diverse prompts and open-source LLMs trained on code, such as Deepseek-Coder, to improve repair accuracy and computational efficiency. We demonstrate that combining different components (e.g., error messages, test case results, reference code) into the prompt triggers divergent repairing behaviors in LLMs, leading to complementary repair. That is, for a given set of buggy code submissions, one prompt configuration may be more likely to generate a correct repair for a subset of the errors, while a different prompt configuration may be more effective for another subset.

We evaluate the performance of ComplementaryRepair on two datasets from the National University of Singapore and the University of Dublin. Results indicate that, with our settings, ComplementaryRepair successfully repairs up to 98% of buggy submissions in the Singapore dataset and up to 95% in the Dublin dataset, requiring at most 18 repair attempts per buggy code. These findings demonstrate the potential of prompt diversity in optimizing APR for introductory programming assignments.

Keywords Large Language Models, Prompt Engineering, Automated Program Repair, Programming Assignment, Computer science education, LLM Quantization

Preface

I want to express my heartfelt gratitude to my supervisor, Arto Hellas, for his guidance and supervision throughout this thesis. His advice, inspiration, and positive affirmation during our weekly meetings helped me navigate challenges more effectively, deepened my understanding of the subject. This project could not have been smoothly completed without his unwavering support and patience.

Otaniemi, 9 February 2023

Jinglin Yang

Contents

Abstract	3
Preface	4
Contents	5
Symbols and abbreviations	7
1 Introduction	8
1.1 Context	8
1.2 Problem Statement	9
1.3 Proposed Approach: ComplementaryRepair	10
1.4 Research Questions and Scope	10
1.5 Structure of The Thesis	11
2 Background	13
2.1 Automated Program Repair	13
2.2 Large Language Models	13
2.3 Prompt Engineering	15
2.4 LLM Quantization	16
2.5 LLM-based Automated Program Repair	17
2.6 Evaluation Metric	19
3 Complementary Approaches and Divergence	21
3.1 Motivation Example: Complementary Repairs	21
3.2 Complementary Prompts	21
3.3 Enhancing Divergence via CoT	24
3.4 Adaptive Repair Strategies	25
4 ComplementaryRepair Framework	27
4.1 Basic Prompt Repair Phase	28
4.2 CoT Prompt Repair Phase	30
4.3 Datasets	31
4.4 Model and Quantization	32
4.4.1 Deepseek Coder	32
4.4.2 Qwen2.5 Coder	33
4.5 Testing Environment	33
5 Evaluation and Results	35
5.1 RQ1: Effectiveness of ComplementaryRepair	35
5.1.1 Results on Singapore	35
5.1.2 Results on Dublin	37
5.2 RQ2: Impact of Prompt Diversity and Pipeline Design	39
5.3 RQ3: Model Size, Quantization, and Performance	41

6	Discussion	44
6.1	Summarizing Research Questions and Answers	44
6.2	Complementary Repair and other APR approaches	44
6.3	Program Repairs and Similarity	45
6.4	Threats to Validity	46
7	Conclusion	48

Symbols and abbreviations

Abbreviations

<i>APR</i>	Automated Program Repair
<i>LLMs</i>	Large Language Models
<i>CoT</i>	Chain of Thought
<i>TD</i>	Task Description
<i>C_b</i>	Buggy Code
<i>EM</i>	Error Messages
<i>TCR</i>	Test Case Results
<i>C_r</i>	Reference Code

1 Introduction

1.1 Context

In introductory programming courses, students are frequently required to write many small programs to solve well-defined problems. As novice programmers, they often make minor errors and would greatly benefit from timely feedback to address these mistakes. Providing students with precise information about the location of bugs and minimal modifications to resolve them can be advantageous for both learners and instructors. Students can learn from their errors, while instructors can offer detailed explanations about why specific fixes are appropriate and how they relate to fundamental programming concepts. However, the high volume of code submissions in such courses makes manual grading and feedback delivery impractical [30]. To address this challenge, many large-scale programming courses have adopted Automated Program Repair (APR) techniques, which provide hints or corrections for faulty code submissions [10]. The typical APR process involves three core stages: identifying the location of the error (error localization), generating a potential correction (patch generation), and verifying the correctness of the proposed fix (patch validation) [52]. The primary objective of APR is to transform defective code into a functional state that meets predefined requirements, such as passing test cases, while minimizing changes to the original code.

A common concern is that providing direct solutions might hinder students' learning by preventing them from figuring out the correct solution independently. However, APR could not only serve as direct feedback but rather as an intermediate step towards the final feedback. For example, APR can be tailored to offer partial repairs, serving as "next-step hints" to enhance the tutoring effect [49]. One potential application of APR is assisting instructors in efficiently locating errors and debugging student submissions. By generating a correct version of the buggy code, APR can provide instructors with insights into the underlying errors, enabling them to deliver more precise and targeted feedback in natural language. Alternatively, APR-generated repairs can be leveraged to support automated feedback generation. Instead of presenting students with a complete solution, the repaired code could be used as a reference to guide Large Language Models (LLMs) in producing hints related to the corresponding errors in the buggy code. Finally, after students submit their final code, they often require reference solutions to reflect on their mistakes and improve their understanding. While standard solutions serve this purpose, a repaired version of their own code which closely resembles their submitted code could improve learning experience as it directly highlights the necessary modifications.

Despite their utility, many traditional APR methods are search-based, relying heavily on a comprehensive search space derived from historical code data. These methods often struggle with novel programming tasks that lack sufficient prior examples, limiting their applicability.

Recent advances in LLMs have demonstrated significant potential in program repair, offering a promising alternative that reduces reliance on historical data. Some studies frame APR as a text-to-text prediction task, where the input includes faulty

code and supplementary debugging information, and the output is the corrected code. For example, Berabi et al. approach bug repair as a text-to-text transformation task, fine-tuning the Text-to-Text Transfer Transformer (T5) model on a large dataset of bug fixes [4]. However, such models still require substantial volumes of high-quality training data. To mitigate this limitation, zero-shot or few-shot learning settings have been explored. AlphaRepair, for example, uses CodeBERT in a zero-shot setting to predict masked tokens in buggy code [46], but encoder-only models like CodeBERT cannot generate text directly. Current research efforts are increasingly focused on decoder-only LLMs due to their remarkable performance and commercial potential across various domains. These models have demonstrated robust capabilities in comprehending and generating code across multiple programming languages [1], making them effective for APR tasks. For example, Kolak et al. employ Codex, a decoder-only model analogous to GPT-3, with prompt engineering strategies to generate corrected functions from defective code inputs [23], highlighting the strong correlation between model size and patch prediction accuracy.

1.2 Problem Statement

Despite these advances, LLM-based APR pipelines face limitations. First, many studies overlook computational resource and course constraints, which is critical for educational scenarios. For example, many LLM-based APR relying on evaluations using advanced, proprietary LLMs such as GPT-4 [18, 48, 38], and Codex [51]. While these models deliver impressive repair capabilities, their high API cost and potential data privacy issues pose barriers to practical deployment and implementation in education settings. In contrast, the emergence of open-source models, such as DeepSeek-Coder family [13], provides competitive options that support local deployment and customization, delivering cost-efficient, secure, and adaptable solutions. Furthermore, some research prioritizes repair accuracy through methods like generating multiple candidate repairs and selecting the one with minimal modifications [51], often neglecting the associated increase in computational overhead. While resource-aware studies often utilize smaller LLMs (<7B parameters) to fit within standard GPU memory [25], the potential of techniques like quantization [9] to enable the use of larger, yet efficient, models in resource-limited environments remains underexplored. Specifically, a comparative analysis of quantized larger models against full-precision smaller models of similar size could offer valuable insights for maximizing available computational resources.

Secondly, a common limitation in current LLM-based APR approaches is the tendency to perform repeated sampling using the same prompt or prompt template. This can lead to the generation of similar repair attempts and repetitive errors, thereby wasting computational resources and limiting the exploration of diverse potential solutions [45, 54]. Moreover, the rich contextual information inherent in programming assignments, such as task descriptions, test cases, and even peer submissions, remains largely underutilized for enhancing the effectiveness of LLM-based repairs.

While numerous studies have explored different prompting strategies to enhance the performance of LLMs in APR, many of these approaches focus on general APR

tasks, such as project-level development scenarios, rather than programming education contexts. These domains diverge significantly in terms of program complexity, the prevalence and nature of errors, and the diversity of input data available to LLMs. For introductory programming assignments, the tasks are typically less complex, the availability of multiple sources of information offers greater opportunities for prompt design, and the consideration for computation cost and data privacy matters. Furthermore, LLMs with varying parameter sizes exhibit differing capabilities in processing and integrating multiple information sources. While certain prompting strategies have proven effective for large-scale models like GPT-4 [1], their efficacy on smaller, quantized, open-source models remains underexplored.

This study aims to address these gaps by proposing and evaluating a cost-efficient LLM-based repair framework specifically tailored for introductory programming assignments. Our approach strategically leverages the diverse information available in this context to provide reliable automated repairs using open-source LLMs. We systematically investigate the impact of prompt design, their combinations and sequencing, as well as LLM size and quantization, on APR performance within this specific educational context. By examining these factors, we seek to optimize prompt engineering techniques to enhance the performance of small- to medium-sized LLMs for APR in programming education, ultimately contributing to more scalable and effective learning support.

1.3 Proposed Approach: ComplementaryRepair

To address these limitations, we propose ComplementaryRepair, a LLM-based APR framework which leverages the complementary characteristics of repairs derived from distinct prompts observed through our experimental findings to achieve a wider exploration of potential solutions. ComplementaryRepair utilizes a two-phase design: (1) a basic prompt phase that integrates available contextual information (e.g., task descriptions and test cases) to generate stable, high-accuracy repairs with efficient token usage, and (2) a zero-shot Chain-of-Thought (CoT) phase that augments the basic prompts with CoT indicators to facilitate more extensive and in-depth exploration of potential solutions.

1.4 Research Questions and Scope

This study focuses on evaluating the performance of ComplementaryRepair in the automated repair of programming assignments. Specifically, we address the following research questions:

- **RQ1: Effectiveness of ComplementaryRepair.** How effective is ComplementaryRepair in repairing introductory programming assignments, considering both repair rate and computational cost? The repair rate is defined as the proportion of buggy code submissions that achieve at least one valid repair before reaching the maximum iteration limit. Computational cost is measured by

the average number of tokens generated (Avg #Tokens) per buggy submission, with lower Avg #Tokens indicating reduced computational time.

- **RQ2: Impact of Prompt Diversity and Pipeline Design.** How do prompt selection and ordering affect the performance of ComplementaryRepair? We investigate the impact of diverse prompting strategies and the sequential ordering of prompts on repair rate and computational cost.
- **RQ3: Model Size, Quantization, and Performance.** How do model size and quantization influence ComplementaryRepair’s performance? Specifically, to what extent can ComplementaryRepair mitigate the performance differences typically observed between smaller and larger LLMs, and how does quantization impact this mitigation?

1.5 Structure of The Thesis

This thesis is organized as follows:

- **Chapter 2** presents a comprehensive review of related research, covering Automated Program Repair (APR), Large Language Models (LLMs), LLM quantization techniques, prompt engineering techniques, the application of LLMs in APR with a focus on programming educational contexts, and the description of metrics we would use for the evaluation and explanation for the following study.
- **Chapter 3** describes the empirical insights derived from preliminary investigations and articulates the rationale how we came up with the ComplementaryRepair framework.
- **Chapter 4** provides a detailed description of the proposed ComplementaryRepair method. It also outlines the experimental settings, including comprehensive information about the datasets used for evaluation, the specific LLM models selected for testing across different parameter sizes and quantization levels, and the computational environment in which the experiments were conducted.
- **Chapter 5** presents a thorough analysis of the results obtained from the experiments. It evaluates the effectiveness of ComplementaryRepair across different configurations and models, providing quantitative and qualitative insights into its performance.
- **Chapter 6** discusses the broader implications of the experimental findings. It summarizes the answers to the research questions posed earlier, compares ComplementaryRepair with other existing LLM-based APR approaches, analyzes the similarity between the automatically repaired code and the original buggy code, and addresses potential threats to the validity of the study’s conclusions.

- **Chapter 7** summarizes the key findings of the thesis and their implications for the application of LLMs in automated program repair for programming education.

2 Background

2.1 Automated Program Repair

Automated Program Repair (APR) aims to automatically correct software bugs without human intervention, ensuring that the repaired program adheres to predefined specifications. The traditional APR pipeline typically comprises three key stages: bug localization, patch generation, and patch validation as illustrated in figure 1. Among the various APR techniques, search-based methods represent a prominent class. These methods rely on symbolic matching with existing code clusters to identify and generate plausible corrective patches. For example, CLARA utilizes a symbolic pairing mechanism to match incorrect student submissions with analogous correct solutions, enabling the identification of errors and the application of minimal repairs [11]. Similarly, TRACER incorporates recurrent neural networks (RNNs) into the repair process to generate fixes that reflect common corrections made by students for similar errors [2].

However, these search-based approaches are heavily dependent on the availability of a comprehensive search space derived from historical code data. They often struggle when applied to novel programming tasks that lack sufficient prior examples. A limited search space reduces the likelihood of containing correct patches, while expanding the search space increases the probability of generating plausible but potentially incorrect patches before identifying the correct one [44]. This trade-off highlights a significant limitation of traditional search-based APR techniques. LLMs could potentially alleviate this issue, as they leverage latent knowledge acquired during pre-training on massive code corpora, enabling them to generate context-aware repairs without dependency on explicit historical patches.

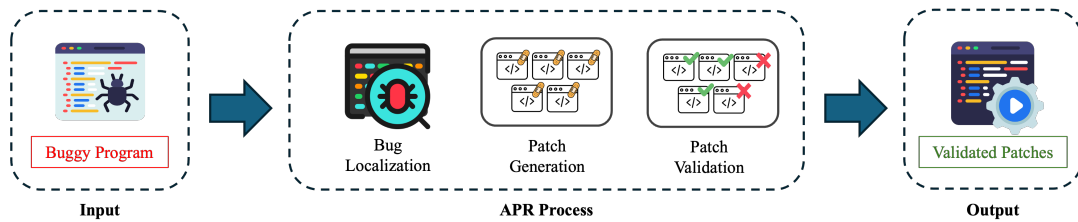


Figure 1: General APR Process: The pipeline begins with bug localization, where techniques are employed to pinpoint the location of the fault(s) within the program code. Following localization, the patch generation stage involves automatically creating potential fixes or patches designed to address the identified bug(s). Finally, patch validation assesses the correctness and effectiveness of the generated patches, often through test cases, to ensure they resolve the bug without introducing new issues.

2.2 Large Language Models

Large Language Models (LLMs) are advanced neural networks built with billions of parameters and undergo extensive training on massive text corpora to perform a diverse

range of language processing tasks. These tasks encompass both natural languages and programming languages, enabling applications such as text generation, translation, summarization, and code analysis. Currently, LLMs are typically based on Transformer architecture. The Transformer architecture leverages attention mechanisms, which allow the model to dynamically weigh the importance of different parts of the input when processing information. This enables the model to effectively capture both short-range and long-range contextual relationships within sequences of tokens [39]. Transformer architecture is composed of two primary components: an encoder and a decoder. The encoder processes input data, transforming it into a high-dimensional vector space that encapsulates the semantic and syntactic features of the input. The decoder, on the other hand, utilizes these representations to produce coherent and contextually relevant output sequences. Depending on their structure, LLMs could be classified into three categories: Encoder-only models, Decoder-only models, and Encoder-decoder models.

Encoder-only models, such as BERT [7], primarily used for tasks requiring comprehension rather than generation, such as text classification and named entity recognition. BERT is trained with Masked Language Modeling (MLM) and Next Sentence Prediction (NSP). In MLM, random tokens in a sentence are masked, and the model learns to predict them based on the surrounding context. NSP, on the other hand, trains the model to determine whether one sentence logically follows another, enabling it to understand relationships between sentences. This dual training approach allows BERT to capture deep contextual representations of text.

Encoder-decoder models such as CodeT5 [33], leverage both the encoder and decoder components of the Transformer architecture. This design makes them particularly effective for sequence-to-sequence transformation tasks, including language translation, text summarization, and question answering. In these models, the encoder processes the input data and generates a vectorized representation, while the decoder uses this representation to produce the corresponding output sequence. This dual functionality enables encoder-decoder models to handle complex tasks that require both understanding and generation of text.

Decoder-only models, such as GPT [1], are a class of autoregressive language models trained to predict the next token in a sequence based on the preceding context. These models leverage the Transformer architecture's decoder component, which employs self-attention mechanisms to capture dependencies between tokens and generate coherent text. The release of ChatGPT in late 2022 demonstrated the advanced capabilities of decoder-only models in generating engaging and contextually relevant text, driving significant attention to this model architecture. This has accelerated the development of more advanced decoder-only models, such as GPT-4 [1], DeepSeek-R1 [12] and more.

Despite their achievements, decoder-only models exhibit inherent limitations. One critical issue is their lack of interoperability, which refers to the difficulty in interpreting the internal decision-making processes that lead to specific outputs [37]. The challenge of interpretability is a common thread across all Transformer-based LLM architectures. While attention mechanisms provide some clues, the distributed nature of knowledge representation in these models makes it difficult to fully understand their

decision-making processes. This “black-box” nature poses challenges for applications requiring transparency and accountability, such as healthcare or legal domains. Another significant challenge is the phenomenon of hallucinations [16], in which the models generate factually incorrect or nonsensical content with unwarranted confidence. This issue stems from their training objective, which prioritizes fluency and coherence over factual accuracy, and can lead to misleading or harmful outputs in real-world applications.

2.3 Prompt Engineering

In the context of LLMs, a prompt is the input text or instructions provided to the model to guide it in generating a specific output. Prompts can take various forms, and they serve as the primary way to communicate the desired task or behavior to the LLM. Prompt engineering encompasses methodologies for crafting task-specific instructions to guide LLMs in generating desired outputs. Effective prompt design can significantly enhance the performance of LLMs for targeted applications without the need for extensive fine-tuning or retraining, making it a cost-efficient approach for task-specific optimization [40]. In this study, we provide an overview of relevant prompt engineering techniques that related to our study (see Figure 2).

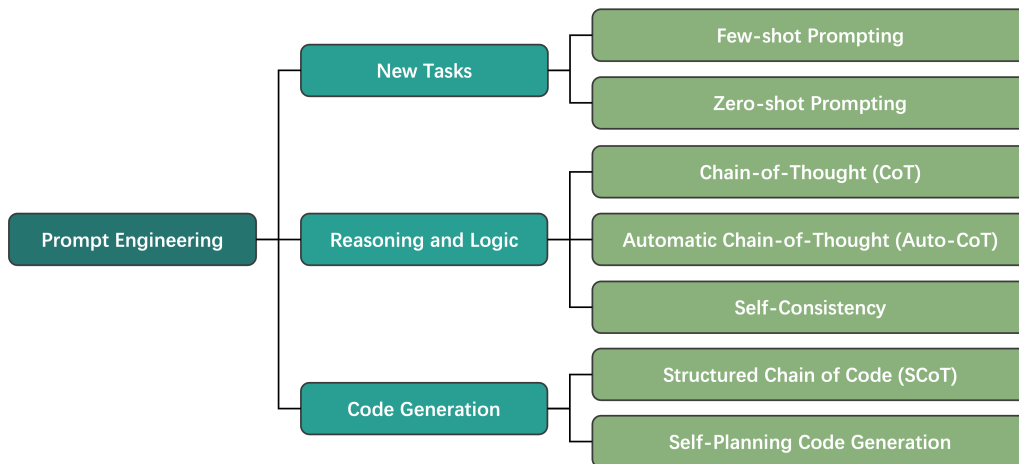


Figure 2: Related prompting engineering techniques in LLMs

For novel tasks lacking extensive training data, zero-shot prompting is commonly employed. In this approach, the model is provided solely with a task description and relies on its pre-trained knowledge to infer and execute the required operations. In contrast, few-shot prompting enhances the task description by including a small number of example input-output pairs. Research has shown that providing these examples can substantially improve LLM performance, especially for more complex tasks that require understanding subtle patterns or relationships [5]. The examples act as a kind of demonstration that helps the LLM to better grasp the desired behavior.

Prompt engineering has been particularly effective in improving the reasoning abilities of LLMs. A key technique in this area is Chain-of-Thought (CoT) prompting.

CoT prompting encourages the LLM to generate a series of intermediate reasoning steps, leading to a final solution [43]. This approach mimics the human problem-solving process by decomposing complex tasks into intermediate steps, which enhances the model’s ability to handle problems requiring extended reasoning. This structured reasoning approach also improves the interpretability of the model’s decision-making process and aids users in diagnosing potential errors.

However, the manual creation of effective CoT prompts can be time-consuming and require significant effort. To address this, zero-shot CoT [22] which simply adding the CoT indicator “Let’s think step by step” could produce impressive improvement in various natural language tasks for substantially large LLMs. It’s important to note, however, that early research indicated CoT reasoning was not consistently effective for smaller LLMs. Building on the findings of zero-shot CoT, Automatic Chain-of-Thought (Auto-CoT) leverages this technique to automate the generation of reasoning chains [53]. Auto-CoT further enhances robustness by clustering questions and sampling diverse examples from multiple clusters, ensuring a broad representation of reasoning patterns. In contrast to CoT, which follows a single reasoning path, self-consistency methods sample multiple reasoning paths from the LLM, potentially yielding different answers, and select the most consistent one as the final output [42]. The intuition behind self-consistency is that the correct answer is more likely to be consistently produced across different reasoning paths, while incorrect answers may vary more randomly.

Although CoT has demonstrated effectiveness in reasoning tasks, its impact on improving code generation performance has been relatively modest [6]. To bridge this gap, Structured Chain-of-Thought (SCoT) incorporates fundamental programming constructs such as branches, loops, and sequences into the reasoning process, significantly improving LLM performance in code generation tasks [27]. SCoT first instructs the LLM to generate a rough structured solving process for the code utilizing the three fundamental code structures and then uses this plan to produce the final code. Consistent with the principles of CoT, programming tasks can be decomposed into multiple subtasks to enhance LLM capabilities. Self-planning code generation further extends this idea by first prompting the LLM to generate a detailed implementation plan based on the task description, followed by utilizing both the task description and the structured plan to generate the final code [20]. The primary distinction between Structured Chain-of-Thought (SCoT) and self-planning code generation lies in the format of the generated implementation plan. SCoT produces plans in a pseudocode-like textual format, explicitly incorporating programming structures, while self-planning code generation formulates plans using natural language descriptions, which are more aligned with human-readable explanations.

2.4 LLM Quantization

LLMs require substantial computational power and memory due to their large number of parameters. This poses significant challenges for deployment in resource-constrained environments, such as educational settings with limited hardware. For instance, a 14 billion parameter model, with parameters typically stored as 16-bit floating-point

values (two bytes per parameter), requires at least 28GB of memory. This often exceeds the VRAM capacity of many mid-range GPUs (e.g., 24GB or less). Such constraints hinder the practical application of larger LLMs in various contexts.

To address the constraints of limited VRAM, one strategy is to offload part of LLM weights to alternative storage, such as system RAM or hard drives. This approach enables the use of larger models on hardware with inadequate GPU memory. However, it introduces considerable drawbacks, including increased latency and reduced processing speed, since RAM and disk access are far slower than VRAM. In educational scenarios, where high throughput is critical to provide timely feedback to many students, this method is less attractive, as delays could undermine the learning experience.

Quantization techniques, on the other hand, address these challenges by reducing the memory footprint of LLMs while preserving acceptable performance levels [9]. Quantization involves representing model weights or activations in lower-precision formats, such as 4-bit integers, instead of high-precision 16-bit or 32-bit floating-point formats. This can reduce memory requirements by a factor (e.g., from 14GB to approximately 3.5GB for a 7B model with 4-bit quantization), enabling deployment on standard GPUs in resource-limited settings.

Two primary quantization approaches are commonly used: post-training quantization (PTQ) and quantization-aware training (QAT). PTQ applies quantization to a pre-trained model without retraining, making it simpler and faster to implement but potentially leading to slight performance degradation. QAT, conversely, incorporates quantization during training, optimizing the model for lower precision and often yielding better accuracy at the cost of increased training complexity. Research indicates that 4-bit quantization, particularly PTQ, strikes an effective balance, maintaining performance close to full-precision models for many tasks, including code-related applications [21]. However, reducing precision further (e.g., to 3 bits) often results in significant performance drops, limiting its utility for APR [21].

In the context of APR, quantization offers several benefits. First, it enables the use of larger, more capable models on hardware with limited VRAM, broadening access to advanced LLMs in educational environments. Second, quantized models can outperform smaller, non-quantized models with comparable memory footprints [21]. For example, a 4-bit quantized 14B model may deliver superior repair accuracy compared to a full-precision 7B model, as it retains more of the original model's knowledge.

2.5 LLM-based Automated Program Repair

Since the emergence of LLMs, many applications leveraging these models have been integrated into educational contexts, encompassing tasks such as automated feedback generation [8], automated grading [35], exercise creation [29], and test case generation [31], among others. Among these areas, automatic feedback generation has emerged as a particularly active field of research, as feedback plays a critical role in the learning process, benefiting both instructors and students. For novice programmers, directly providing a complete solution is generally discouraged, as

it may impede critical thinking and hinder learning from errors. Instead, offering explanatory guidance and structured support is considered more effective in fostering their development [28]. In this context, Automated Program Repair (APR) serves as a vital intermediary step toward improving feedback quality [25]. By generating fixes that address specific errors made by learners, APR enables the creation of more meaningful and targeted feedback. For instance, performing syntax error repair as an initial step enables LLMs to generate more precise and comprehensible explanations of these errors in natural language [32].

LLMs have demonstrate great promising in enhancing bug fixing tasks, with a significant improvement in performance compared to deep-learning-based approach [19]. To improve LLMs' capability in APR, both fine-tuning-based and prompt-based approaches have been widely explored and evaluated.

Fine-tuning-based approaches treat APR as a downstream task for pretrained LLMs. For instance, AlphaRepair [47] adopts a cloze-style methodology, framing APR as a Masked Language Modeling task, where CodeBERT predicts the masked buggy line based on its surrounding context. Alternatively, APR can be formulated as a text-to-text prediction problem, where the input is buggy code, and the output is the fixed code. For example, TFix [4] fine-tunes the Text-to-Text Transfer Transformer (T5) to predict fix by leveraging information such as error types, error messages, and error contexts obtained from an error detector. Fine-tuning has been shown to significantly enhance model performance, with improvements ranging from 31% to 1,267% across various LLMs [19]. Another advanced frameworks, such as Toggle [14], introduces a repair framework that fine-tunes CodeT5 to predict bug locations, utilizes LLMs with dynamic prompts to generate fixes, and incorporates an adjustment model to refine predicted bug locations, ultimately optimizing bug-fix performance. Instead of generating the fixed code in one step, RePair propose to repair the code in multiple steps, and improve the codes with the continuous feedback. Unlike traditional one-step code correction, RePair [55] proposes a multi-step iterative repair process that continuously improves code quality through feedback. This approach integrates reinforcement learning, where a reward model serves as a critic, providing iterative feedback to guide the LLM's output until convergence or reaching a predefined iteration limit.

Prompt-based approaches directly leverage pretrained LLMs without modifying their parameters, relying on strategic prompt design to enhance APR performance. For example, Wang et al. [41] propose tailored prompting strategies for Python code generation, demonstrating their effectiveness in educational scenarios. However, certain prompting techniques, such as 1-shot learning, coding guideline integration, or step-by-step reasoning, may reduce performance for smaller models like LLaMA (llama3-8b) and Mistral (mixtral-8x7b). Specific to APR, Tian [38] experiments with Codex using four different prompt configurations and observes that including bug descriptions significantly aids LLMs in fixing errors, with fault localization providing further benefits in most cases. Several advanced prompt-based approaches integrate CoT reasoning and few-shot learning to enhance APR. ThinkRepair incorporates the ideas of CoT and few-shot learning into the APR procedure. ThinkRepair [50] follows a two-phase approach: a collection phase, where a pool of verified bug-fix examples with reasoning chains is created, and a fixing phase, where the LLM iteratively

repairs code using the most semantically similar examples until all test cases pass or a maximum iteration limit is reached, optionally incorporating failure information to guide further improvements. Peer-aided Repair [54] builds on a similar concept, leveraging large-scale programming course submissions as a high-quality sample pool. By identifying previously submitted solutions with the highest similarity to the target buggy code, the approach selects relevant examples likely to contain similar errors and corrections. PyDex [51] also utilizes peer programs as a sample pool, addressing both syntax and semantic errors by first iteratively correcting syntax errors before leveraging peer solutions, task descriptions, and test cases to resolve semantic issues. Beyond these methods, test cases serve as a valuable resource for improving APR. ContrastRepair [24] employs a contrastive approach by utilizing pairs of failing and successful test cases with minimal differences to provide informative feedback. Experimental results indicate that such contrastive guidance significantly improves LLMs’ ability to understand and rectify coding errors. Additionally, a novel technique known as Round-Trip Translation (RTT) has shown promising results in APR [36]. This approach exploits the structural regularities present in real-world code corpora used for training LLMs, which are predominantly error-free. It first translates the buggy code into an intermediate representation (either in natural language or another programming language) and subsequently translates it back into the original programming language. Instead of translation, an alternative strategy involves refactoring reference code into various structural variants and incorporating these refactored versions into prompt design. This method has been shown to reduce patch size and enhance bug-fixing performance [18].

2.6 Evaluation Metric

We focus on three categories of evaluation, one to measure the functional correctness of the repairs, one to measure the computational cost for generating a valid repair, and one to measure the modification made to produce the repairs.

Pass@k: is a metric designed to measure the accuracy of generated programs [6]. For each buggy program, we generate n repaired programs, which are subsequently evaluated against a predefined suite of test cases. A repair is considered valid if it successfully passes all test cases. In this way, we would get c plausible correct repaired program. For a given k , all possible combinations of k repairs from the n repairs are generated. We then count the number of combinations where at least one repair successfully passes all unit tests. Dividing this count by the total number of combinations yields the probability that a random sample of k repairs will successfully fix the defective program. The unbiased Pass@k is calculated as:

$$\text{Pass@k} = E \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \quad (1)$$

Repair rate: a buggy code is considered repaired if at least one of the generated repairs passes all test cases. The verifiable nature of most introductory programming assignments makes this metric particularly valuable, as it allows us to automatically

validate the repaired code in a short time. In the evaluation dataset, the repair rate is calculated as the number of buggy code submissions for which at least one successful repair was generated, divided by the total number of buggy code submissions.

Number of tokens: the number of tokens generated by the LLMs. Specifically, we would primarily use the average of tokens generated until a valid fix found. This serves as an indicator of the proposed framework, as the computation cost is closely related to the tokens generated by LLMs, where larger number of tokens implies a higher computation cost as generating each token requires the model to perform a series of complex operations.

Similarity: is calculated based on Levenshtein distance, which measures the number of single character edits: insertions, deletions, or substitutions required to transform one string into another [26]. We use this to assess the similarity between the original buggy program and its repaired version. Our purpose is to repair the buggy code that have similar structure and minimum modification, rather than come up with a complete new program. Therefore, we prefer repaired code with higher similarity scores to the original code. To enable better comparison, the distance is normalized, where for two programs a and b , with character lengths $|a|$ and $|b|$, the similarity is defined as:

$$\text{Similarity}(a,b) = 1 - \frac{\text{Levenshtein Distance}(a, b)}{\max(|a|, |b|)} \quad (2)$$

3 Complementary Approaches and Divergence

3.1 Motivation Example: Complementary Repairs

Consider two buggy codes, C_1 and C_2 , each containing distinct types of errors, and two “Tutors”, T_1 and T_2 , tasked with repairing buggy codes for students. Suppose T_1 can repair C_1 with a 100% success rate per attempt but has a 0% chance of repairing C_2 . Conversely, T_2 can repair C_2 with a 100% success rate per attempt but has a 0% chance of repairing C_1 . If only T_1 is working, the overall repair rate is $(1 + 0) \div 2 = 0.5$. The same applies if only T_2 is working, yielding an identical repair rate of 0.5.

Now, let’s explore what happens if each tutor makes multiple attempts, say 2 attempts per buggy code (e.g., trying different repair strategies). For T_1 , the repair rate remains $(2 + 0) \div 4 = 0.5$, with only C_1 being repaired twice, and similarly for T_2 , only C_2 is repaired, leaving the overall repair rate unchanged at 0.5. However, a more effective approach emerges if we assign each tutor to attempt repairing both codes once, then: T_1 repairs C_1 , and T_2 repairs C_2 . Here, the overall repair rate across the 4 attempts is still $(1 + 1) \div 4 = 0.5$, but now both C_1 and C_2 have one valid repair each. In other words, while the average repair rate doesn’t increase, the outcome improves as both buggy codes are successfully repaired.

This scenario highlights a key insight: leveraging the complementary strengths of T_1 and T_2 can maximize the number of distinct repairs even though the overall repair rate remains unchanged. This approach has practical value in APR for programming education. By combining complementary repair capabilities, we can increase the likelihood of obtaining at least one valid repair for each buggy code. This is critical since repair correctness can be efficiently verified by running the repaired code against a test suite. More broadly, this strategy applies to any domain where solutions can be automatically validated, enhancing the chances of obtaining at least one correct outcome.

3.2 Complementary Prompts

We propose that by feeding the prompts with different templates and information, it can trigger LLMs to make more diverse repair attempts, resulting in complementary repairs. In programming assignment repair, the debugging process tends to present distinct focus depending on the information provided. For example, if an error message explicitly points to a runtime issue at a particular line of code, effort is more likely to focus on that line and make targeted modifications. Similarly, exposure to failing test cases directs attention toward case-specific adjustments, while reference solutions direct comparative analysis between defective and correct implementations. This variation in focus suggests that different information sources can lead to distinct repair strategies, potentially resulting in complementary repairs, where one approach succeeds while another fails across different types of buggy code. Therefore, we hypothesize that tailoring prompts with diverse information will lead LLMs to adopt distinct repair strategies, yielding varied success rates across different buggy codes.

To investigate this, we conduct preliminary experiments that employ four prompt

configurations on a dataset from the National University of Singapore containing five assignments (detailed in Section 4):

- **Prompt 1** ($TD + C_b$): Serves as the baseline, containing only the programming task description (TD) and the buggy code (C_b). The LLM is instructed to act as an APR tool and generate fixes for the erroneous code.
- **Prompt 2** ($TD + C_b + EM + TCR$): Incorporates the prompt 1 with error messages (EM) generated during execution, as well as a list of passing and failing test cases results (TCR).
- **Prompt 3** ($TD + C_{buggy} + C_r$): Includes a reference implementation (C_r) obtained from a peer submission, selected based on similarity scores computed using CodeBLEU [34].
- **Prompt 4** ($TD + C_b + EM + TCR + C_r$): Integrates all elements from the previous three prompts.

Table 1: Overall prompt comparison in fixing Singapore

Model	Size	Prompts	Pass@1	Pass@3	Pass@5	Relative Improvement
ds-coder-instruct	6.7B	$TD + C_b$	58.0	69.0	72.1	24.3
		(+) $EM + TCR$	61.7	71.6	75.4	22.2
		(+) C_r	72.9	84.2	87.0	19.3
		(+) $EM + TCR + C_r$	73.8	87.3	90.9	23.2

Table 2: Results on five programming assignments in Singapore dataset

Model	Size	Prompts	Q1	Q2	Q3	Q4	Q5
			Pass@1	Pass@1	Pass@1	Pass@1	Pass@1
ds-coder-instruct	6.7B	$TD + C_b$	54.4	30.3	83.2	77.3	47.4
		(+) $EM + TCR$	61.6	26.9	91.8	84.2	36.5
		(+) C_r	68.7	72.7	89.1	79.3	35.2
		(+) $EM + TCR + C_r$	66.2	72.8	91.1	84.3	41.3

The results in Tables 1 and 2 reveal several key observations. First, they validate our hypothesis that different types of information produce complementary repairs at least at the assignment level. Specifically, Prompt 1 performs better on assignment 1, Prompt 2 better on assignment 3, Prompt 3 better on assignment 1, and Prompt 4 better on assignment 2 and 4. Figure 3 further illustrates this complementarity at the submission level, where a significant proportion of buggy codes are uniquely resolved by only one of the individual prompts. Notably, simply increasing the amount of information in the prompt does not necessarily enhance repair performance. Instead, excessive or conflicting information can mislead the LLM, reducing overall repair accuracy. For example, Prompt 4, which includes everything from Prompt 1 to Prompt 3, underperforms Prompt 1 on fixing assignment 5, Prompt 2 on assignment 3, and Prompt 3 on assignment 1.

Second, prompts influence how repair accuracy evolves with multiple attempts, measured as $\text{pass}@k$ (the probability of at least one correct repair in k attempts). We define convergence speed as how quickly accuracy improves as k increases. Prompts encouraging broader exploration (divergent behavior) show larger accuracy gains at higher k , while those promoting narrow focus (convergent behavior) converge early. For example, Prompt 1, with a relatively low overall accuracy, gains 24.3% improvement from $\text{pass}@1$ to $\text{pass}@5$, indicating a high exploratory potential. This kind of improvement could be considered as a kind of complementary repairs generated within one single prompt, where randomness helps avoid repeated failed attempts. Notably, the 4 prompts gain substantial increase from $\text{Pass}@1$ to $\text{Pass}@3$, with a relatively marginal gain from $\text{Pass}@3$ to $\text{Pass}@5$, which implies that generating 3 responses for each prompt per buggy code could already explore most of the potentials of one prompt, offering a good balance between repair rate and computation cost.

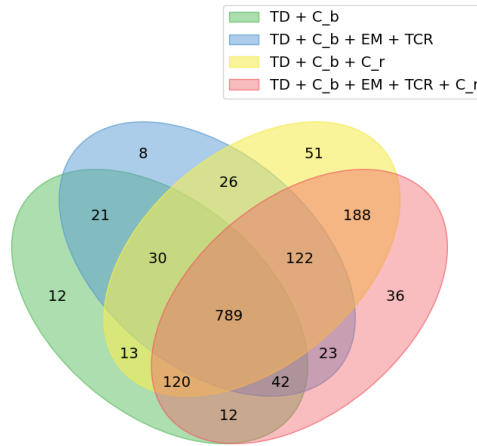


Figure 3: Number of buggy codes repaired by different prompts

To further assess complementarity across prompts, we compared the unbiased $\text{pass}@k$ metrics for different combinations of prompts compared to the best individual prompt in the group as illustrated in Table 3. In our experiment, we first use each prompt to make 5 repair attempts for each buggy code, yielding 20 total repairs across all four prompts. Therefore, for a combination like (p1, p2), we get a pool of 10 repairs (5 from p1, 5 from p2) and compute unbiased $\text{pass}@k$ as the expected probability of at least one correct repair among k randomly selected attempts from this pool. There are several key observations:

First, at $\text{pass}@1$, combinations consistently underperform the best individual prompt in the group due to averaging effects. Since $\text{pass}@1$ for a combination is the mean of the individual prompts' $\text{pass}@1$ values, it is inherently lower than the $\text{pass}@1$ of the highest individual prompt.

Second, combinations of different prompts outperform individual prompts as k increases. For instance, the (p1, p2) combination achieves a $\text{pass}@3$ of 76.1%,

surpassing p1 (69.0%) and p2 (71.6%) individually, with similar observations at pass@5. This trend holds across all combinations, demonstrating that pooling repairs from multiple prompts increases the likelihood of success with more samples, leveraging their complementary strengths.

Third, the performance of combinations with more prompts becomes increasingly dependent on larger k . For example, the (p1, p2, p3) combination gets 85.5% at pass@3, slightly below its subgroup (p2, p3) at 86.2%. However, at pass@10, (p1, p2, p3) significantly outperforms all subgroups, suggesting that broader combinations offer greater benefits as more attempts are sampled.

Table 3: Results on combination of prompts in Singapore dataset

Model	Size	Combination	Prompts	Pass@1	Pass@3	Pass@5	Pass@10	Pass@15
ds-coder-instruct	6.7B	1	p1	58.0	69.0	72.1	-	-
			p2	61.7	71.6	75.4	-	-
			p3	72.9	84.2	87.0	-	-
			p4	73.8	87.3	90.9	-	-
		2	(p1,p2)	59.8	76.1	80.9	85.0	-
			(p1,p3)	65.4	86.2	91.4	94.6	-
			(p1,p4)	65.9	86.6	92.1	95.6	-
			(p2,p3)	67.3	87.3	92.3	95.2	-
			(p2,p4)	67.8	86.8	91.8	95.3	-
			(p3,p4)	73.4	90.2	93.9	96.3	-
		3	(p1,p2,p3)	64.2	85.5	91.6	95.6	96.7
			(p1,p2,p4)	64.5	85.2	91.3	95.8	97.1
			(p1,p3,p4)	68.2	89.4	94.4	97.3	98.2
			(p2,p3,p4)	69.5	89.8	94.5	97.3	98.0

In conclusion, these findings confirm a significant complementary repair effect across different prompts. A weaker effect exists within the same prompt, likely because LLMs tend to repeat similar successes or errors when given the same input, limiting divergence. The key to maximizing complementary repairs lies in enhancing prompt-driven diversity.

3.3 Enhancing Divergence via CoT

In the previous section, we have showed how divergence drives complementary repairs. As demonstrated in Section 3.2, different types of prompt information influence the extent to which divergent solutions are generated. Here, we propose that CoT prompt could also enhance divergence for small LLMs. The theoretical basis for this lies in the structural differences between direct and CoT-based reasoning. In a standard LLM response, the problem-to-solution mapping follows the form $\langle input \rightarrow output \rangle$. By contrast, CoT introduces an intermediate reasoning step, following the mapping structure $\langle input \rightarrow rationale \rightarrow output \rangle$. This additional reasoning layer introduces greater variability, as small change in either $\langle input \rightarrow rationale \rangle$ or $\langle rationale \rightarrow output \rangle$ would lead to significant different outputs. Consequently, CoT can increase the likelihood of generating diverse paths toward the correct fixes. We also conduct preliminary experiments to evaluate this hypothesis. Following the setup described in Section 3.2, we modify the prompts by appending a zero-shot

CoT instruction: "Let's think step by step", a widely adopted practice for enhancing reasoning depth in LLMs [22].

Table 4: Comparison between basic prompts and zero-shot CoT prompts in repairing accuracy

Model	Size	Prompts	Pass@1	Pass@3	Pass@5	Relative Improvement	Avg. #Tokens
ds-coder-instruct	6.7B	$TD + C_b$	58.0	69.0	72.1	24.3	205
		$TD + C_b + CoT$	57.1	75.5	81.0	41.9	335
		$(+)EM + TCR$	61.7	71.6	75.4	22.2	241
		$(+)EM + TCR + CoT$	60.7	78.9	83.6	37.7	342
		$(+)C_r$	72.9	84.2	87.0	19.3	196
		$(+)C_r + CoT$	72.2	91.9	96.4	33.5	350
		$(+)EM + TCR + C_r$	73.8	87.3	90.9	23.2	232
		$(+)EM + TCR + C_r + CoT$	72.2	91.1	95.3	32.0	343

The experimental outcomes, presented in Table 4, reveal distinct behavioral patterns between conventional and Chain-of-Thought (CoT) prompting approaches. The analysis demonstrates that while CoT yields lower mean success rates (pass@1) compared to basic prompts, it exhibits a substantial improvement in pass@5 performance, ultimately achieving superior repair accuracy at higher sampling thresholds. From a statistical standpoint, basic prompts exhibit lower variance and higher initial accuracy, producing more consistent but potentially less diverse solutions, whereas CoT prompts introduce greater variance in solution quality, leading to lower mean accuracy but improved performance over multiple attempts. This divergence suggests that CoT's exploratory nature enhances its capacity to discover correct fixes through broader solution space exploration.

However, the enhanced exploratory capability of CoT comes with significant computational trade-offs. A practical implementation of CoT generally requires extended context windows and generation limits to achieve full effectiveness, which imposes considerable computational overhead. Besides, as demonstrated in Table 4, CoT responses generally contain two to three times more tokens than their non-CoT counterparts, leading to higher computational costs and extended processing times. Furthermore, the effectiveness of CoT prompting depends heavily on the number of generated solution variants, further intensifying computational resource demands. These factors must be carefully considered, particularly in scenarios where computational resources are constrained.

3.4 Adaptive Repair Strategies

The debugging process typically involves generating and validating hypotheses about error causes and their fixes. Beyond identifying these hypotheses, an implicit meta-cognitive evaluation occurs, where the likelihood of each hypothesis being correct should be assessed to determine which to pursue first. A logical approach involves first testing the most promising hypotheses before considering less probable alternatives. If the initial hypotheses fail to resolve the issue, shifting to a more exploratory and divergent problem solving strategy may prove beneficial, rather than persistently

refining unpromising solution paths. However, engaging in divergent exploration too early risks losing focus, increasing the computational effort needed for a solution. The observed repair patterns highlight an inherent trade-off between solution diversity and computational efficiency, which must be carefully optimized in the design of APR systems.

Therefore, in the context of APR for introductory programming assignments, even though the diversity of prompts could produce complementary repairs that increase the repair rate, generating responses from all prompts indiscriminately can be computationally expensive, particularly for resource-constrained environments. To address this, an adaptive strategy that prioritizes prompts based on their expected effectiveness for a given buggy code can optimize both repair success and computational efficiency. Prioritizing prompts expected to succeed early could minimize unnecessary computations while retaining the benefits of prompt diversity.

4 ComplementaryRepair Framework

Based on empirical observations, we introduce ComplementaryRepair, a LLM-based ARP framework designed to leverage complementary repairs by adopting diverse prompts. This approach aims to balance the trade-off between repair accuracy and computational efficiency. ComplementaryRepair operates in two phases: the basic prompt repair phase and the CoT prompt repair phase as illustrated in Figure 4.

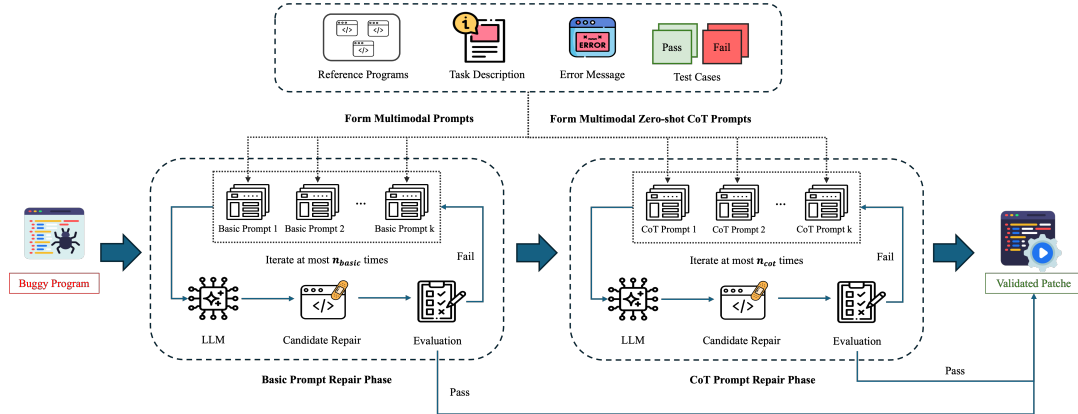


Figure 4: ComplementaryRepair Framework: operates in two phases: a basic prompt repair phase, and a Chain-of-Thought prompt repair phase. In the basic prompt phase, the framework employs diverse prompts containing task descriptions, buggy code, error messages, test case results, and reference code to generate potential fixes. If this phase doesn’t yield a correct repair, the CoT prompt phase is initiated. This phase uses the same prompts but adds “Let’s think step by step” to encourage the LLM to conduct more exploratory attempts.

The key idea of ComplementaryRepair is the use of diverse prompts, rather than the specific content of any single prompt. In the basic prompt repair phase, several basic prompts, each incorporating different combinations of available information, are used iteratively to generate repairs for the buggy code until a valid repair is identified or a maximum number of iterations is reached. If this phase is unsuccessful, the process transitions to the CoT prompt repair phase, which mirrors the basic prompt repair phase, but employs CoT-style prompts.

The prompts employed in ComplementaryRepair incorporate the following components, all of which represent commonly available information in introductory programming assignments:

- **Task Description (TD):** A natural language specification detailing the problem statement and the functional requirements that the submitted code is expected to fulfill.
- **Buggy Code (C_b):** The erroneous code submitted by students.
- **Error Messages (EM):** Diagnostic messages generated when executing the buggy code against test cases, including runtime and compilation errors. We

validate the correctness of the buggy code through test assertion, thus we exclude the detailed assertion errors from the error messages, since they are already captured within the test case results.

- **Test Case Results (TCR):** Information on passing and failing test cases, formatted as: "Pass test n : assert function(input) == output" and "Fail test n : assert function(input) == output"
- **Reference Code (C_r):** The reference code from peer-submitted solutions and reference implementations.

4.1 Basic Prompt Repair Phase

Preliminary experiments indicate that the complementary effects of different basic prompts are already significant. Compared to divergence introduced through CoT prompting, basic prompts offers two key advantages. First, basic prompts converge more rapidly, thus a relatively small number of attempts is sufficient to explore the repair potential. Second, basic prompts generate fewer tokens than CoT-based approaches, introducing lower computational cost and potentially faster repairing. To test the repair effectiveness and efficiency, we employ three prompts previously introduced in Section 3.2 to ComplementaryRepair, and the detailed prompt example is demonstrated in figure 5. The design rationale for these prompts is elaborated below:

- **Prompt 2 ($TD + C_b + EM + TCR$):** Incorporates the problem description, the erroneous submission, execution error messages, and test case results. Assertion errors are omitted since they are inherently captured within test case results, with emphasis placed on runtime and compilation errors.
- **Prompt 3 ($TD + C_b + C_r$):** Includes a reference implementation obtained from a peer submission, selected based on similarity scores computed using CodeBLEU [34]. CodeBLEU incorporates four components: standard BLEU (measuring similarity to human translations), weighted BLEU (assigning higher weights to programming language keywords), an Abstract Syntax Tree (AST) match score, and a semantic data-flow match score. Prior studies on peer-aided repair [54] suggest that reference solutions with high syntactic and semantic similarity have greater reference value.
- **Prompt 4 ($TD + C_b + EM + TCR + C_r$):** Integrates all elements from the previous three prompts.

The prompts are executed sequentially in the order: $\langle Prompt3 \rightarrow Prompt4 \rightarrow Prompt2 \rangle$. After obtaining a response from the LLM, we extract the repaired program from the output, as it often includes both explanatory text and source code. The extracted program is then tested against the full suite of test cases to assess correctness. A repair is considered successful if it passes all test cases, and we exit the iteration to save computational cost. If the repair is unsuccessful, we proceed to the next prompt in the sequence. This process is repeated for up to n_{basic} iterations.

```

1 Your responsibility is to provide a fix for the buggy code.
2
3 ### Problem description:
4 Sequential Search: Implement a function that determines the appropriate insertion index for a given value x in a
5 sequence seq, ensuring that the sequence remains in non-decreasing order.
6
7 ### Buggy code:
8 def search(x, seq):
9     for i in range(len(seq)):
10         if x <= seq[i]:
11             return i
12         return i + 1
13
14 ### Error messages:
15 Runtime Error: Traceback (most recent call last):
16   File line 7, in <module>
17     assert search(42, (-5, 1, 3, 5, 7, 10))==6 and search(42, [1, 5, 10])==3 and search(5, (1, 5, 10))==1 and search(7,
18     [1, 5, 10])==2 and search(3, (1, 5, 10))==1 and search(-5, (1, 5, 10))==0 and search(10, (-5, -1, 3, 5, 7, 10))==5
19     and search(-100, (-5, -1, 3, 5, 7, 10))==0 and search(0, (-5, -1, 3, 5, 7, 10))==2 and search(100, [])==0 and
20     search(-100, ())==0
21
22   File line 5, in search
23     return i + 1
24     ^
25 UnboundLocalError: cannot access local variable 'i' where it is not associated with a value
26
27 ### The test cases information is:
28 passed case 1: assert search(42, (-5, 1, 3, 5, 7, 10))==6
29 passed case 2: assert search(42, [1, 5, 10])==3
30 passed case 3: assert search(5, (1, 5, 10))==1
31 passed case 4: assert search(7, [1, 5, 10])==2
32 passed case 5: assert search(3, (1, 5, 10))==1
33 passed case 6: assert search(-5, (1, 5, 10))==0
34 passed case 7: assert search(10, (-5, -1, 3, 5, 7, 10))==5
35 passed case 8: assert search(-100, (-5, -1, 3, 5, 7, 10))==0
36 passed case 9: assert search(0, (-5, -1, 3, 5, 7, 10))==2
37 failed case 1: assert search(100, [])==0
38 failed case 2: assert search(-100, ())==0
39
40 ### Reference code:
41 def search(x, seq):
42     for i in range(len(seq)):
43         if x <= seq[i]:
44             return i
45     return len(seq)

```

Figure 5: Prompt example

While the specific value of n_{basic} is not critical, we propose using a relatively small n_{basic} because basic prompts tend to converge quickly, either succeeding consistently or repeating errors. If no valid repair is found after n_{basic} iterations, the process shifts to the CoT phase for more exploratory repair attempts.

This ordering is motivated by the observation that reference code has been shown to be highly effective in facilitating bug fixes, as demonstrated in prior studies [54, 51, 18]. While the core principle of ComplementaryRepair lies in prompt diversity, prioritizing prompts with higher repair potential enhances efficiency. These studies highlight improvements achieved either through enhanced selection strategies for identifying the most beneficial reference solutions or by refining the reference pool using various optimization techniques. Prompt 4 follows, integrating all available information to maximize contextual breadth, thereby facilitating a comprehensive evaluation before narrowing focus. Prompt 2 is executed last, as its primary focus on error messages and test outcomes ensures a more targeted diagnostic approach, minimizing potential distractions from reference solution. The key of ComplementaryRepair is not about the detailed prompt design, but the diversity in prompts could contribute to complementary repairs.

4.2 CoT Prompt Repair Phase

The iterative process in the CoT prompt repair phase follows the same fundamental logic as the basic prompt repair phase but introduces two key distinctions to enhance repair success through exploratory reasoning. First, it employs the best practice of zero-shot Chain-of-Thought (CoT) prompting [22] by appending the phrase "Let's think step by step." to the end of each basic prompt as shown in Figure 6. This encourages the LLM to break down the repair process into intermediate steps, generating a rationale before proposing a fix. Second, a higher iteration limit n_{cot} is suggested compared to the basic prompt repair phase, as the effectiveness of CoT-based repair strategies is more dependent on generating a greater number of repair attempts. The higher n_{cot} reflects CoT's exploratory nature. While basic prompts tend to converge quickly, either succeeding or repeating similar errors, CoT prompts produce more varied outputs due to the variability introduced by the reasoning step. This divergence is critical for repairing buggy submissions that resist straightforward fixes. However, CoT prompting introduces higher computational costs due to longer responses, as the model typically generates explanatory rationale alongside the repaired code.

Operationally, the CoT phase follows the same prompt sequence as the basic phase $\langle prompt3 \rightarrow prompt4 \rightarrow prompt2 \rangle$, with the CoT instruction appended to each. After each response, the repaired code is extracted often from outputs, which typically containing both reasoning and repaired code, and validated against the full test suite.

```
1 Your responsibility is to provide a fix for the buggy code.
2
3 ### Problem description:
4 Sequential Search: Implement a function that determines the appropriate insertion index for a given value x in a
5 sequence seq, ensuring that the sequence remains in non-decreasing order.
6
7 ### Buggy code:
8 def search(x, seq):
9     for i in range(len(seq)):
10         if x <= seq[i]:
11             return i
12             i + 1
13
14 ### Error messages:
15 Runtime Error: Traceback (most recent call last):
16 File line 7, in <module>
17     assert search(42, (-5, 1, 3, 5, 7, 10))==6 and search(42, [1, 5, 10])==3 and search(5, (1, 5, 10))==1 and
18         search([1, 5, 10])==2 and search(3, (1, 5, 10))==1 and search(-5, (1, 5, 10))==0 and search(10, (-5, -1, 3, 5, 7,
19             10))==5 and search(-100, (-5, -1, 3, 5, 7, 10))==0 and search(0, [ ])==0 and search(-100, ( ))==0
20
21 File line 5, in search
22     return i + 1
23 UnboundLocalError: cannot access local variable 'i' where it is not associated with a value
24
25 ### The test cases information is:
26 passed case 1: assert search(42, (-5, 1, 3, 5, 7, 10))==6
27 passed case 2: assert search(42, [1, 5, 10])==3
28 passed case 3: assert search(5, (1, 5, 10))==1
29 passed case 4: assert search(7, [1, 5, 10])==2
30 passed case 5: assert search(3, (1, 5, 10))==1
31 passed case 6: assert search(-5, (1, 5, 10))==0
32 passed case 7: assert search(10, (-5, -1, 3, 5, 7, 10))==5
33 passed case 8: assert search(-100, (-5, -1, 3, 5, 7, 10))==0
34 passed case 9: assert search(0, (-5, -1, 3, 5, 7, 10))==2
35 failed case 1: assert search(-100, [ ])==0
36 failed case 2: assert search(0, ( ))==0
37
38 ### Reference code:
39 def search(x, seq):
40     for i in range(len(seq)):
41         if x <= seq[i]:
42             return i
43     return len(seq)
44
45 Let's think step by step.
```

Figure 6: CoT prompt example

4.3 Datasets

In introductory programming course assignments, the problem typically includes a well-defined problem description, at least one reference solution, a suite of test cases, and fundamental feedback such as compiler errors and test case failure details. Additionally, peer or prior submissions may be available, except in cases where the course or exercise is newly introduced. However, information such as fault locations or detailed issue descriptions is not inherently provided within the original assignment system and should not be treated as primary inputs in prompt design. To address these requirements, we propose utilizing two open datasets from the National University of Singapore [15], and the University of Dublin [3].

The Singapore dataset comprises five distinct introductory-level programming tasks and an initial collection of 4225 submissions from 361 students. This dataset is well-suited for our research due to its clearly defined programming problems, comprehensive test suites, and the presence of a variety of common errors made by novice programmers.

The Dublin dataset offers a larger scale, encompassing 34 different programming tasks and a greater volume of student submissions. For our analysis, we specifically selected submissions from the academic years 2015 to 2018, covering three academic semesters. To manage the dataset size and focus on instances where students likely require assistance, we retained only the final failed submission for each student. While this selection strategy might not capture the full spectrum of errors encountered during the programming process, the final failed submission often represents a point where students have struggled through multiple attempts. For correct submissions within the Dublin dataset, we only removed exact duplicate entries to maintain a valid pool of correct reference code.

After applying these filtering procedures, the Singapore dataset yielded 1529 buggy submissions, and the Dublin dataset contained 1475 buggy submissions. Although the number of erroneous submissions is similar across both datasets after filtering, their structural differences allow for distinct research focuses. The Dublin dataset, with its 34 unique assignments, provides a broader evaluation of the LLM’s ability to generalize repair capabilities across a wider range of programming problems. Conversely, the Singapore dataset, consisting of only five assignments, enables a more in-depth analysis of the model’s effectiveness in addressing the diverse types of errors that can arise within a specific set of programming tasks. A summary of the key characteristics of the filtered datasets is presented in Table 5.

Table 5: Datasets statistics

	#Correct	#Incorrect	Avg. #Tokens	Avg. #Lines	#Problems
Singapore	1299	1529	102	11.9	5
Dublin	1803	1475	60	6.7	34

4.4 Model and Quantization

To simulate scenarios with restricted hardware support, we primarily use the Q4-quantized version of DeepSeek-Coder:6.7B-Instruct in our study. To compare the performance of various open-source models and examine the effects of parameter size and quantization, we conducted extensive experiments evaluating ComplementaryRepair across the DeepSeek-Coder family [13] (v1-1.3B, v1-6.7B, and v2-16B) and the Qwen2.5-Coder family [17] (1.5B, 3B, 7B, and 14B). Specifically, the models we evaluated and tested are listed in Table 6. Within this rapidly advancing field, DeepSeek-Coder and Qwen2.5-Coder have emerged as prominent families of open-source language models explicitly engineered and optimized for code-related tasks. DeepSeek-Coder is presented as a series of open-source language models specifically designed to excel in coding tasks. Similarly, Qwen2.5-Coder is introduced as a code-specific model built upon the Qwen2.5 architecture, with a focus on programming-related applications.

Table 6: All artifacts used in this study

Model	Size	Quantaztion	Public Link
ds-coder-instruct	1.3B	q3	https://ollama.com/library/deepseek-coder:1.3b-instruct-q3_K_M
		q4	https://ollama.com/library/deepseek-coder:1.3b-instruct-q4_K_M
		fp16	https://ollama.com/library/deepseek-coder:1.3b-instruct-fp16
ds-coder-v2-instruct	6.7B	q3	https://ollama.com/library/deepseek-coder:6.7b-instruct-q3_K_M
		q4	https://ollama.com/library/deepseek-coder:6.7b-instruct-q4_K_M
		fp16	https://ollama.com/library/deepseek-coder:6.7b-instruct-fp16
ds-coder-v2-instruct	16B	q3	https://ollama.com/library/deepseek-coder-v2:16b-lite-instruct-q3_K_M
		q4	https://ollama.com/library/deepseek-coder-v2:16b-lite-instruct-q4_K_M
qwen2.5-coderinstruct	1.5B	q3	https://ollama.com/library/qwen2.5-coder:1.5b-instruct-q3_K_M
		q4	https://ollama.com/library/qwen2.5-coder:1.5b-instruct-q4_K_M
		fp16	https://ollama.com/library/qwen2.5-coder:1.5b-instruct-fp16
	3B	q4	https://ollama.com/library/qwen2.5-coder:3b-instruct-q4_K_M
		fp16	https://ollama.com/library/qwen2.5-coder:3b-instruct-fp16
	7B	q3	https://ollama.com/library/qwen2.5-coder:7b-instruct-q3_K_M
		q4	https://ollama.com/library/qwen2.5-coder:7b-instruct-q4_K_M
	14B	fp16	https://ollama.com/library/qwen2.5-coder:7b-instruct-fp16
		q3	https://ollama.com/library/qwen2.5-coder:14b-instruct-q3_K_M
q4	https://ollama.com/library/qwen2.5-coder:14b-instruct-q4_K_M		

4.4.1 Deepseek Coder

The DeepSeek-Coder family [13] is a series of open-source language models optimized for code-related tasks, with model sizes ranging from 1 billion to 33 billion parameters, providing users with the flexibility to select a model that aligns with their specific computational resources and performance needs. Each model is pre-trained from scratch on a massive corpus of 2 trillion tokens, comprising 87% code and 13% natural language in English and Chinese, enabling robust handling of both programming and explanatory text. A notable aspect of the data preparation is the organization of pre-training data at the repository level, along with the use of dependency parsing. This approach is designed to enable the model to better understand code within the context of multi-file projects, mirroring the complexities of real-world software

development. This design makes DeepSeek-Coder particularly effective for tasks like APR where understanding instruction and generating valid codes are critical. Specifically, the instruction-tuned versions, especially the 33B parameter variant, have demonstrated performance comparable to or exceeding that of OpenAI’s GPT-3.5 Turbo on code evaluation benchmarks, significantly narrowing the performance gap with more advanced models like GPT-4. The models’ open-source nature also enables local deployment, addressing privacy and cost concerns compared to closed-source models.

4.4.2 Qwen2.5 Coder

The Qwen2.5 Coder family [17] is developed by Alibaba Qwen team, consisting of models with sizes ranging from 0.5 billion to 32 billion parameters. The architecture of Qwen2.5 Coder is directly derived from the Qwen2.5 series of LLMs, inheriting its fundamental design and tokenizer. All models within the series share a core architecture in terms of head size but differ in parameters such as hidden size, the number of layers, and attention heads. These models undergo pre-training on an extensive dataset exceeding 5.5 trillion tokens, named Qwen2.5-Coder-Data. This dataset is a carefully constructed mix of source code from 92 programming languages, text-code grounding data, synthetic data generated using previous Qwen models, where math-related data, and general natural language data with code segments removed. The pre-training process involves a sophisticated three-stage pipeline encompassing file-level pretraining, repo-level pretraining, and post-training instruction tuning. Currently, Qwen2.5 Coder-32B-Instruct has emerged as a state-of-the-art open-source code model, demonstrating coding capabilities that match those of OpenAI’s GPT-4o on several benchmarks.

4.5 Testing Environment

A primary objective of this research is to demonstrate the feasibility of deploying ComplementaryRepair in resource-constrained educational settings by utilizing small, quantized LLMs locally, thereby avoiding reliance on costly cloud-based Application Programming Interfaces (APIs). Consequently, all experiments were conducted on a standard mid-range desktop PC with the hardware specifications detailed in Table 7.

Table 7: Hardware Specifications

Component	Specifications
CPU	13th Gen Intel(R) Core(TM) i5-13600K 3.50 GH
GPU	NVIDIA RTX 4060Ti 16GB
RAM	32GB DDR5 6000MHZ

The processes of code generation and evaluation were implemented in a Python environment leveraging the Ollama API on the Windows 11 operating system. The key software components and their respective versions are listed in Table 8.

Table 8: Software Specifications

Component	Version
Operating System	Windows 11 Pro 23H2
Cuda	12.6
Python	3.12.2
Ollama	0.6.2
Transformers	4.47.0

For the LLM inference settings, we primarily adhered to the default parameters provided by the Ollama API. However, to carefully manage the balance between the diversity and coherence of the generated code, we set the temperature parameter to 0.2. Furthermore, recognizing that the input prompts, comprising student code, task descriptions, error messages, and potential modal solutions, could be substantial in length, we increased the context window size from the default 2048 tokens to 4096 tokens. This adjustment ensured that the models could effectively utilize the complete contextual information provided in the prompt. To prevent excessively long or potentially infinite generation, we set the maximum number of tokens to be generated by the LLM to 2048.

The evaluation of the functional correctness of the generated repairs was conducted by appending the extracted repaired code with all the provided test cases and executing the resulting code. A repair was considered successful if the execution completed without any runtime errors and all test cases passed. Conversely, the occurrence of an `AssertionError` or any other runtime exception was classified as a failed repair. Given that the original datasets did not specify explicit time or memory constraints for the programming assignments, we applied lenient limits on execution time and memory usage during the evaluation process, prioritizing the assessment of functional correctness over performance optimization.

5 Evaluation and Results

To clarify, ComplementaryRepair includes 3 basic prompts in the basic phase and 3 CoT prompts in the CoT phase, executed sequentially in a fixed order (*Prompt3* → *Prompt4* → *Prompt2* in our settings). Each phase iterates this sequence as a loop. The process stops immediately once a generated repair passes all test cases. Therefore, with n_{basic} iterations for the basic phase and n_{cot} iterations for the CoT phase, a maximum of $3(n_{basic} + n_{cot})$ repairs could be generated per buggy code if no valid repair is found earlier (or if one is found in the final prompt of the last iteration). In the table below, "Avg. #Repairs" and "Avg. #Tokens" represent the average number of responses generated and the total tokens produced per buggy code before a valid repair is found or the maximum iteration limit is reached respectively. These metrics provide a measure of the computational cost of ComplementaryRepair.

We focus our analysis on models with parameter size around 7 to 8B, as they can generally fit within one large GPU while ensuring a relatively large context. Specifically, we utilize the Deepseek-Coder-Instruct:6.7B [13], and with the INT4 quantization version, which could help save the memory requirement a lot while keeping an acceptable performance reduction.

5.1 RQ1: Effectiveness of ComplementaryRepair

We conducted experiments across different settings, with different number of iterations in the two phases. We first grouped the experiments by the total number of iterations ($n_{basic} + n_{cot}$) and then evaluated performance based on different distributions of these iterations between the two phases.

5.1.1 Results on Singapore

Several key observations can be drawn from the results of the experiments, as shown in Table 9:

Table 9: Results of ComplementaryRepair performance in fixing Singapore (1529 buggy code submissions in total)

Model	Size	Basic Phase #Iteration	CoT Phase #Iteration	Repaired in Basic	Repaired in CoT	Fail	Repair Rate	Avg. #Repairs	Avg. #Tokens
ds-coder-instruct	6.7B	1	0	1416	-	113	92.6	1.40	297
		0	1	-	1430	99	93.5	1.39	494
		2	0	1467	-	62	95.9	1.56	348
		1	1	1417	87	25	98.4	1.50	349
		0	2	-	1492	37	97.6	1.55	561
		3	0	1483	-	46	97.0	1.66	370
		2	1	1468	43	18	98.8	1.63	372
		1	2	1419	99	11	99.3	1.54	363
		0	3	-	1522	7	99.5	1.49	532

- **First, ComplementaryRepair efficiently repairs most buggy codes.** With just one iteration in the basic prompt phase and one in the CoT prompt phase (1,1),

it achieves a repair rate of 98.4%. Increasing iterations in either phase slightly increases the repair rate, reaching 98.8% with an additional basic phase iteration (2,0) and 99.3% with an extra CoT phase iteration (1,2).

- **Second, the CoT prompt repair phase demonstrates greater exploratory potential compared to basic prompt repair phase.** Across paired settings with equal total iterations: (1,0) vs. (0,1), (2,0) vs. (0,2), and (3,0) vs. (0,3), the CoT-only strategy consistently produces higher or at least equal repair rates compared to basic-only strategy. The (0,3) combination even achieves the highest repair rate of 99.5%, outperforming all other groups.
- **Third, while a CoT-only strategy could deliver high repair rate, this improvement comes at a significant computational cost.** This cost is reflected in the average number of tokens generated per buggy code until a valid repair is found or the iteration limit is reached (Avg. #Tokens). For instance, (0,2) requires an average of 561 tokens per buggy code, significantly higher than the 348 for (2,0). Similarly, (0,3) averages 532 tokens, substantially exceeding the costs of (3,0), (2,1) and (1,2).
- **Fourth, combining basic and CoT phases provides a balanced trade-off between repair rate and computational cost.** For group with two total iterations, the (1,1) combination reduces the average tokens from 561 in (0,2) to 349, while even increasing the repair rate by 0.8%. Compared to (2,0), (1,1) increases the repair rate from 95.9% to 98.4% only 1 token increased in Avg. #Tokens (though this may partly due to LLM randomness, it suggests close computational costs). Similar patterns hold for (2,1) and (1,2) versus (3,0) and (0,3), highlighting the efficiency of mixed strategies.
- **Lastly, when both phases have at least one iteration, a higher number of CoT iterations (n_{cot}) tends to be more effective than a higher number of basic iterations (n_{basic}).** For instance, (1,2) outperforms (2,1) in repair rate, average number of repairs (Avg. #Repairs), and average tokens (Avg. #Tokens). This reflects the divergent nature of CoT prompts in small LLMs discussed earlier, which produce more varied and complementary repairs, unlike basic prompts that often generate similar attempts with correlated success or failure.

We also compare ComplementaryRepair with the current state-of-the-art APR method tested on the Singapore dataset from Ishizue et al.[18]. Their approach integrates code refactoring techniques, drawing from the work of Hu et al. [15], with the capabilities of ChatGPT to repair buggy student submissions. This method involves providing detailed instructions outlining the repair requirements, along with modal solutions derived from refactored code exhibiting the highest structural similarity to the erroneous code. Given that their study did not focus on computational cost, our comparison primarily centers on repair rates. The detailed result is shown in the Table 10.

For this comparison, we select three ComplementaryRepair configurations: (0,1), (1,1), and (0,3). The reasoning behind these choices is as follows:

- **(0,1)**: To ensure a fair comparison with the state-of-the-art method, which performs a maximum of three repair attempts per buggy code, we selected the (0,1) ComplementaryRepair configuration. This configuration also adheres to a maximum of three attempts (one CoT iteration) and achieved a repair rate of 93.3% in our experiments.
- **(1,1)**: This configuration, allowing up to six repair attempts, represents the core design of ComplementaryRepair, incorporating at least one iteration of both the basic and CoT prompting phases. It provides insight into the performance of our fundamental approach.
- **(0,3)**: This configuration, permitting a maximum of nine repair attempts, yielded the highest repair rate in our experimental evaluation, reaching 99.3%. It demonstrates the upper-bound performance achievable by ComplementaryRepair with increased exploration.

Table 10: Comparison with state-of-the-art

Method	Model	Size	Q1	Q2	Q3	Q4	Q5	Avg. Repair Rate
Refactory Only	-	-	99.3	78.6	97.4	85.4	83.3	88.8
Refactory with GPT	gpt-3.5	175B	99.3	96.6	99.4	95.8	87.0	95.6
Refactory with GPT	gpt-4	-	99.8	99.5	99.4	98.3	89.8	97.4
ComplementaryRepair (0,1)			93.9	87.3	97.8	97.3	90.7	93.5
ComplementaryRepair (1,1)	ds-coder-instruct	6.7B	98.4	97.1	99.6	99.7	95.4	98.4
ComplementaryRepair (0,3)			99.8	98.3	100	100	100	99.5

It is important to note that while the most direct comparison is with the (0,1) configuration, which achieved a repair rate of 93.3% and underperformed the reported state-of-the-art repair rates of 95.6% and 97.4%, the state-of-the-art method relies on two advanced commercial LLMs with significantly larger parameter counts. In contrast, ComplementaryRepair achieved its 93.3% repair rate using a considerably smaller 6.7 billion parameter quantized model. This highlights the potential cost-efficiency and effectiveness of our approach even with more resource-constrained models. Furthermore, when ComplementaryRepair was allowed a greater number of attempts, as in the (1,1) and (0,3) configurations, it surpassed the state-of-the-art repair rates, achieving 98.4% and 99.5% respectively, compared to the state-of-the-art’s highest reported rate of 97.4%. This suggests that with increased exploration, ComplementaryRepair can achieve superior repair performance, even when compared to methods leveraging much larger commercial models.

5.1.2 Results on Dublin

Compared to the Singapore dataset, Dublin contains much more problems, and thus potentially more types of errors among different types of problems. Overall speaking, the observations and conclusion drawn from the results of Singapore dataset also holds for the Dublin datasets:

- **First, Complementary Repair demonstrated robust effectiveness on the Dublin dataset, achieving a repair rate of 92.7% with the (1,1) prompting configuration.** Further increasing the number of CoT iterations to two in the (1,2) configuration resulted in an improved repair rate of 94.0%, indicating the continued benefit of deeper exploration through CoT prompting.
- **Second, consistent with the findings from the Singapore dataset, the CoT prompting phase exhibited a greater capacity for exploring the solution space compared to the basic prompting phase.** However, a notable difference in the Dublin dataset was that a strategy relying solely on CoT prompts did not consistently outperform a mixed strategy (combining basic and CoT prompts). Specifically, the (1,1) configuration, employing one iteration of both basic and CoT prompts, achieved a higher repair rate (92.7%) than the (0,2) configuration, which utilized two CoT prompt iterations (92.0%). Furthermore, the (1,1) strategy exhibited greater efficiency, requiring fewer average tokens (283) compared to the (0,2) strategy (437). This outcome suggests that, for the Dublin dataset, the integration of basic and CoT prompts yielded a more effective and resource-efficient repair process than relying solely on CoT-based exploration.
- **Third, the CoT-only strategy continued to exhibit a higher computational cost, as evidenced by a significantly larger average number of generated tokens.** For instance, utilizing one basic prompt iteration (1,0) resulted in an average of 210 tokens, whereas a single CoT prompt iteration (0,1) yielded a substantially higher average of 369 tokens. This reinforces the observation that the more elaborate reasoning process inherent in CoT prompting incurs greater computational resource consumption.
- **Fourth, the mixed prompting strategy effectively balanced repair rate and computational cost.** The (1,1) configuration not only outperformed the (0,2) configuration in terms of repair rate but also required fewer average tokens. When compared to using two basic prompt iterations (2,0), the (1,1) strategy increased the repair rate from 91.1% to 92.7% with a relatively small increase of 18 tokens in the average number of tokens generated, establishing it as the most cost-efficient 2-iteration setting.
- **Finally, a higher number of CoT iterations n_{cot} still tended to yield a better repair rate than a higher number of basic iterations n_{basic} when both prompting phases were allocated at least one iteration, although this advantage was less pronounced than in the Singapore dataset.** For example, the (1,2) configuration achieved a 0.3% higher repair rate than the (2,1) configuration, at an additional cost of 14 tokens on average. This contrasts with the Singapore results, where increasing n_{cot} in similar configurations not only improved the repair rate but also sometimes reduced the average token count.

In summary, the results obtained from the Dublin dataset largely mirrored the key observations from the Singapore dataset, affirming the overall effectiveness of

ComplementaryRepair and the distinct characteristics of basic and CoT prompting. However, the Dublin results underscored a more pronounced synergistic effect when basic and CoT prompts were combined to maximize both repair success and computational efficiency. Furthermore, the advantage of increasing CoT iterations over basic iterations, while still present, was less pronounced in the Dublin experiments compared to the Singapore ones.

Table 11: Results of ComplementaryRepair performance in fixing Dublin (1475 buggy code submissions in total)

Model	Size	Basic Phase #Iteration	CoT Phase #Iteration	Repaired in Basic	Repaired in CoT	Fail	Repair Rate	Avg #Repairs	Avg #Tokens
ds-coder-instruct	6.7B	1	0	1298	-	177	88.0	1.36	210
		0	1	-	1298	177	88.0	1.41	369
		2	0	1344	-	131	91.1	1.68	265
		1	1	1294	73	108	92.7	1.64	283
		0	2	-	1357	118	92.0	1.68	437
		3	0	1341	-	134	90.9	1.98	312
		2	1	1326	56	93	93.7	1.95	332
		1	2	1290	96	89	94.0	1.88	346
		0	3	-	1370	105	92.9	1.93	498

5.2 RQ2: Impact of Prompt Diversity and Pipeline Design

We conducted experiments to assess the impact of prompt design strategies, specifically focusing on diverse prompting strategies and the two-phase pipeline design. To evaluate the effect of diverse prompts, we compared our standard approach, which use three distinct prompts per iteration, against an alternative where a single prompt is reused to generate three responses per iteration, maintaining the same maximum response generation limit. Specifically, we performed three sets of experiments: one with the full two-phase design (basic and CoT prompt phases), one using only the basic prompt repair phase, and one using only the CoT prompt repair phase. In these experiments, the first set follows the prompt sequence of ComplementaryRepair (*Prompt3* → *Prompt4* → *Prompt2*). The other sets each use a single prompt, either *Prompt2*, *Prompt3*, or *Prompt4*, and its corresponding CoT version, repeated 3 times to ensure a fair comparison. Detailed results are shown in Table 12 13.

The results demonstrate that diverse prompts significantly outperform single-prompt strategies across all experimental settings. For example, in the Singapore dataset, the diverse prompt approach in the basic phase achieves a repair rate of 92.6%, substantially higher than single-prompt baselines using Prompt 2 (70.2%), Prompt 3 (81.2%), or Prompt 4 (83.2%). Moreover, it maintains a low computational cost, averaging 297 tokens (Avg #Tokens), which is lower than or comparable to the single-prompt groups (448, 297, and 356 tokens, respectively). Similar trends hold for the CoT phase and two-phase experiments, where diverse CoT prompts yield higher repair rates while keeping Avg #Tokens competitive. These findings highlight the effectiveness of prompt diversity in improving repair success and computational efficiency.

Table 12: Results of different prompts design in fixing Singapore (1529 buggy code submissions in total)

Model	Size	Prompts	Repaired in Basic	Repaired in CoT	Fail	Repair Rate	Avg. #Repairs	Avg. #Tokens
ds-coder-instruct	6.7B	basic(p3,p4,p2) ->cot(p3,p4,p2)	1417	87	25	98.4	1.50	349
		basic(p2,p2,p2) ->cot(p2,p2,p2)	1035	222	272	82.2	2.42	717
		basic(p3,p3,p3) ->cot(p3,p3,p3)	1241	234	54	96.5	1.79	425
		basic(p4,p4,p4) ->cot(p4,p4,p4)	1324	157	48	96.9	1.67	436
		basic(p3,p4,p2)	1416	-	113	92.6	1.40	297
		basic(p2,p2,p2)	1074	-	455	70.2	1.70	448
		basic(p3,p3,p3)	1242	-	287	81.2	1.49	297
		basic(p4,p4,p4)	1243	-	286	83.2	1.45	356
		cot(p3,p4,p2)	-	1430	99	93.5	1.39	494
		cot(p2,p2,p2)	-	1184	345	77.4	1.68	593
		cot(p3,p3,p3)	-	1359	170	88.9	1.44	525
		cot(p4,p4,p4)	-	1341	188	87.7	1.44	510

Table 13: Results of different prompts design in fixing Dublin (1475 buggy code submissions in total)

Model	Size	Prompts	Repaired in Basic	Repaired in CoT	Fail	Repair Rate	Avg. #Repairs	Avg. #Tokens
ds-coder-instruct	6.7B	basic(p3,p4,p2) ->cot(p3,p4,p2)	1294	73	108	92.7	1.64	283
		basic(p2,p2,p2) ->cot(p2,p2,p2)	995	126	354	76.0	2.50	502
		basic(p3,p3,p3) ->cot(p3,p3,p3)	1242	120	113	92.3	1.73	298
		basic(p4,p4,p4) ->cot(p4,p4,p4)	1163	132	180	87.8	1.95	378
		basic(p3,p4,p2)	1298	-	177	88.0	1.36	210
		basic(p2,p2,p2)	1009	-	466	68.4	1.68	281
		basic(p3,p3,p3)	1238	-	237	83.9	1.40	211
		basic(p4,p4,p4)	1165	-	310	79.0	1.47	252
		cot(p3,p4,p2)	-	1298	177	88.0	1.41	369
		cot(p2,p2,p2)	-	1063	412	72.1	1.69	468
		cot(p3,p3,p3)	-	1288	187	87.3	1.41	365
		cot(p4,p4,p4)	-	1209	266	82.0	1.47	387

The benefits of the two-phase design have been partly demonstrated in previous sections, where the two-phase design effectively balances repair rate and computational cost. To further reveal the two-phase design, we conducted additional experiments reversing the phase order: starting with the CoT prompt repair phase followed by the basic prompt repair phase. The results are presented in Table 14. For the Singapore dataset, besides the slight improvement in repair rate, the proposed standard ComplementaryRepair structure (basic phase first, CoT phase second) significantly outperforms the reversed order in efficiency, reducing the Avg. #Tokens from 519 to 349. Similarly, for the Dublin dataset, the standard structure not only improves efficiency by lowering Avg. #Tokens from 414 to 283, but also achieves a higher repair rate (92.7% compared to 91.7%). These results indicate that the proposed structure maintains the overall repair rate while substantially reducing computational cost, as measured by Avg. #Tokens.

Table 14: Results of different phase orders

Model	Size	Datasets	Method	Repaired in First Phase	Repaired in Second Phase	Fail	Repair Rate	Avg. #Repairs	Avg. #Tokens
ds-coder-instruct	6.7B	Singapore	Basic ->CoT	1417	87	25	98.4	1.50	349
			CoT ->Basic	1426	76	27	98.2	1.49	519
		Dublin	Basic ->CoT	1294	73	108	92.7	1.64	283
			CoT ->Basic	1299	54	122	91.7	1.68	414

5.3 RQ3: Model Size, Quantization, and Performance

We investigated how varying the size and quantization level of the underlying LLMs impacts the effectiveness of ComplementaryRepair. To ensure a consistent evaluation, we employed the (1,1) prompting configuration (one basic and one CoT iteration). Our analysis included the DeepSeek-Coder family (1.3B, 6.7B, and 16B parameters) and the Qwen2.5-Coder family (1.5B, 3B, 7B, and 14B parameters), each tested across different quantization levels. The detailed performance metrics are presented in Table 15 and Table 16. Our findings reveal two primary insights:

First, quantization offers a substantial reduction in model size with minimal performance degradation. Across all tested LLMs, quantizing the models significantly decreased their memory footprint while maintaining comparable repair capabilities to their full-precision counterparts. Specifically, models quantized to 4 bits (Q4) exhibited negligible differences in repair accuracy on introductory programming tasks. However, reducing the quantization to 3 bits (Q3) resulted in a more noticeable performance drop, particularly pronounced in smaller models. For example, the Qwen2.5-Coder 1.5B model experienced a decrease in repair rate from 93.9% (Q4) to 88.4% (Q3) on the Singapore dataset, accompanied by an increase in the average number of tokens generated (from 562 to 721). Conversely, larger models like the Qwen2.5-Coder 7B and 14B showed only minor performance variations between Q3 and Q4 quantization, with Q3 occasionally even exhibiting slightly better repair rates (likely due to inherent LLM variability, indicating practically equivalent performance). These results underscore the utility of quantization as a resource-efficient strategy. Notably, Q4-quantized versions of larger models consistently outperformed full-precision smaller models of similar size, highlighting the benefits of scale even with quantization. For instance, Qwen2.5-Coder 14B (Q4) surpassed Qwen2.5-Coder 7B (FP16), and Qwen2.5-Coder 3B (Q4) outperformed Qwen2.5-Coder 1.5B (FP16) in terms of repair rate.

Second, ComplementaryRepair provides a greater relative advantage for smaller LLMs. Smaller models, which inherently exhibit a higher failure rate on average, benefit more significantly from the diverse repair attempts generated by ComplementaryRepair. We observed a trend where larger models typically achieved successful repairs with fewer attempts, indicated by a lower average number of repairs. In contrast, smaller models leveraged ComplementaryRepair’s ability to produce varied and meaningful repair suggestions, avoiding repetitive outputs often seen with single prompting strategies. This enabled smaller models to achieve competitive repair rates despite the baseline performance superiority of larger models. For the Singapore dataset, the DeepSeek-Coder 6.7B model even outperformed the 16B model in repair rate

while generating fewer average tokens. However, on the more challenging Dublin dataset, the performance gains associated with larger models were more pronounced, suggesting that problem complexity amplifies the performance differences between model sizes. Therefore, the selection of an LLM for automated program repair should consider the difficulty level of the programming assignments. For highly complex tasks, larger models may offer a substantial advantage, whereas for assignments of appropriate difficulty, ComplementaryRepair can effectively mitigate the performance gap between smaller and larger LLMs. Considering that introductory programming assignments are generally not excessively complex, we recommend utilizing models around the 7 billion parameter scale, as they offer a robust balance between repair effectiveness and computational demands.

Table 15: Results of different LLMs in fixing Singapore (1529 buggy code submissions in total): the result demonstrates that Q4 quantization offers substantial memory reduction with minimal performance loss across DeepSeek-Coder and Qwen2.5-Coder families. Smaller models benefit relatively more from ComplementaryRepair’s diverse attempts, enabling competitive repair rates, with Q4-quantized larger models outperforming full-precision smaller counterparts.

Model	Size	Quantization	Repaired in Basic	Repaired in CoT	Fail	Repair Rate	Avg #Repairs	Avg #Tokens
ds-coder-instruct	1.3B	q3	172	108	1249	18.3	5.47	2882
		q4	253	124	1152	24.7	5.25	3041
		fp16	265	146	1118	26.9	5.17	3121
	6.7B	q3	1428	71	30	98.0	1.51	412
		q4	1417	87	25	98.4	1.50	349
		fp16	1432	73	24	98.4	1.47	353
ds-coder-v2-instruct	16B	q3	1472	35	22	98.6	1.27	512
		q4	1451	49	29	98.1	1.34	518
qwen2.5-coder-instruct	1.5B	q3	1187	165	177	88.4	2.26	721
		q4	1331	104	94	93.9	1.83	562
		fp16	1256	169	104	93.2	2.02	600
	3B	q4	1387	87	55	96.4	1.64	550
		fp16	1392	86	51	96.7	1.58	533
	7B	q3	1490	33	6	99.6	1.20	247
		q4	1493	29	7	99.5	1.2	254
		fp16	1492	25	12	99.2	1.19	251
	14B	q3	1522	6	1	99.9	1.06	332
		q4	1516	9	4	99.7	1.08	340

Table 16: Results of different LLMs in fixing Dublin (1475 buggy code submissions in total): on the more challenging Dublin dataset, the performance advantage of larger LLMs becomes more pronounced. While quantization still offers memory savings with limited impact at Q4, the benefits of scale are amplified by task complexity, suggesting that larger models yield more significant gains for harder programming assignments.

Model	Size	Quantaztion	Repaired in Basic	Repaired in CoT	Fail	Repair Rate	Avg #Repairs	Avg #Tokens
ds-coder-instrct	1.3B	q3	387	161	927	37.2	4.79	2148
		q4	552	209	714	51.6	4.30	2115
		fp16	563	204	708	52	4.26	2196
	6.7B	q3	1290	64	121	91.8	1.68	341
		q4	1294	73	108	92.7	1.64	283
		fp16	1309	61	105	92.9	1.62	291
ds-coder-v2-instruct	16B	q3	1268	56	151	89.8	1.75	477
		q4	1321	56	98	93.4	1.59	393
qwen2.5-coder-instruct	1.5B	q3	1112	152	211	85.7	2.27	494
		q4	1191	108	176	88.1	1.97	404
		fp16	1178	108	189	87.2	2.08	409
	3B	q4	1261	77	137	90.7	1.83	451
		fp16	1228	110	137	90.7	1.89	493
	7B	q3	1268	66	141	90.4	1.73	264
		q4	1291	57	127	91.4	1.65	252
		fp16	1300	53	122	91.7	1.61	238
	14B	q3	1351	36	88	94	1.43	376
		q4	1345	37	93	93.7	1.45	385

6 Discussion

6.1 Summarizing Research Questions and Answers

Below, we summarize the answers to our research questions (RQs) based on the experimental findings:

RQ1: Effectiveness of ComplementaryRepair

ComplementaryRepair efficiently repairs introductory programming assignments, achieving high repair rates with modest computational resources. It fixes up to 99.5% of buggy submissions in the Singapore dataset and 94.0% in the Dublin dataset. Given that our framework is primarily tested based on a quantized (Q4) 6.7B model (DeepSeek-Coder), which has significantly lower computational requirements than larger or closed-source models, ComplementaryRepair substantially enhances the repair capabilities of small LLMs. This offers a cost-efficient, locally deployable alternative, reducing reliance on expensive APIs and addressing privacy concerns.

RQ2: Impact of Prompt Diversity and Pipeline Design.

Prompt diversity significantly increases repair rates compared to using a single prompt. By leveraging varied prompts, ComplementaryRepair achieves higher success with fewer attempts, reducing the average number of tokens (Avg #Tokens) and thus computational cost. The pipeline design, which executes basic prompts first before CoT prompts optimizes efficiency, yielding repair rates comparable to a CoT-first approach but with substantially lower Avg #Tokens.

RQ3: Model Size, Quantization, and Performance.

ComplementaryRepair performs robustly across various small LLMs. Quantization (e.g., Q4) significantly reduces model size for all tested LLMs while maintaining competitive repair performance compared to their full-precision counterparts. Notably, ComplementaryRepair provides greater relative benefits to smaller models than to larger ones, enhancing their effectiveness in APR tasks despite limited parameters. This demonstrates the framework’s ability to empower resource-constrained models for educational settings.

6.2 ComplementaryRepair and other APR approaches

The key idea of ComplementaryRepair is using prompt diversity to increase overall repair rates. While diversity drives our approach, the quality of individual prompts remains essential. Our preliminary experiments on the Singapore dataset also demonstrate this, where distinct prompts yield varied pass@k values, highlighting the impact of prompt design. Therefore, we draw inspiration from existing APR methods to craft effective prompts, tailoring them to leverage complementary repair effects.

For instance, Peer-aided Repair [54] emphasizes the role of reference code in guiding LLMs to fix buggy submissions, suggesting that successful peer submissions form a valuable reference pool. They argue that reference code most similar to the buggy code provides the greatest insight. Inspired by this, we select reference code using CodeBLEU [34], which measures syntactic and semantic similarity, choosing the submission with the highest score to inform repairs.

Test cases and error messages are also common in APR prompts. For example, ChatRepair [48] iteratively tests generated code and incorporates new error messages if repairs fail, refining the LLM’s focus. In contrast, ComplementaryRepair collects only the initial buggy code’s error messages, as our framework prioritizes exploring diverse repair paths over iterating on a single reasoning trajectory.

The concept of complementary repairs is partly inspired by Pydex [51], which notes that varied prompts can outperform a single prompt in repair rate. However, Pydex focuses on dividing APR process into syntax and semantic phases without fully exploring or evaluating complementary effects. Additionally, it generates multiple responses from different prompts without a structured pipeline or consideration of computational cost. In contrast, ComplementaryRepair proposes a deliberate pipeline to maximize complementary effects by exploiting prompt characteristics. Basic prompts, with high pass@1 and lower average token counts (Avg #Tokens), offer efficient, stable repairs. CoT prompts, with higher pass@k at larger k and more tokens, excel at divergent exploration, avoiding repetitive errors. This informs our design: basic prompts are used first with fewer iterations for quick wins, followed by CoT prompts to tackle unresolved cases creatively, optimizing both repair success and computational efficiency.

6.3 Program Repairs and Similarity

In the evaluation, we did not mention about the modification distance between the original buggy code and the repaired code. In practice, we would prefer repaired code that with minimal modification and higher similarity, as we want the generated code to be based on the original buggy code rather than a completely new code. Only this kind of repaired code could provide more insights related to the errors, and give values to students.

To evaluate whether ComplementaryRepair can repair code with minimal modifications, we measured the similarity between the buggy code and the repaired code based on edit distance. The results are shown in Figure 7. The similarity is relatively stable among different structural settings of ComplementaryRepair, with all median similarity values around 0.64. This suggests that ComplementaryRepair repairs buggy code rather than generating nearly new code. Additionally, repairs generated by CoT prompts tend to be less similar to the original buggy code compared to those from basic prompts. This pattern is consistent across all three group comparisons: (1,0) vs. (0,1), (2,0) vs. (0,2), and (3,0) vs. (0,3). This difference may stem from the divergent nature of CoT prompts, which tends to produce repairs that deviate more from the original code.

For the Dublin dataset, the similarity results is shown in Figure 8, which are more promising than those for Singapore. The median similarity values in Dublin are consistently around 0.8, higher than the 0.65 observed in Singapore. A similar trend is also noted in Dublin, where repairs generated by CoT prompts are less similar to the original buggy code compared to those produced by basic prompts. Overall, these findings suggest that the level of modification introduced by ComplementaryRepair remains acceptable.

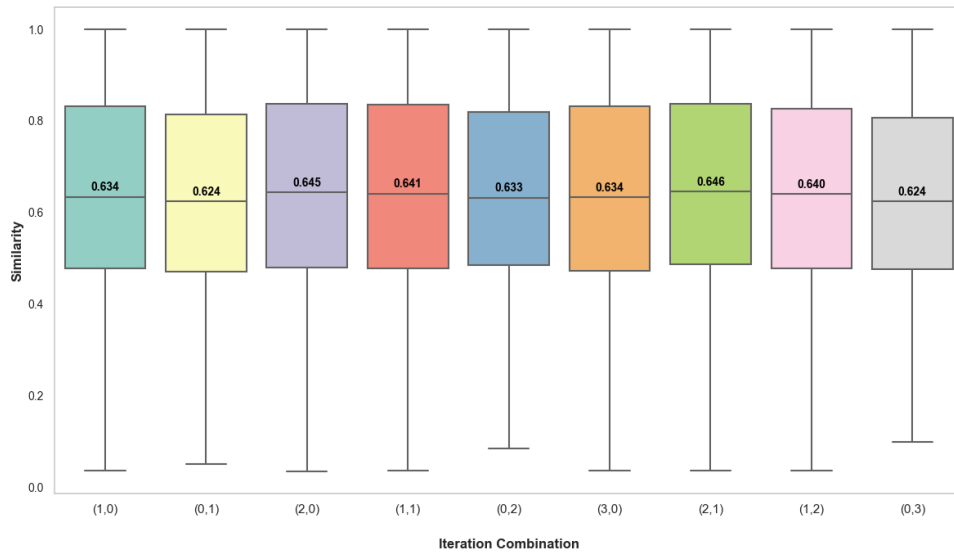


Figure 7: Similarity between buggy and repaired code across different ComplementaryRepair settings for Singapore

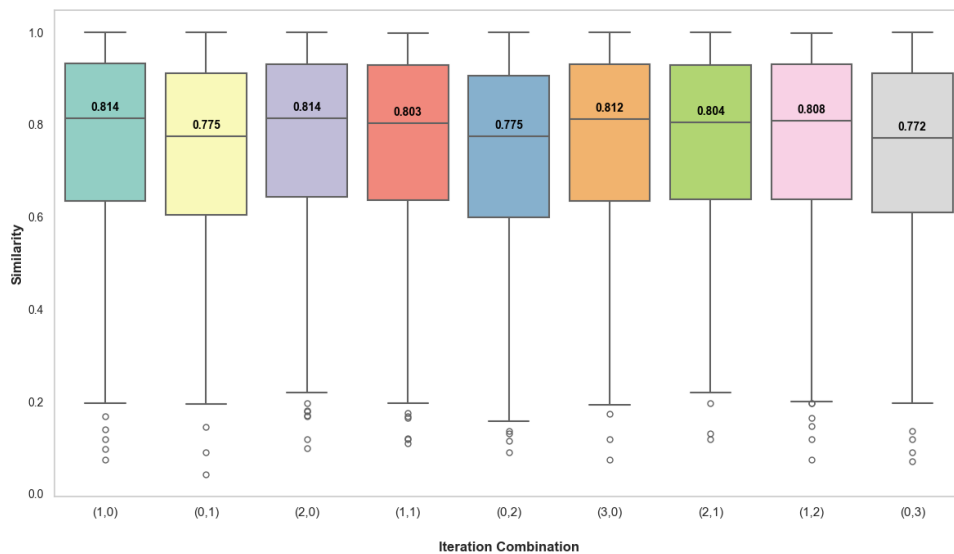


Figure 8: Similarity between buggy and repaired code across different ComplementaryRepair settings for Dublin

6.4 Threats to Validity

Internal Validity. The original datasets lack explicit time and memory constraints, which are common in programming education and affect solution design and feasibility. To address this, we applied relaxed time and memory limits during testing, focusing primarily on repair accuracy. Additionally, our setup assumes access to peer submissions as a source of reference code, which may not be available in all educational contexts. However, ComplementaryRepair aims to exploit diverse prompts

to generate varied solution paths and complementary repairs, so the specific prompt components are less critical than their diversity. Another concern arises from the inherent randomness of LLMs. We set the temperature to a low value (0.2) to reduce variability, but some uncertainty in repair generation persists. Given the close statistical margins in some of our results, this randomness could introduce bias, potentially skewing the observed outcomes.

External Validity. Both datasets used in this study consist exclusively of Python programming assignments. As a result, our evaluation of the complementary effects and performance of ComplementaryRepair is limited to Python programs, and its effectiveness for other programming languages remains unexplored. Additionally, both datasets focus on introductory-level programming assignments, typically consisting of single-file programs with one or a few functions and well-defined problem statements. Consequently, our study evaluates ComplementaryRepair in the context of introductory level programming tasks, without assessing its applicability to more complex programming assignments.

7 Conclusion

We propose ComplementaryRepair, an APR framework designed to repair introductory programming assignments. The proposed framework primarily utilizes the complementary repairs contributed by diverse prompts. This framework leverages complementary repairs generated by diverse prompts. The verifiable nature of introductory programming assignments makes exploring more potential solutions a promising strategy for improving repair rates. ComplementaryRepair operates in two phases: a basic prompt repair phase and a CoT prompt repair phase. The two-phase design allows the basic prompt phase to quickly generate valid repairs for most buggy code, while the CoT phase exploits its divergent nature to explore a wider range of potential solutions for difficult cases unresolved by the basic phase. This combination enables ComplementaryRepair to achieve a strong balance between repair rate and computational cost.

We evaluate this framework on two open-source datasets of introductory programming assignments from the National University of Singapore [15] and the University of Dublin [3]. Results demonstrate a strong balance between repair rate and computational cost. For instance, a repair-rate-oriented setting fixes 99.5% of buggy code in the Singapore dataset, averaging 1.49 attempts and 532 tokens per assignment, while a balanced setting achieves 98.4% with 349 tokens. We developed ComplementaryRepair on a small-sized, quantized version of an LLM, specifically the DeepSeek-Coder:6.7B model with Q4 quantization. Our experiments demonstrate that, with an effective pipeline design, even this compact model can outperform state-of-the-art GPT-4-based methods in repair rate for introductory programming assignments. Given that such assignments are typically simple and well-defined, we show that small, properly quantized LLMs offer competitive repair capabilities suitable for practical deployment. Proper quantization settings help mitigate computational resource constraints and reduce computational costs. We further evaluate ComplementaryRepair over small open-source LLMs with different quantization settings, demonstrating that Q4 quantization generally offers a good balance between repair rate and memory requirements. Notably, ComplementaryRepair provides greater benefits for smaller models than for larger models, enhancing their capability to generate reliable repairs for introductory programming assignments. This makes them appropriate for educational scenarios where computational resources are limited or data privacy is a concern.

References

- [1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.
- [2] Umair Z. Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. Compilation error repair: for the student programs, from the student programs. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training, ICSE-SEET '18*, page 78–87, New York, NY, USA, 2018. Association for Computing Machinery.
- [3] David; Smeaton Azcona. *Alan (2020)*. +5 Million Python & Bash Programming Submissions for 5 Courses & Grades for Computer-Based Exams over 3 academic years.. figshare. Dataset, 2020.
- [4] Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. Tfix: Learning to fix coding errors with a text-to-text transformer. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 780–791. PMLR, 18–24 Jul 2021.
- [5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.
- [6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- [7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.
- [8] Hagit Gabbay and Anat Cohen. Combining llm-generated and test-based feedback in a mooc for programming. In *Proceedings of the Eleventh ACM Conference on Learning @ Scale, L@S '24*, page 177–187, New York, NY, USA, 2024. Association for Computing Machinery.
- [9] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference, 2021.
- [10] Sumit Gulwani. Example-based learning in computer-aided stem education. *Commun. ACM*, 57(8):70–80, August 2014.

- [11] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. Automated clustering and program repair for introductory programming assignments. *SIGPLAN Not.*, 53(4):465–480, June 2018.
- [12] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.
- [13] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024.
- [14] Soneya Binta Hossain, Nan Jiang, Qiang Zhou, Xiaopeng Li, Wen-Hao Chiang, Yingjun Lyu, Hoan Nguyen, and Omer Tripp. A deep dive into large language models for automated bug localization and repair. *Proceedings of the ACM on Software Engineering*, 1(FSE):1471–1493, 2024.
- [15] Yang Hu, Umair Z. Ahmed, Sergey Mehtaev, Ben Leong, and Abhik Roychoudhury. Re-factoring based program repair applied to programming assignments. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, ASE '19*, page 388–398. IEEE Press, 2020.
- [16] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems*, 43(2):1–55, January 2025.
- [17] Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.
- [18] Ryosuke Ishizue, Kazunori Sakamoto, Hironori Washizaki, and Yoshiaki Fukazawa. Improved program repair methods using refactoring with gpt models. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1, SIGCSE 2024*, page 569–575, New York, NY, USA, 2024. Association for Computing Machinery.
- [19] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. Impact of code language models on automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1430–1442. IEEE, 2023.
- [20] Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–30, 2024.

- [21] Renren Jin, Jiangcun Du, Wuwei Huang, Wei Liu, Jian Luan, Bin Wang, and Deyi Xiong. A comprehensive evaluation of quantization strategies for large language models. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 12186–12215, 2024.
- [22] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.
- [23] Sophia D Kolak, Ruben Martins, Claire Le Goues, and Vincent Josua Hellen-doorn. Patch generation with language models: Feasibility and scaling behavior. In *Deep Learning for Code Workshop*, 2022.
- [24] Jiaolong Kong, Mingfei Cheng, Xiaofei Xie, Shangqing Liu, Xiaoning Du, and Qi Guo. Contrastrepair: Enhancing conversation-based automated program repair via contrastive test case pairs, 2024.
- [25] Charles Koutcheme, Nicola Dainese, Sami Sarsa, Juho Leinonen, Arto Hellas, and Paul Denny. Benchmarking educational program repair. *arXiv preprint arXiv:2405.05347*, 2024.
- [26] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics. Doklady*, 10:707–710, 1965.
- [27] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. Structured chain-of-thought prompting for code generation, 2023.
- [28] Mark Liffiton, Brad E Sheese, Jaromir Savelka, and Paul Denny. Codehelp: Using large language models with guardrails for scalable support in programming classes. In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research*, pages 1–11, 2023.
- [29] Evanfiya Logacheva, Arto Hellas, James Prather, Sami Sarsa, and Juho Leinonen. Evaluating contextually personalized programming exercises created with generative ai. In *Proceedings of the 2024 ACM Conference on International Computing Education Research-Volume 1*, pages 95–113, 2024.
- [30] Andy Nguyen, Christopher Piech, Jonathan Huang, and Leonidas Guibas. Codewebs: scalable homework search for massive open online programming courses. In *Proceedings of the 23rd International Conference on World Wide Web, WWW '14*, page 491–502, New York, NY, USA, 2014. Association for Computing Machinery.
- [31] Wendkûuni C Ouédraogo, Kader Kaboré, Haoye Tian, Yewei Song, Anil Koyuncu, Jacques Klein, David Lo, and Tegawendé F Bissyandé. Large-scale, independent and comprehensive study of the power of llms for test case generation. *arXiv preprint arXiv:2407.00225*, 2024.

- [32] Tung Phung, José Cambronero, Sumit Gulwani, Tobias Kohn, Rupak Majumdar, Adish Singla, and Gustavo Soares. Generating high-precision feedback for programming syntax errors using large language models. *arXiv preprint arXiv:2302.04662*, 2023.
- [33] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2023.
- [34] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis, 2020.
- [35] Pablo Rivas, Donald R Schwartz, and Ernesto Quevedo. Bert goes to sql school: Improving automatic grading of sql statements. In *2023 Congress in Computer Science, Computer Engineering, & Applied Computing (CSCE)*, pages 83–90. IEEE, 2023.
- [36] Fernando Vallecillos Ruiz, Anastasiia Grishina, Max Hort, and Leon Moonen. A novel approach for automatic program repair using round-trip translation with large language models, 2024.
- [37] Chandan Singh, Jeevana Priya Inala, Michel Galley, Rich Caruana, and Jianfeng Gao. Rethinking interpretability in the era of large language models, 2024.
- [38] Xiaolong Tian. Evaluating the repair ability of llm under different prompt settings. In *2024 IEEE International Conference on Software Services Engineering (SSE)*, pages 313–322. IEEE, 2024.
- [39] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.
- [40] Shubham Vatsal and Harsh Dubey. A survey of prompt engineering methods in large language models for different nlp tasks, 2024.
- [41] Tianyu Wang, Nianjun Zhou, and Zhixiong Chen. Enhancing computer programming education with llms: A study on effective prompt engineering for python code generation. *arXiv preprint arXiv:2407.05437*, 2024.
- [42] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models, 2023.
- [43] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

- [44] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering, ICSE '18*, page 1–11, New York, NY, USA, 2018. Association for Computing Machinery.
- [45] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1482–1494. IEEE, 2023.
- [46] Chunqiu Steven Xia and Lingming Zhang. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE '22*, page 959–971. ACM, November 2022.
- [47] Chunqiu Steven Xia and Lingming Zhang. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 959–971, 2022.
- [48] Chunqiu Steven Xia and Lingming Zhang. Keep the conversation going: Fixing 162 out of 337 bugs for \$0.42 each using chatgpt. *arXiv preprint arXiv:2304.00385*, 2023.
- [49] Jooyong Yi, Umair Z Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. A feasibility study of using automated program repair for introductory programming assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 740–751, 2017.
- [50] Xin Yin, Chao Ni, Shaohua Wang, Zhenhao Li, Limin Zeng, and Xiaohu Yang. Thinkrepair: Self-directed automated program repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1274–1286, 2024.
- [51] Jialu Zhang, José Pablo Cambroner, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. Pydex: Repairing bugs in introductory python assignments using llms. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):1100–1124, 2024.
- [52] Quanjun Zhang, Chunrong Fang, Yang Xie, YuXiang Ma, Weisong Sun, Yun Yang, and Zhenyu Chen. A systematic literature review on large language models for automated program repair, 2024.
- [53] Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. Automatic chain of thought prompting in large language models, 2022.

- [54] Qianhui Zhao, Fang Liu, Li Zhang, Yang Liu, Zhen Yan, Zhenghao Chen, Yufei Zhou, Jing Jiang, and Ge Li. Peer-aided repairer: Empowering large language models to repair advanced student assignments. *arXiv preprint arXiv:2404.01754*, 2024.
- [55] Yuze Zhao, Zhenya Huang, Yixiao Ma, Rui Li, Kai Zhang, Hao Jiang, Qi Liu, Linbo Zhu, and Yu Su. Repair: Automated program repair with process-based feedback. *arXiv preprint arXiv:2408.11296*, 2024.