On-the-fly Generation-Quality Enhancement of Deep Code Models via Model Collaboration

WEIFENG SUN*, Chongqing University, China
NAIQI HUANG*, Chongqing University, China
MENG YAN†, Chongqing University, China
ZHONGXIN LIU, Zhejiang University, China
HONGYAN LI, Chongqing University, China
YAN LEI, Chongqing University, China
DAVID LO, Singapore Management University, Singapore

The growing prominence of deep code models in automating software engineering tasks is undeniable. However, their deployment encounters significant challenges in *on-the-fly performance enhancement*, which refers to dynamically improving the performance of deep code models during real-time execution. Conventional techniques, such as retraining or fine-tuning, are effective in controlled pre-deployment scenarios but fall short when adapting to on-the-fly adjustments post-deployment. CodeDenoise, a notable on-the-fly performance enhancement technology, leverages uncertainty-based methods to identify misclassified inputs and applies an *input modification strategy* to rectify classification errors. While effective for classification tasks, this approach is inapplicable to generative tasks due to two key challenges: ❶ Uncertainty-based methods are unsuitable for identifying *challenging inputs*, especially in generative tasks with diverse and open-ended outputs. *Challenging inputs* refers to a class of inputs where, due to the inherent complexity of the task or insufficient context in the input samples, the model struggles to generate high-quality outputs. ❷ Input modification strategies cannot be applied to generative tasks, as modifying the input can unpredictably affect the entire sequence of generated outputs. These limitations highlight the need for novel techniques that can enhance the generation quality of deep code models in real-time.

To bridge this gap, we propose CodEn, a framework designed to enhance the generation quality of deployed deep code models through model collaboration and real-time output repair. CodEn employs an ensemble learning approach, integrating multiple generic output quality assessment metrics to identify *challenging inputs*. By combining these diverse metrics, CodEn overcomes the limitations of uncertainty-based methods, making it effective across various generative tasks. Additionally, we introduce an elaborate on-the-fly repair method for the outputs of *challenging inputs*, leveraging a large language model (LLM) and a novel dual-prompt strategy. This strategy utilizes both generation and selection-based prompts to provide potential fixes and employs an adaptive mechanism to select the optimal output. Our experiments, conducted on 12 deep code models across three pre-trained code models, three popular code-related generation tasks, and four datasets, demonstrate the effectiveness of CodEn. For example, in the assertion generation task, CodEn enhances the SAM (Semantic Accuracy Match) of baseline models with improvements ranging from 12.14% to 21.65%. In the bug fixing task, CodEn achieves exact match gains ranging

*Both authors contributed equally to this research.
†corresponding author.

Authors' addresses: Weifeng Sun, weifeng.sun@cqu.edu.cn, Chongqing University, China; Naiqi Huang, npcxh@cqu.edu.cn, Chongqing University, China; Meng Yan, Chongqing University, China, mengy@cqu.edu.cn; Zhongxin Liu, Zhejiang University, China, liu_zx@zju.edu.cn; Hongyan Li, Chongqing University, China, hongyan.li@cqu.edu.cn; Yan Lei, Chongqing University, China, yanlei@cqu.edu.cn; David Lo, Singapore Management University, Singapore, davidlo@smu.edu.sg.

ACM 1557-7392/2025/9-ART
https://doi.org/10.1145/3765752

from 17.51% to 30.64% on TFix dataset. For the code summarization task, CodEn significantly boosts performance across key metrics: BLEU scores improved by 5.72%∼11.79%, ROUGE-L by 4.41%∼7.70%, METEOR by 7.51%∼12.29%, and CIDEr by 8.09%∼15.80%. Besides, we conduct experiments of CodEn on different open-source LLMs and demonstrate that CodEn can still achieve significant improvements.

CCS Concepts: • **Computing methodologies** → **Natural language processing**; *Classification and regression trees*.

Additional Key Words and Phrases: Deep Code Models, In-Context Learning, Model Collaboration

1 INTRODUCTION

Deep code models have notably advanced the field of software engineering, particularly in automating code-related generation tasks such as bug fixing [70, 82, 89], code summarization [27, 80, 88], test assertion generation [63, 84], and code mutants injection [68, 69]. Despite the success of deep code models, they face significant performance challenges once deployed. Prior studies [6] indicate that each GitHub Java project introduces around 56.49 new identifiers per thousand lines of code—identifiers never seen in the training data. Additionally, coding styles and conventions often vary at the file level, reflecting unique organizational practices, project-specific patterns, or even developer preferences (*e.g.,* using *i* instead of *j* for loop variables). Such diverse and previously unseen contexts can overwhelm a frozen model's learned parameters, making it difficult to maintain robust performance across evolving real-world codebases [61].

To address these challenges, we propose focusing on *on-the-fly performance enhancement*, which dynamically improves model outputs post-deployment without extensive retraining. This approach provides several critical benefits for the practical deployment of deep code models: 1) *Sustaining Output Quality*: By enabling real-time adjustments, models consistently generate accurate and relevant outputs even when confronted with novel or unforeseen inputs. 2) *Rapid Adaptation to Dynamic Environments*: Software development contexts are continuously evolving, driven by changing requirements, coding conventions, and external dependencies. On-the-fly performance enhancement allows deep code models to immediately adapt to such dynamic conditions, eliminating the need for frequent and costly retraining cycles. 3) *Resource Efficiency*: Traditional model retraining and fine-tuning demand substantial computational resources and extensive labeled datasets, rendering continuous updates impractical. In contrast, on-the-fly enhancements refine model performance at a lower computational cost, thus offering a more scalable and practical solution for resource-constrained scenarios. 4) *Continuous Reliability Between Major Updates*: Even when periodic updates to the model are planned, substantial intervals between deployments may lead to performance degradation due to distribution shifts. On-the-fly enhancement provides a lightweight mechanism to bridge these gaps, maintaining robust output quality and reducing the risk of user-facing errors or service degradation. This not only minimizes disruptions but also lowers the operational burden of emergency hotfixes or manual patching. By addressing critical operational requirements, including sustained quality, immediate adaptability, resource efficiency, and continuous reliability, on-the-fly performance enhancement represents a promising strategy for deploying deep code models effectively in rapidly changing real-world software environments.

Currently, only one notable work focuses on the on-the-fly performance enhancement of deep code models: CodeDenoise by Tian *et al.* [67]. CodeDenoise improves the classification performance of deep code models through an *input modification strategy*, specifically by denoising inputs. The method identifies potentially misclassified inputs by evaluating model uncertainty, localizes noisy identifiers using an attention-based mechanism, and replaces them with clean counterparts. While this approach effectively enhances classification tasks, its application is limited to such tasks. Generative tasks, however, pose different challenges due to their sequential and open-ended nature. Below, we highlight two major challenges that explain why existing methods designed for classification tasks, such as CodeDenoise, cannot be applied to generative tasks:

Challenges. ① **C1:** ***Existing uncertainty-based methods for identifying challenging inputs are unsuitable for generation tasks.*** Accurately identifying inputs likely to produce low-quality outputs is crucial for effective on-the-fly performance enhancement, especially in the absence of ground-truth. This problem is analogous to the test selection challenge in software testing [33, 64, 75], which aims to identify inputs prone to incorrect predictions. For classification tasks, existing approaches typically measure prediction uncertainty to assess model confidence in predicted labels. Uncertainty quantifies the likelihood of misclassification, where lower confidence (*i.e.,* higher uncertainty) often correlates with erroneous predictions. Consequently, uncertainty-based methods leverage probability distributions or entropy criteria to identify inputs susceptible to misclassification [22, 55, 64]. However, generative tasks introduce distinct challenges that diminish the effectiveness of uncertainty-based metrics. Unlike classification tasks, which associate each input with a single correct label, generative models can produce multiple plausible outputs from the same input, often with similar uncertainty levels [55]. In such contexts, uncertainty does not necessarily indicate error likelihood; instead, it may reflect the model's inherent capacity to generate diverse yet acceptable outputs. Furthermore, the criteria for defining "low-quality outputs" in generative tasks extend beyond correctness to include syntactic validity, semantic alignment, and naturalness. Consequently, an input with low uncertainty scores may nonetheless yield outputs that fail to meet critical generative quality standards. For clarity, we define *low-quality outputs* as those outputs that do not meet specific evaluation thresholds determined by task-specific correctness metrics, semantic alignment scores, or other criteria established by developers. These thresholds can be derived from domain-specific requirements or practical development constraints. Correspondingly, *challenging inputs* refer to inputs that are dynamically identified as likely to result in low-quality outputs under the current model configuration and runtime context, often due to distributional shifts, insufficient context, or task-specific complexity. These inputs pose a fundamental difficulty for generative models, making it challenging to generate outputs that meet correctness and semantic alignment requirements. Accurately identifying these challenging inputs, especially when ground-truth references are unavailable, remains a critical, unresolved challenge for enabling effective on-the-fly performance enhancement in generative model deployment.

② **C2:** ***Existing input modification strategies are unsuitable for generative tasks.*** Classification and generative tasks fundamentally differ in their input-output relationships. In classification tasks, the mapping $f : X \rightarrow Y$ defines a deterministic correspondence between input features and labels within a finite label space Y. As a result, minor, localized input modifications can yield predictable changes in labels without compromising the overall semantic integrity of X [67]. Conversely, generative models rely on a stochastic sequence mapping $f : X \rightarrow P(Y)$, where Y denotes an unbounded token sequence space. The stochastic and autoregressive nature of generative models introduces unique difficulties that render conventional input repair methods inapplicable. Specifically, generative tasks face the following critical challenges for input modification: 1) Autoregressive Dependence. Generative models produce outputs sequentially, with each token depending heavily on previously generated tokens. As a result, even small input perturbations Δx can propagate exponentially through subsequent tokens due to the cumulative nature of the generation process, formalized as $\prod_{t=1}^{T} P(y_t \mid x + \Delta x,\ y_{<t})$. 2) Task Complexity and Model Limitations. Not all low-quality outputs result directly from issues within the input itself. Sometimes, low-quality outputs originate from inherent task complexities or model limitations. In such instances, "denoising" or revising the input provides limited benefit, as the root cause lies in deeper constraints related to the model's capabilities. 3) User Intent and Practical Constraints. In real-world development, the original input (*e.g.,* source code provided by developers) must remain intact to preserve intended functionality and semantics. Modifying these inputs risks disrupting developer workflows, eroding trust in automated tools, and introducing additional manual verification efforts to confirm the modified input remains aligned with the original design intentions. 4) Preservation of Original Context. Inputs typically contain project-specific identifiers, API calls, or dependencies crucial to maintaining compatibility and semantic coherence with existing codebases. Altering

these context-specific tokens can render the generated outputs irrelevant or incompatible. Recent studies [13, 91] has shown that industry practitioners favor interventions at the output level rather than modifying inputs for generative AI systems. For instance, repeated sampling from Codex has been found to be an effective strategy for generating functional solutions to difficult prompts [13]. Building on these insights, we propose a context-aware *output repair strategy*, which refines the generated output rather than modifying the input. This approach ensures that the original input remains unchanged, thereby preserving developer intent while enhancing the quality of the generated results.

Our work. Importantly, generative tasks, such as code summarization and test assertion generation, are common and crucial in numerous real-world applications. Therefore, there is an urgent need for a new method that can on-the-fly improve the generation quality of deep code models. To tackle the challenges outlined above, we present a novel on-the-fly generation-quality enhancement technique for (deployed) deep code models, namely CodEn.

Addressing Challenge C1: We introduce an automated strategy to evaluate the likelihood of challenging inputs without the ground truth. This strategy proposes and harnesses multiple generic output quality assessment metrics to effectively identify challenging inputs. We adopt an ensemble learning approach, using gradient tree boosting technique [24], to amalgamate various individual metrics. For an incoming code snippet, we evaluate its output from three distinct perspectives: 1) *Perplexity*: This metric measures the model's uncertainty and confusion for its output. 2) *Syntax-based Output Similarity*: We compare the generated output with desired outputs from lexically similar *PCM training data*, which consists of the code-task pairs originally used to fine-tune the PCM for each downstream task. 3) *Semantic-based Output Similarity*: This assesses the similarity between the generated output and desired outputs from semantically similar PCM training samples.

Tackling Challenge C2: We leverage the advanced semantic understanding capabilities of large language models (LLMs). Although directly using LLMs to generate outputs for challenging inputs may appear intuitive, such an approach is not universally effective. To illustrate this limitation, we compared ChatGPT and CodeT5 on the TFix dataset [10]: Under in-context learning, ChatGPT achieves an exact match accuracy of 30.41%, which is slightly lower than the 30.94% attained by CodeT5. A similar trend appears with LLaMA-3.1, whose BLEU score (24.74) lags behind that of CodeT5 (33.98). These observations highlight that LLMs, despite their general semantic capacity, do not consistently outperform task-specialized code models when used in a straightforward generation manner. Consistent with recent insights presented by Fan *et al.* [21], we find that LLMs function more effectively when utilized to select the best option from multiple output candidates generated by specialized code models, rather than directly generating solutions from scratch. Specifically, Fan *et al.* [21] demonstrated the effectiveness of leveraging LLMs as test oracles to distinguish and select correct outputs among multiple candidate solutions. Inspired by this insight, we design specialized chain-of-thought prompts tailored to two distinct prompt paradigms: *generation* and *selection*. Subsequently, we selectively leverage the highest-quality predictions from the LLM, integrating the strengths of multiple prompting paradigms to enhance the overall quality of generated outputs.

Usage scenario. CodEn serves as an effective plug-and-play post-processing module, seamlessly integrated with the deployed deep code model (target model) as a system. Specifically, the system processes incoming code snippets and generates corresponding results. When CodEn identifies a code snippet as a potentially challenging input for the target model, it intervenes to provide a corrected output. Otherwise, the system returns the original result produced by the target model. Crucially, CodEn requires no modifications to the parameters of the deep code models, only that the inputs are code-related.

Evaluation. To evaluate the performance of CodEn, we conduct experiments on three popular code-related generation tasks including test assertion generation, code summarization, and automatic bug fixing. Following previous work [28, 29, 54, 73], we integrate our approach with three widely used open-source pre-trained code models: CodeT5 [78], UniXcoder [31], and CodeGen [53]. In total, we fine-tune 12 deep code models on

corresponding datasets to tailor them for specific tasks. Our extensive experiments consistently demonstrate CodEn's effectiveness in enhancing the performance of pre-trained code models across various tasks. For example, in the assertion generation task, CodEn equipped with ChatGPT-3.5-turbo improves the SAM (Semantic Accuracy Match) of baseline models by 12.14% to 21.65%. In the bug fixing task, CodEn achieves accuracy gains ranging from 17.51% to 30.64% on TFix dataset. In the code summarization task, CodEn significantly boosts baseline models' performance across key metrics: BLEU scores improved by 5.72%~11.79%, ROUGE-L by 4.41%~7.70%, METEOR by 7.51%~12.29%, and CIDEr by 8.09%~15.80%. Furthermore, CodEn integrated with open source LLMs (*i.e.,* DeepSeek, Llama3) also shows substantial improvements over baseline models, with performance gains of 10.14%~21.01% in assertion generation and 8.80%~24.17% in bug fixing. **Remarkably, CodEn combines the strengths of deep code models and LLMs.** Compared to ChatGPT-3.5-turbo, when CodeGen is equipped with CodEn, accuracy improves by 16.22% on the bug fixing task, BLEU score by 7.88% on the code summarization task, and SAM by 4.13% on the assertion generation task.

Novelty & Contributions. To sum up, the contributions of this paper are as follows:

(1) **New Idea.** To the best of our knowledge, we are the first to propose an on-the-fly output repair framework to improve the generation quality of deep code models.
(2) **Effective Technique.** We propose a novel technique, CodEn, to realize this idea. CodEn consists of a challenging input identification strategy that integrates multiple output evaluation criteria and a novel dual-prompt repair strategy to guide LLMs in providing potential fixes for low-quality outputs. Additionally, CodEn can adaptively select the most accurate LLM prediction as the final repaired output.
(3) **Extensive study.** We conduct an extensive study on 12 deep code models. Experimental results on three tasks demonstrate that our method can on-the-fly improve the performance of (deployed) deep code models on many downstream tasks.
(4) **Open Science.** To support the open science community, we open-source all the studied datasets, experimental data, and our scripts for follow-up studies [1].

2 MOTIVATION

We utilize a simplified example to illustrate the fundamental principles that drive our approach. This example highlights how CodEn identifies challenging inputs and conducts online repairs to enhance the quality of their outputs.

Figure 1 illustrates the focal method `genNextAvailable` and its associated test prefix. The `genNextAvailable` method identifies the subsequent available network port number, starting from a specified port number called `fromPort`. It first validates that `fromPort` falls within an acceptable range of 1 to 49151. The method then iterates, incrementing the port number by one in each cycle. Upon discovering an available port, the method returns the respective port number. The test prefix, a sequence of statements that configures the focal method to reach a defined state, invokes the focal method with 2000 as the starting port. It is anticipated that the focal method will then search for and return the next available port number greater than or equal to 2000.

Observation 1. Overconfidence of Deep Code Models. Identifying challenging inputs in real-world scenarios, especially without ground truth, is a non-trivial task. To tackle this issue, uncertainty-based evaluation methods, such as perplexity (PPL) [37], are commonly employed. While these metrics provide insight into a model's confidence, they primarily capture its internal doubt regarding its predictions rather than actual output quality. This limitation becomes evident when models exhibit overconfidence: despite generating outputs with low PPL scores (indicating high confidence), their outputs may still fall short of human expectations. As depicted in Figure 1, CodeT5 generates an `assertEquals` assertion with a PPL value of 1.005, nearing the theoretical minimum of 1.0. While grammatically correct, this assertion could fail if port number 2000 is already in use, which is unexpected. In response to this, our insight is that using multiple evaluation metrics can more effectively

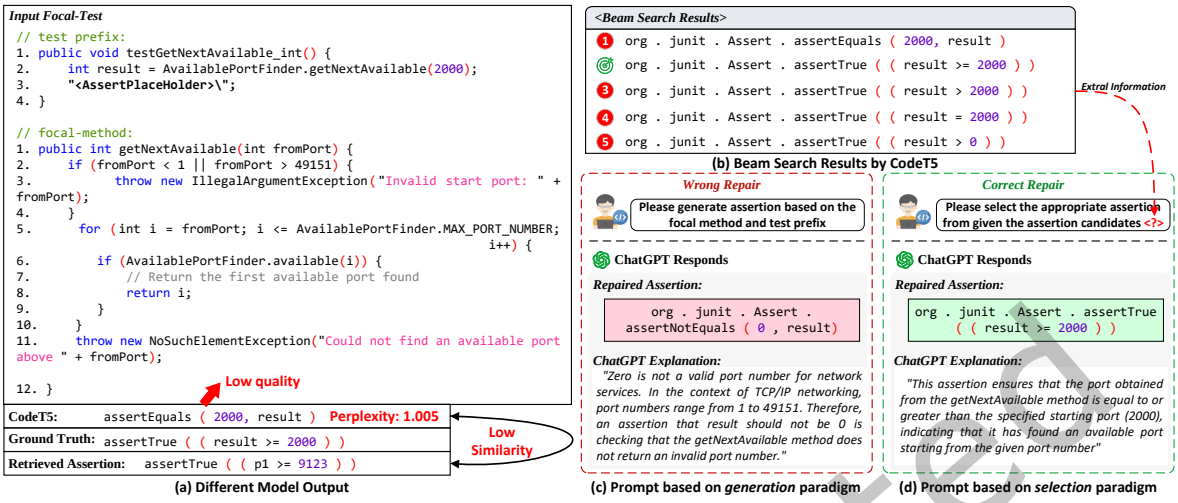


Fig. 1. A simplified example.

identify challenging inputs in generative tasks. Employing a variety of metrics allows for a comprehensive and nuanced assessment by cross-verifying outputs from different evaluative perspectives. For example, in the provided example, we retrieve a training sample similar to the input at the lexical level. By comparing the generated assertion with the ground truth of the retrieved sample (Jaccard similarity of 0.22), we successfully identify the generated assertion as a low-quality output. Considering the diversity of generating tasks, we propose an algorithm for identifying challenging inputs that integrates multiple generic output evaluation metrics through an ensemble learning strategy (detailed in Section 3.2).

Table 1. The experimental results of LLMs with different values of beam size (k) in assertion generation.

Model	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 10$
CodeT5	56.47%	67.86%	71.75%	74.24%	75.98%	80.21%
	-	(↑ 20.16)	(↑ 27.06)	(↑ 31.47)	(↑ 34.56)	(↑ 42.04)
UniXcoder	59.60%	70.85%	74.65%	76.65%	78.16%	81.39%
	-	(↑ 18.88)	(↑ 25.25)	(↑ 28.60)	(↑ 31.14)	(↑ 36.55)
CodeGen	55.14%	65.47%	69.33%	71.60%	73.10%	76.91%
	-	(↑ 18.73)	(↑ 25.73)	(↑ 29.85)	(↑ 32.57)	(↑ 39.48)

↑ denotes performance improvement against the performance at $k = 1$

Observation 2. Challenges in Generating Repaired Results from Scratch using LLMs. Large language models (LLMs) have shown impressive reasoning abilities, significantly outperforming deep code models in understanding natural language and code [9]. Our observations reveal that while LLMs, such as ChatGPT, can accurately comprehend the focal method and test prefix shown in Figure 1, they sometimes struggle to generate applicable assertions from scratch. For example, ChatGPT generates the assertion `assertNotEquals(0, result)`. Although this assertion is syntactically correct, it fails to detect critical logical errors. A specific example of this limitation is this assertion fails to detect functionality errors when a programmer changes line 8 from `return i` to `return i+1`. Additionally, previous research [62] has found that ChatGPT usually generates out-of-distribution outputs that may not align with the requirements of the task.

On one hand, deep code models can provide valuable insights even for challenging inputs. For example, the desired output lies among five candidates generated through beam search, as illustrated in Figure 1(b). Table 1 further demonstrates that increasing the beam size (k) consistently improves the performance of top-k predictions on the ATLAS dataset [79], a benchmark for assertion generation. Leveraging these insights, we adopt a strategy where the LLM is guided to select the most accurate fix from candidates generated by the target model. This approach has proved effective, as the LLM successfully identifies the desired output, as detailed in Figure 1(d). It is possible that there is no expected result in the candidates. In such situation, we can leverage the LLM to generate the output. In this way, our approach uses the LLM in two roles: first as a selector to pick the best candidate from deep code model outputs, and second as a generator to produce an entirely new fix when the available candidates are insufficient. Finally, we introduce an adaptive mechanism (discussed in Section 3.3) that dynamically decides whether to rely on selection or generation to provide the most accurate fix for low-quality outputs.

Attentive readers may wonder why we do not exclusively employ LLMs for all code-related inputs, regardless of their complexity. Our preliminary investigations identify three key reasons that motivate our approach: 1) ***LLMs do not always outperform domain-specialized PCMs***. While LLMs exhibit strong generalization abilities, our experiments show that their performance is not consistently superior to domain-specialized PCMs. For example, on the TFix dataset [10], ChatGPT under in-context learning achieves an exact match accuracy of 30.41%, slightly lower than the 30.94% achieved by CodeT5. Even after parameter-efficient fine-tuning (*e.g.,* LoRA) to adapt Llama-3.1 for bug fixing task, it only achieves 3.91% of exact match, compared to 30.94% for CodeT5. A detailed breakdown of these results is presented later in Tables 5, 7, and 8. 2) ***Unnecessary computational overhead when PCMs succeed***. In cases where smaller, domain-specialized PCMs produce correct or acceptable outputs, invoking an LLM unnecessarily increases inference latency and computational cost. This overhead becomes particularly significant in high-throughput or time-sensitive software development workflows. 3) ***Resource constraints in edge computing environments***. Despite the widespread adoption of deep learning models, real-world deployment scenarios increasingly involve edge computing due to bandwidth limitations, latency considerations, and network instability [92]. Deploying LLMs on edge devices is often impractical due to hardware constraints (*e.g.,* limited GPU memory or CPU resources). Conversely, smaller, domain-specific PCMs can be effectively deployed on such devices, offering low-latency inference for common code tasks without incurring substantial resource costs. Considering these practical constraints, our proposed method strategically integrates the efficiency and domain-specific strengths of PCMs with the advanced reasoning and repair capabilities of LLMs. Specifically, CodEn processes typical or straightforward inputs locally using PCMs—ideal for resource-limited environments—while selectively offloading challenging inputs to an LLM. This hybrid strategy balances computational efficiency with output quality, ensuring rapid inference for routine tasks and leveraging powerful generation and repair abilities when necessary. By uniting the complementary strengths of PCMs and LLMs, CodEn could achieve greater practical applicability than either model could individually.

3 APPROACH

3.1 Overview

We introduce the first technique designed to enhance the generation quality of deep code models on-the-fly. Our approach focuses on identifying challenging inputs and facilitating model collaboration, rather than resorting to retraining or fine-tuning. Figure 2 illustrates the high-level workflow of CodEn. When a new input arrives, the PCM first produces its output in the standard manner. This output is then passed to the challenging input identification module (detailed in Section 3.2), which evaluates its quality using an ensemble of task-agnostic metrics, including perplexity, syntax-based similarity, and semantic-based similarity. Based on these signals, the quality classifier determines whether the input is likely to yield a low-quality output. (1) If the output is predicted to be of acceptable quality, the system directly returns the PCM's original result to the user, ensuring

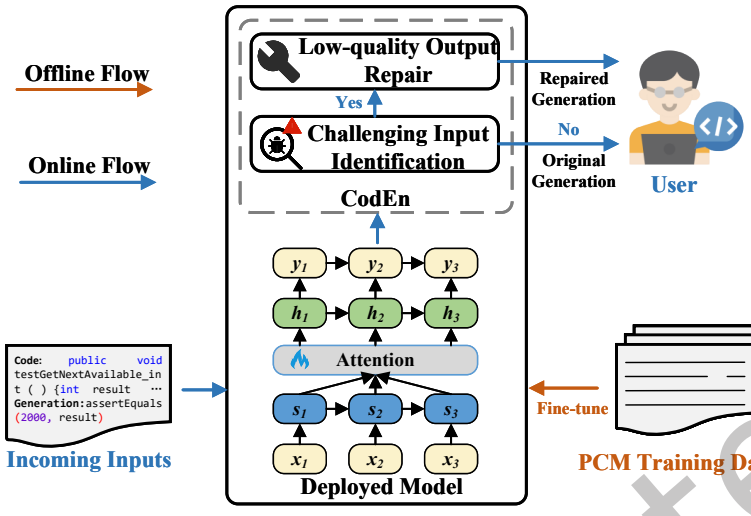


Fig. 2. The overall framework of our approach.

minimal overhead. (2) If the output is flagged as low-quality, the process transitions to the output repair stage (discussed in Section 3.3). In this stage, CodEn leverages the training data of the PCM to retrieve similar examples and constructs prompts under two complementary paradigms: generation and selection. These prompts are provided to a large language model (LLM), which either generates a refined solution from scratch or selects the best candidate among multiple PCM outputs. Finally, an adaptive mechanism compares the repair results across paradigms and chooses the most confident one as the final output. Through this workflow, CodEn operates as a plug-and-play enhancement module: it selectively intervenes only when necessary, preserves the efficiency of PCMs for routine cases, and ensures that challenging inputs receive higher-quality outputs without requiring model retraining or fine-tuning.

To avoid ambiguity, we adopt a consistent notation across tasks. We denote the task input by x, which corresponds to the input provided to the deployed model (*e.g.,* a focal method and test prefix in assertion generation, a code snippet in summarization, or a buggy program in bug fixing). The model's output for this input is denoted as y, and the corresponding reference output is denoted as y_{ref}, representing the developer-provided ground truth (*e.g.,* the correct assertion, summary, or fixed program). At the token level, we use x_t to denote the t-th token of an input sequence x, and y_t to denote the t-th token of an output sequence y. The supervised corpora used to fine-tune PCMs for each downstream task, such as buggy code with developer fixes, code snippets with summaries, or test cases with assertions, are denoted by $\{(x^{(i)}, y_{\text{ref}}^{(i)})\}$. We refer to these corpora collectively as the *PCM training data*, which serve two roles in CodEn: 1) They provide references for computing syntax- and semantic-based quality metrics; 2) During output repair, CodEn retrieves relevant samples to construct few-shot demonstrations, which are then supplied to the LLM to guide the generation of refined outputs.

3.2 Challenging Input Identification

For an incoming code snippet, we introduce an automated strategy to identify whether it is a challenging input by leveraging multiple output quality evaluation metrics. These metrics are consolidated using gradient tree boosting (GTB) [24], an ensemble learning technique. We chose GTB for its robustness and ability to capture complex relationships between features [40, 57]. Specifically, our approach employs three task-agnostic metrics to evaluate output quality. First, the perplexity quantifies the model's uncertainty in generating an output. Next,

the syntax-based output similarity metric measures how closely a newly generated output aligns with ground truth from lexically similar PCM training data. Finally, the semantic-based output similarity metric evaluates output consistency based on the model's hidden states, leveraging semantically relevant examples from the PCM training data.

❶ **Perplexity (PPL):** In this paper, we employ perplexity [37] as a metric to quantify model generation confusion. Perplexity measures the effectiveness of a deep code model in predicting samples and can be used to assess the predictability of the next token or line of code within a given context. A lower perplexity indicates more accurate predictions by the model, reflecting better context comprehension.

$$perplexity\,(x) = \sqrt[N]{\prod_{t=1}^{N} \frac{1}{P\,(x_t|x_{t-1})}} \tag{1}$$

where N is the length of the sequence, x_t is the t-th token in the sequence, and $P(x_t|x_{t-1})$ is the probability assigned by the model to the token x_t given the previous token x_{t-1}.

❷ **SynTax-based Output Similarity (STOS):** Given the common practice of code reuse in software development [39, 41], we propose a retrieval-based approach to help predict the output quality of inputs. We employ the Jaccard similarity coefficient [65] to identify the top-k lexically similar samples $\{x^1, x^2, \ldots, x^k\}$ from the PCM training set and calculate their similarity scores. The Jaccard similarity measures text similarity by calculating the overlap between the sets of words in two texts, defined as $JacSim\,(A, B) = \frac{|A \cap B|}{|A \cup B|}$, where A and B represent sets of words from the two texts. A straightforward method to obtain the top-k lexically similar samples (nearest neighbors) is to compare each input x against all PCM training data and retain the k elements with the highest Jaccard scores. However, this approach entails significant computational overheads, particularly in on-the-fly scenarios. To mitigate this, we sacrifice some accuracy in nearest-neighbor retrieval through the use of approximate nearest-neighbor retrieval. Consequently, CodEn follows a three-step process for each input x.

***Step 1. Retrieval Vector Database Construction*:** In the *offline phase*, we tokenize each sample in the PCM training dataset using the `Javalang` tool [66]. We then calculate the frequency of each word across the dataset, limiting the vocabulary size to the 15,000 most common words. Using the constructed retrieval vocabulary V, we encode each PCM training sample into a one-hot vector $\mathbf{v}_i \in \mathbb{R}^{|V|}$, where $\mathbf{v}_i[j] = 1$ if the j-th word in V is present in the sample, and $\mathbf{v}_i[j] = 0$ otherwise. During the *online phase*, we similarly leverage the vocabulary V to vectorize the input data.

***Step 2. Identifying the Most Similar Subset*:** Next, we identify the subset of samples most similar to the input x, based on the *retrieval vector database* constructed in Step 1. To achieve this, we calculate the intersection and union of the vectors corresponding to x and PCM training samples, thereby deriving the similarity scores. Given that the samples are encoded as binary vectors, we leverage parallelized computation, such as GPU-accelerated retrieval, to compute these scores. Ultimately, we obtain a subset Sim consisting of the 500 most similar PCM training samples.

***Step 3. Determine the Top-k Nearest Neighbors*:** Step 2 aims to expedite the retrieval process by roughly filtering out the least similar samples to x. While this approach reduces computational overhead, it does not provide a comprehensive similarity assessment. Therefore, to achieve accurate nearest neighbor identification, we apply the Jaccard similarity measure on the raw code token to select the top-k elements from the Sim subset.

Notably, we retain the top-k most similar PCM training samples to support the subsequent **Output Repair stage** (see Section 3.3). However, within the STOS criterion, we utilize only the top-1, *i.e.,* the most similar instance. Based on the retrieved most similar instance $x^{(i)}$, we further compute the Jaccard similarity between the generated output y and the reference output $y^{(i)}_{\text{ref}}$ of $x^{(i)}$. A higher similarity score between samples generally indicates a higher likelihood of their corresponding ground truths being closely aligned. Building on this, we incorporate a weighted mechanism that uses the similarity between x and $x^{(i)}$ as a weighting factor to adjust the

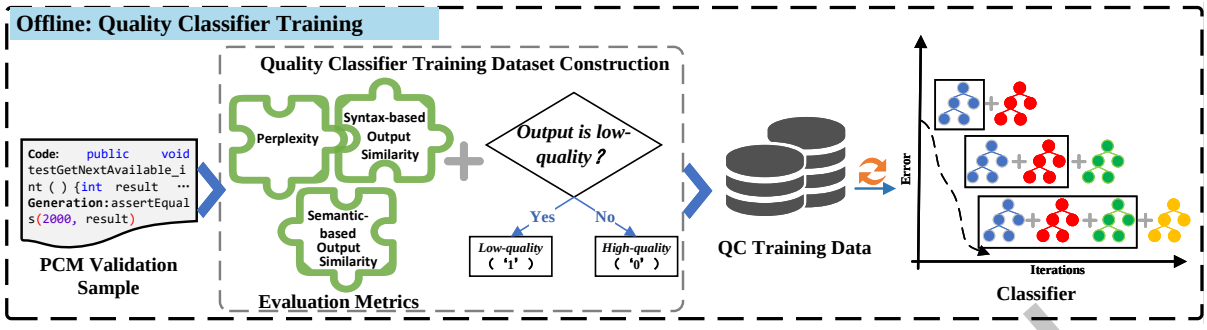


Fig. 3. Overview of challenging input identification.

final output similarity score.

$$\text{STOS}(x) = \text{JacSim}(x, x^{(i)}) \times \text{JacSim}(y, y^{(i)}_{\text{ref}}) \tag{2}$$

❸ **SeMantic-based Output Similarity (SMOS):** Assessing text similarity (*i.e.,* **STOS**) often fails to capture semantic nuances, as code fragments with high semantic similarity might exhibit substantial differences in lexical composition, and vice versa. Therefore, we introduce the semantic-based output similarity metric. In the *offline phase*, CodEn constructs an *embedding database* from the PCM training set. Specifically, CodEn tokenizes the PCM training data and employs the target model to convert these tokens into vector embeddings. During the *online phase*, CodEn processes the input in a same manner. The input is tokenized and then converted into a vector embedding using the same target model. To determine the semantic relevance between the input x and each PCM training sample x^j, CodEn computes the cosine similarity between their respective embeddings: $CosSim\left(x, x^j\right) = \frac{\mathbf{X} \cdot \mathbf{X^j}}{\|\mathbf{X}\| \|\mathbf{X^j}\|}$, where $\mathbf{X}$ and $\mathbf{X^j}$ represent the embeddings of x and x^j, respectively. To accelerate the similarity computation, CodEn leverages vector parallelization, utilizing GPU resources to efficiently compute cosine similarity across the embedding database. For the most semantically PCM similar training data x^s, we utilize the Jaccard similarity criterion to determine the similarity between the generated output y and the reference output $y^{(s)}_{\text{ref}}$ of $x^{(s)}$. Similar to **STOS**, we incorporate the cosine similarity as a weighting factor to adjust the final output similarity score.

$$\text{SMOS}(x) = \text{CosSim}(x, x^{(s)}) \times \text{JacSim}(y, y^{(s)}_{\text{ref}}) \tag{3}$$

Classifier Construction and Prediction. ❶ *Task Requirement.* As shown in Figure 3, our goal is to construct a binary classifier to categorize inputs as high or low quality. The classifier takes three output evaluation metrics as inputs to predict quality labels. ❷ *Quality Classifier Training Dataset Construction.* In the *offline phase*, we create a custom training dataset for the quality classification for each target model, denoated as *QC training data*. QC training data is derived from the validation split of each downstream task. Specifically, for every validation input x, the target PCM generates an output y. We then compute three evaluation metrics: perplexity score, syntax-based output similarity, and semantic-based output similarity, on (x, y) at the sequence level. These metric values are stored as a fixed-length feature vector. We automate the labelling process based on whether the PCM validation sample's generation result meets task-specific generation criteria like correctness and naturalness. Inputs whose outputs satisfy these criteria are labeled as `0`, while others are labeled as 1 (challenging). Each training instance in the QC dataset thus takes the form $(f^{(i)}, \ell^{(i)})$, where $f^{(i)} \in \mathbb{R}^3$ is the feature vector of metric values and $\ell^{(i)} \in \{0, 1\}$ is the quality label. This structured dataset enables the GTB-based classifier to learn the mapping from evaluation features to input quality categories. ❹ *Classifier Prediction.* In the *online phase*, the trained GTB classifier predicts the output quality of incoming inputs. Unlike *PCM training data*, *QC training data* is exclusively used to train the input classifier. Drawn from the model's validation set, this dataset constructs

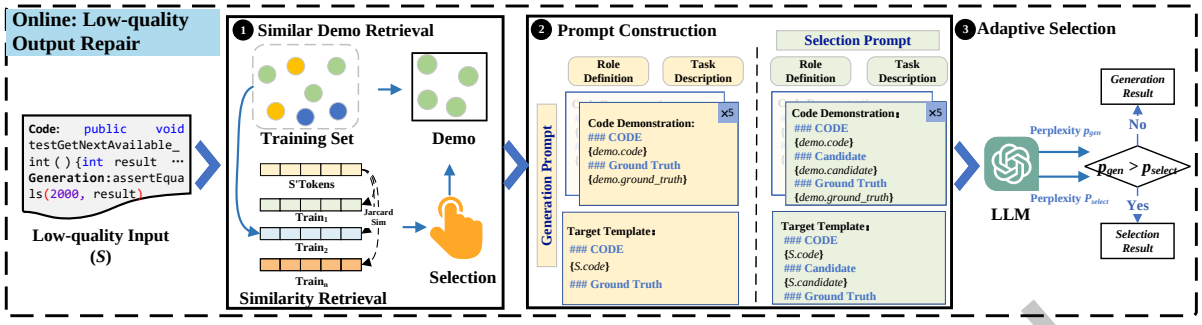


Fig. 4. Overview of low-quality output repair.

input features by computing three evaluation metrics on the target model's outputs, with labels automatically assigned based on task-specific quality criteria. This enables CodEn's classifier to predict whether the output generated from a new input is low-quality, without requiring reference.

3.3 Low-quality Output Repair

In the output repair phase (as shown in Figure 4), CodEn follows a structured approach to repair the low-quality outputs. This phase encompasses three critical tasks: 1) **Similar Demo Retrieval**: CodEn first selects similar samples from the PCM training dataset to serve as demonstrations. These examples guide the subsequent repair process with the LLM. 2) **Prompt Construction**: CodEn formulates chain-of-thought prompts based on the *generation* and *selection* paradigms. These prompts enable interaction with the LLM to produce corresponding outputs. 3) **Adaptive Selection**: Given that the pre-trained GTB classifier requires access to the model's hidden state information, which is not available for closed-source LLMs like ChatGPT, we propose using the *perplexity* metric to select the most accurate LLM predictions as the final enhanced results.

3.3.1 Task1: Similar Demo Retrieval. Recent studies have demonstrated that LLMs can effectively learn within the prompt context to perform new tasks without task-specific fine-tuning. This few-shot prompting is known as *in-context learning* (ICL) [11, 18, 43]. Specifically, when a prompt contains (1) a clear task instruction, (2) a few labeled examples demonstrating task-specific knowledge, and (3) a query from the same task domain, the LLM can infer the correct output by identifying semantic relationships between the examples and the query within a single inference pass. In our approach, **Similar Demo Retrieval** provides the few-shot examples that enable ICL within CodEn. We employ information retrieval techniques [58] to search the top-k most relevant/similar samples from the PCM training dataset that closely align with the target input. These retrieved samples serve as in-context demonstrations, guiding the LLM to understand the expected behavior or output style. Notably, during the **STOS** metric computation, the top-k nearest neighbors have already been retrieved. Therefore, we directly utilize these previously identified neighbors without the need for additional retrieval. Due to the constraints of the LLM's dialogue window, we limit k to the maximum number of samples that fit within these constraints. By integrating relevant examples into the prompt, our approach enables the LLM to learn and adapt dynamically without requiring further fine-tuning. This design maximizes the ICL capability of CodEn: the model observes a small set of reference cases that clarify the task's nature and expected outputs, then applies this understanding to generate or refine new outputs. As a result, **Similar Demo Retrieval** plays a crucial role in our few-shot prompting pipeline, ensuring that CodEn effectively tailors LLM responses to each specific query.

3.3.2 Task2: Prompt Construction. A prompt, denoted as P, is structured as follows:

$$P = \{\mathcal{NL} + \mathcal{CD} + \mathcal{TT}\} \tag{4}$$

where $\mathcal{NL}$ represents a natural language template containing explanatory or instructive text. $\mathcal{NL}$ typically includes the role definition and a detailed task description. $\mathcal{CD}$ comprises a set of code demonstrations, whose composition may vary depending on the employed prompt paradigm. Finally, $\mathcal{TT}$ signifies a developer's query that the LLM must process. In CodEn, we employ two distinct types of prompts based on the *generation* and *selection* paradigms. We will explore the construction of these prompts in the context of the assertion generation task. Detailed prompt construction for other tasks is available in our anonymous replication package [1].

❶ **Generation Prompt**. The generation prompt instructs the LLM to produce results from scratch. Figure 5(a) illustrates an example of the generation prompt in assertion generation task. The composition is as follows:

• *Role Definition*: The generation prompt assigns a specific role to the LLM, typically defined by an instruction such as: `You are a helpful assistant specializing in Java testing with the JUnit framework.`

• *Task Description*: The LLM is then given concrete task instructions: "`For the provided focal method` '`###Focal_method`' `and test prefix` '`###Test_prefix`', `generate the appropriate assertion. Let's think step by step!`". Variables, such as `###Focal_method`, serve as placeholders, which are substituted with specific sample properties during implementation.

• *Code Demonstration*: In Task 1 (as described in Section 3.3.1), we retrieve k similar training examples for each challenging input. These examples are used to create code demonstrations. In the context of the generation prompt, code demonstrations $\mathcal{CD}$ are defined as $\mathcal{CD} = \{\left(x^{(i)}, y^{(i)}_{\text{ref}}\right)\}_{i=1}^{k}$, where $x^{(i)}$ denotes the i-th retrieved PCM training data and $y^{(i)}_{\text{ref}}$ is its associated reference (ground-truth) output.

It's worth noting that, to streamline the extraction of the final repair result and filter out redundant text, we incorporate locator pairs ⟨`Repair`⟩ and ⟨`/Repair`⟩ within our prompts. Specifically, each code demonstration's desired output, $y^{(i)}_{\text{ref}}$, is enclosed by this locator pair, instructing the LLM to generate output within this specified format. This allows CodEn to accurately extract the repaired result using regular expressions while disregarding extraneous text.

• *Target Template*: Recent research [51] has highlighted the importance of aligning the structure of code demonstrations with the target inputs in few-shot learning scenarios to improve performance. Therefore, we meticulously ensure that the encoding of the target inputs is consistent with that used for the code demonstrations in corresponding prompts. Specifically, we provide the input sequence x to the LLM to generate the repaired output.

❷ **Selection Prompt**. As discussed in Section 2, directing LLMs to generate results from scratch does not always yield the expected output. Conversely, in certain scenarios, LLMs prove more effective in repairing low-quality outputs by selecting the best option from multiple candidates generated by target models. Therefore, we propose the selection prompt (as shown in Figure 5(b)) to complement the generation prompt, leveraging their combined strengths. The differences between the selection and the generation prompts are as follows:

• *Role Definition*: The role definition for this prompt is the same as that for the generation prompt.

• *Task Description*: In the selection prompt, the LLM is provided with instructions to choose the best option from a list of candidates: "`For the provided focal method` '`###Focal_method`' `and test prefix` '`###Test_prefix`', `select the most appropriate assertion from the given list` '`###Assertions`'. `Let's think step by step!`".

• *Code Demonstration*: In this context, $\mathcal{CD}$ additionally includes multiple candidates for a given input, denoted as $\mathcal{CD} = \{(x^{(i)}, \{c^{(i)}_1, c^{(i)}_2, \ldots, c^{(i)}_m\}, y^{(i)}_{\text{ref}})\}_{i=1}^{k}$, where $x^{(i)}$ is the i-th retrieved task input, $y^{(i)}_{\text{ref}}$ is its corresponding reference output, and $\{c^{(i)}_1, c^{(i)}_2, \ldots, c^{(i)}_m\}$ is the candidate set of potential outputs produced by the deployed model. To construct the candidate set, CodEn employs beam search [26], a widely used technique for obtaining multiple high-scoring candidates. Subsequently, CodEn applies the following preprocessing steps: 1) *Incorporating ground truth*: Beam search does not always guarantee that the expected results are included in the candidate set. To

```
Role Definition: You are a helpful assistant specializing in Java
testing with the JUnit framework.

Task Description: For the provided focal method '###Focal_method'
and test prefix '###Test_prefix', generate the appropriate
assertion. Let's think step by step!

###Test_prefix:                                        Demonstration 1
getProduction(java.lang.String){
  return productionsByName.get(name);
}
###Focal_method:
testJustifications(){
  runTest("testJustifications",2);
  org.jsoar.kernel.Production j = agent.getProductions()
                                  .getProduction("justification-1");
}
###ASSERTION:
    <Repair>org.junit.Assert.assertNull(j)</Repair>

                                                      Demonstration 2
                                    ...
                                                      Demonstration N

Target Template:
###Test_prefix:
getLogManager(){
  return logManager;
}
###Focal_method:
testLogManagerCreation(){
  org.jsoar.kernel.LogManager logManager=agent.getLogManager();
}
###ASSERTION:
```

(a) An example of generation prompt.

```
Role Definition: You are a helpful assistant specializing in Java
testing with the JUnit framework.

Task Description: For the provided focal method '###Focal_method' and
test prefix '###Test_prefix', select the most appropriate assertion
from the given list '###Assertions'. Let's think step by step!

###Test_prefix:                                        Demonstration 1
getProduction(java.lang.String){
  return productionsByName.get(name);
}
###Focal_method:
testJustifications(){
  runTest("testJustifications",2);
  org.jsoar.kernel.Production j = agent.getProductions()
                                  .getProduction("justification-1");
}
###Assertions:
    org.junit.Assert.assertNotNull(j)
    org.junit.Assert.assertNull(j)
    org.junit.Assert.assertFalse(j)
    org.junit.Assert.assertTrue(j)
    org.junit.Assert.assertNotFalse(j)
###ASSERTION:
    <Repair>org.junit.Assert.assertNull(j)</Repair>

                                                      Demonstration 2
                                    ...
                                                      Demonstration N

Target Template:
###Test_prefix:
getLogManager(){
  return logManager;
}
###Focal_method:
testLogManagerCreation(){
  org.jsoar.kernel.LogManager logManager=agent.getLogManager();
}
###Assertions:
    org.junit.Assert.assertNotNull(logManager)
    org.junit.Assert.assertNull(logManager)
    org.junit.Assert.assertFalse(logManager)
    org.junit.Assert.assertTrue(logManager)
    org.junit.Assert.assertNotFalse(logManager)
###ASSERTION:
```

(b) An example of selection template.

Fig. 5. Assertion repair prompt template.

ensure their presence, we include the ground truth in the candidate set and remove any redundant elements. 2) *Randomizing candidate order*: The order of elements in the candidate set could inadvertently influence the LLM's selection. To counteract this, we randomly shuffle the candidate set, mitigating any positional bias that might affect the LLM's predictions. The above two preprocessing steps are only applicable to Code Demonstration.

- *Target Template*: In the selection prompt paradigm, the candidate set within the target template is confined to the m candidates generated through beam search. While a larger beam size can increase the likelihood of including the desired result (as evidenced in Table 1), it also leads to longer model inference times. To balance effectiveness and computational efficiency, we opt for a beam size of $m = 10$ in our implementation.

3.3.3 Task3: Adaptive Selection. Following the outlined process, we obtain two repaired results from the LLM using generation and selection prompts, denoted as $Repair_{gen}$ and $Repair_{sel}$, respectively. For both, we retain the generation probability for each token. We employ Equation 1 to compute the perplexity of LLM under each prompt paradigm. The output with the lowest perplexity is selected as the final fix, ensuring it consistently aligns with the LLM's highest confidence level.

4 EVALUATION SETUP

4.1 Research Questions

CodEn is a generic framework without specific assumptions regarding the underlying model or task characteristics. To validate the generalizability of CodEn, we investigate its performance across multiple pre-trained code models

and three code-related generation tasks. Our evaluation is structured around the following seven research questions:

RQ1 Effectiveness Evaluation: Can CodEn improve the performance of a target model?
RQ2 Model Agnosticism: Does the effectiveness of CodEn depend on specific large language models?
RQ3 Integration Advantages: Are the performance improvements of CodEn primarily due to its specific integration or the inherent strengths of LLMs?
RQ4 Comparative Evaluation: How does CodEn compare to other techniques designed to enhance model performance in terms of effectiveness and efficiency?
RQ5 Accuracy in Identifying Challenging Inputs: How does CodEn compare to existing test input selection methods in accurately identifying challenging inputs?
RQ6 Component Impact: What is the impact of component within CodEn on its performance?
- **RQ6.1:** *Do all metrics of CodEn contribute to the challenging input identification?*
- **RQ6.2:** *How does the weighting mechanism affect challenging input identification in CodEn?*
- **RQ6.3:** *Does the proposed dual-prompt repair strategy consistently enhance the effectiveness of CodEn?*

RQ7 Hyperparameter Selection: How do different hyperparameters influence the effectiveness of CodEn?

Table 2. Evaluation Metrics for Different Tasks and Datasets

Task	Dataset	Metrics	PCM Training-set	PCM Validation-set	Test-set
Assertion Generation	ATLAS [79]	SAM LCS ED	125,408	15,676	15,676
Code Summarization	JCSD [36]	Standard BLEU ROUGE-L METEOR SentenceBERT CIDEr	69,708	8,714	8,714
Bug Fixing	TFix [10]	Exact Match	74,342	10,504	10,504
	BFP_{small} [69]	BLEU-4	46,680	5,835	5,835

4.2 Evaluation Tasks

We perform experiments on three fundamental code-related generation tasks: assertion generation, code summarization, and bug fixing, aiming to encompass diverse task types, *i.e.,* Code → Text and Code → Code transformations. The downstream tasks and datasets are summarized in Table 2. Due to limitations in space, we offer a comprehensive description of the evaluation metrics and dataset statistics in the provided replication package [1].

4.2.1 Assertion Generation (AG). The assertion generation task aims to automatically produce assertions that effectively validate the behavior of the focal method. Well-crafted assertions not only facilitate faster failure detection but also contribute to improving overall software quality [48, 84].

Datasets. In line with prior studies [48, 84], we adopt the ATLAS dataset [79] as the benchmark for evaluating assertion generation models. The ATLAS dataset is derived from the 2.5 million test methods in GitHub. Each test method within this dataset is associated with a corresponding focal method, meaning the main production code method exercised by the test. Then, the ATLAS dataset is preprocessed to remove test methods exceeding 1k tokens and assertions containing *unknown* tokens [49, 86]. After preprocessing, the dataset is reduced to a total of 156,760 usable data items. Each item in the ALTAS dataset is represented as a triple $\langle T, TM_n, A \rangle$, where T denotes the test case, TM_n is the corresponding focal method, and A is the assertion to be generated. We divide

the dataset into PCM training, PCM validation, and test sets with an 8:1:1 split ratio, which mirrors the original division of the ATLAS dataset. Deep code models take the focal method as well as the test case as inputs to generate relevant assertions.

Metrics. Most previous studies [48, 84] utilize the Exact Match metric. However, our experiments show that this metric is insufficient since some assertions are semantically equivalent despite having different lexical syntax. Therefore, we develop a new metric, *Semantic Match* (SM), which returns a binary value: 1 for an exact semantic match and 0 for no match. Specifically, SM uses matching templates to evaluate assertions that are not lexically identical to the ground truth but are semantically equivalent. The SM compares the generated assertion with the ground truth as follows: 1) *Stripping Full Form*: We identify and retain only the portion of the code that starts with the assertion keyword (*e.g.,* `assert`). 2) *Handling Parentheses*: We normalize any extra parentheses to ensure consistency in syntax structure. 3) *Replacing Equivalent Assertions*: We identify and replace semantically equivalent assertions, such as `assertTrue(a)` with `assertEquals(true, a)` and `assertFalse(a)` with `assertEquals(false, a)`. 4) *Matching Arguments*: We compare the arguments of the assertions to ensure they match semantically, even if their order differs. Based on SM, we further propose the SAM (Semantic Accuracy Match) metric, which determines the percentage of samples with an $SM = 1$ over the evaluation dataset. Moreover, in line with previous work [28, 51], we use the Longest Common Subsequence (LCS) and Edit Distance (ED) as additional evaluation metrics. In this task, we define a output as low quality when its SM with the reference output falls below 1. This definition is adopted for evaluation purposes to measure the performance of PCMs in assertion generation.

4.2.2 Code Summarization (CS). Code summarization generates concise, human-readable descriptions for a given code snippet. This process aids developers in quickly understanding the code functionality, enhancing productivity and code maintainability.

Datasets. We conduct our experiments using the Java Code Summarization Dataset (JCSD) [36], a dataset extensively utilized in current code summarization research [4, 23, 30, 88]. Compiled by Hu *et al.* [36], JCSD comprises 85K pairs of Java methods and their corresponding descriptions, sourced from GitHub. The dataset is divided into PCM training, PCM validation, and testing sets, each containing 69,708, 8,714, and 8,714 pairs, respectively.

Metrics. For code summarization, we adhere to methodologies established in previous research [4, 50] and use four widely accepted metrics: the standard BLEU score [13], ROUGE-L [42], METEOR [8], and CIDEr [71]. However, these measures primarily focus on token overlap, which may not fully capture the semantic accuracy of generated summaries. To overcome the limitations of word overlap–based metrics in handling synonyms and varied expressions, we additionally incorporate a semantic similarity metric [34]. Specifically, we adopt a SentenceBERT-based Cosine Similarity (SBCS) approach, encoding both the generated and reference summaries into vector representations using a pre-trained SentenceBERT model. We then compute the cosine similarity between these vectors. Following [2, 72], we regard a set of generated comments as low quality if their BLEU score is less than 40.

4.2.3 Bug Fixing (BF). Bug fixing endeavors to automate the correction of software errors, thereby significantly reducing manual effort and expediting the debugging process.

Datasets. Following prior work [51, 76], we employ the TFix dataset [10] to evaluate the capability of CodEn in the bug fixing task. TFix [10] is a large-scale program repair dataset consisting of JavaScript code patch pairs extracted from 5.5 million GitHub commits. The dataset covers 52 error types detected by ESLint, comprising a total of 104,804 samples. Each sample includes a buggy code snippet and an associated error context, which provides information about the error type, message, and warning line. Additionally, we use the dataset from Tufano *et al.* [69], which contains buggy Java methods paired with their corresponding fixed versions. Tufano *et al.* mine around 2.3M bug-fix pairs (BFPs). Subsequently, the BFPs are abstracted to minimize source code

identifiers, as detailed in [69]. Tufano *et al.* create two subsets: BFP_{small} and BFP_{medium}. Due to computational resource constraints, we opt for BFP_{small}, which contains a total of 58,350 instances. We adhere to the original data partitioning scheme of BFP_{small}, dividing it into PCM training, PCM validation, and test sets.

Metrics. Following the methodology of prior studies [15, 25, 47, 48, 76, 90], we evaluate the effectiveness of PCMs in bug fixing task, using *Exact Match (EM)*. Exact Match measures the proportion of generated fixes that are identical to the developer-provided patches. We acknowledge that in general automated program repair (APR) research, functional correctness is typically assessed by dynamically validating generated patches against a test suite. However, the benchmarks adopted in this study (*e.g.,* TFix and $\text{BFP}_{\text{small}}$) target ESLint-style or abstracted bug-fix tasks, where executable test cases are not available. Consequently, Exact Match has become the default evaluation criterion in these benchmarks [94]. To provide a complementary measure of repair quality, following existing work [78, 90], we also report the BLEU-4 score, which assesses semantic closeness between generated fixes and reference patches. BLEU-4 provides a more flexible evaluation by measuring subword overlap, whereas exact match imposes a stricter requirement, considering a prediction correct only if it matches the ground-truth patch exactly. In this context, any generated output that does not exactly match the ground truth is considered low-quality.

4.3 Base Models and Baselines

Following previous work [28, 29, 54, 73], we evaluate the performance of CodEn by building it on the top of three popular open-source pre-trained code models, *i.e.,* CodeT5 [78], UniXcoder [31], and CodeGen [52]. We chose CodeGen-350M-Multi for our Java-related evaluation tasks. Importantly, no established baseline exists for on-the-fly generation-quality enhancement scenarios, and CodeDenoise [67] only applies to classification tasks due to its operational mechanics. Therefore, we use the base models as baselines. Future work will extend our evaluations to include these larger models to further validate CodEn's generalizability and effectiveness.

4.4 Implementation Details

We implement CodEn using Python and the PyTorch framework [3], encapsulating the functionality of LLMs through API support (*e.g.,* ChatGPT) or source code availability (*e.g.,* DeepSeek). We default to the gpt-3.5-turbo-0125 model from the ChatGPT family in our experiments. Additionally, we evaluate CodEn with gpt-4o-2024-08-06 to examine improvements over previous versions. To demonstrate the generalization of CodEn, we also consider several other LLMs, particularly open-source models such as DeepSeek-Coder-V2 [93] and Llama3 [19]. For each prompt, we generate results using a sampling temperature of 0 to ensure more deterministic outputs, which is crucial for reproducibility and consistent evaluation. By default, we configure k (the number of code demonstrations) to 5 for the assertion generation and code summarization tasks, and to 15 for the bug fixing task. The beam size m is uniformly set to 10 across all tasks. We further explore the impact of these hyperparameters in Section 5.5. We reproduce the results of all deep code models according to the official repositories published by the model authors. All experiments are conducted on a system running Ubuntu 20.04, equipped with four NVIDIA GeForce RTX 3090 GPUs, a 32-core processor, and 256GB of memory.

5 RESULT ANALYSIS

5.1 RQ1: Effectiveness of CodEn

Motivation and Approach. In this section, we evaluate the effectiveness of CodEn across three code intelligence tasks, including assertion generation, code summarization, and bug fixing. We conduct experiments using the dataset described in Section 4.2 and compare the evaluation results for each downstream task with the baseline models. CodEn employs ChatGPT-3.5-turbo as the backbone model to improve low-quality outputs. Moreover,

we investigate the performance of CodEn using gpt-4o-2024-08-06 to assess its effectiveness with a more advanced LLM variant.

Results. We present experimental results integrating CodEn with three prevalent deep code models—CodeT5, UniXcoder, and CodeGen—across assertion generation, code summarization, and bug fixing tasks. We compare the baseline performance of each model with two variants of CodEn: CodEn$_{GPT3.5}$ and CodEn$_{GPT4o}$. The former leverages ChatGPT-3.5, while the latter employs ChatGPT-4o.

• *Assertion Generation.* As shown in Table 3, incorporating CodEn$_{GPT3.5}$ significantly improves the SAM metric, with CodeT5 increasing from 56.47% to 65.31% (+15.66%) and CodeGen from 55.14% to 67.08% (+21.65%). Similar consistent improvements occur for LCS and ED metrics, confirming CodEn effectively identifies and repairs low-quality outputs. Employing ChatGPT-4o for on-the-fly repairs leads to additional but typically smaller increments, indicating that CodEn$_{GPT4o}$ can further refine certain challenging inputs (*e.g.,* complex or domain-specific assertion patterns) but does not guarantee uniform improvements. The overall trend remains clear: by repairing low-quality outputs, CodEn raises the quality of automatically generated assertions to a level difficult to achieve with the base model alone.

• *Code Summarization.* Table 4 reports the evaluation metrics for the code summarization task. Here, the results differ significantly from the assertion generation scenario. With CodEn$_{GPT3.5}$, all three base models consistently achieve substantial improvements over their original performance. For example, CodeT5 gains +9.33% on BLEU, +6.65% on ROUGE-L, and +3.25% on SBCS. However, when employing CodEn$_{GPT4o}$, the performance improvements become less pronounced and, in some cases, decrease compared to CodEn$_{GPT3.5}$. For example, for CodeT5, improvements in BLEU decrease from 9.33% (CodEn$_{GPT3.5}$) to only 2.77% (CodEn$_{GPT4o}$). Nonetheless, CodEn$_{GPT4o}$ slightly enhances the SBCS metric over CodEn$_{GPT3.5}$ in CodeGen (2.94% → 3.42%). A similar phenomenon occurs with UniXcoder and CodeGen, where CodEn$_{GPT4o}$'s gains are generally smaller than those obtained with CodEn$_{GPT3.5}$. This indicates that ChatGPT-4o does not universally outperform ChatGPT-3.5-turbo on code summarization, highlighting the importance of LLM selection based on the downstream tasks.

• *Bug Fixing.* The results for the bug fixing task are presented in Table 5, covering both the TFix dataset and the BFP$_{small}$ dataset. The BFP$_{small}$ samples used in this experiment are abstracted versions, where source code identifiers are minimized, thus reducing identifier-related noise and complexity. On the TFix dataset, CodEn significantly enhances the Exact Match (EM) performance for all base models when integrated with ChatGPT-3.5-turbo. CodeT5, UniXcoder, and CodeGen achieve Exact Match improvements of 17.51%, 30.64%, and 20.09%, respectively. ChatGPT-4o further elevates Exact Match scores for UniXcoder (+36.99%) and CodeGen (+24.52%), while slightly reducing performance improvement for CodeT5 (+17.26%) compared to ChatGPT-3.5. Moreover, on the BLEU-4 metric, CodEn$_{GPT4o}$ shows diminished or negative improvements (*e.g.,* CodeT5 at -0.87%). In contrast, on the abstracted BFP$_{small}$ dataset, both CodEn$_{GPT3.5}$ and CodEn$_{GPT4o}$ substantially improve performance, yet CodEn$_{GPT3.5}$ consistently delivers notably greater gains. Specifically, CodEn$_{GPT3.5}$ yields striking Exact Match improvements—211.73% (CodeT5), 227.84% (UniXcoder), and 166.82% (CodeGen)—compared to 95.92%, 124.23%, and 73.18% respectively achieved by CodEn$_{GPT4o}$. Additionally, CodEn$_{GPT3.5}$ outperforms CodEn$_{GPT4o}$ in terms of BLEU-4 across all models.

Table 3. Experimental results on assertion generation (AG).

Base Model	SAM			LCS			ED		
	Base	+CodEn$_{GPT3.5}$	+CodEn$_{GPT4o}$	Base	+CodEn$_{GPT3.5}$	+CodEn$_{GPT4o}$	Base	+CodEn$_{GPT3.5}$	+CodEn$_{GPT4o}$
CodeT5	56.47%	65.31% (↑ 15.66%)	66.66% (↑ 18.04%)	79.56%	83.30% (↑ 4.70%)	83.85% (↑ 5.39%)	12.95	10.80 (↑ 16.63%)	10.36 (↑ 20.04%)
UniXcoder	59.60%	66.83% (↑ 12.14%)	68.60% (↑ 15.09%)	81.69%	84.21% (↑ 3.09%)	84.97%(↑ 4.03%)	11.68	9.34 (↑ 20.05%)	8.94 (↑ 23.42%)
CodeGen	55.14%	67.08% (↑ 21.65%)	68.98% (↑ 25.10%)	79.04%	84.12% (↑ 6.43%)	84.92% (↑ 7.43%)	12.99	10.11 (↑ 22.18%)	9.54 (↑ 26.58%)

↑ denotes performance improvement of CodEn against base models

Table 4. Experimental results on code summarization (CS).

	CodeT5			UniXcoder			CodeGen		
Metric	**Base**	+CodEn$_{GPT3.5}$	+CodEn$_{GPT4o}$	**Base**	+CodEn$_{GPT3.5}$	+CodEn$_{GPT4o}$	**Base**	+CodEn$_{GPT3.5}$	+CodEn$_{GPT4o}$
BLEU	33.98	37.15 (↑ 9.33%)	34.92 (↑ 2.77%)	33.49	37.44 (↑ 11.79%)	35.29 (↑ 5.36%)	38.10	40.28 (↑ 5.72%)	39.93 (↑ 4.81%)
ROUGE-L	44.39	47.34 (↑ 6.65%)	45.93 (↑ 3.47%)	43.79	47.16 (↑ 7.70%)	45.88 (↑ 4.78%)	47.93	50.05 (↑ 4.41%)	49.98 (↑ 4.27%)
METEOR	21.30	23.92 (↑ 12.29%)	23.00 (↑ 7.95%)	20.83	23.27 (↑ 11.71%)	22.19 (↑ 6.54%)	23.79	25.57 (↑ 7.51%)	25.96 (↑ 9.15%)
CIDEr	2.75	3.09 (↑ 12.54%)	2.85 (↑ 3.64%)	2.67	3.09 (↑ 15.80%)	2.86 (↑ 7.14%)	3.18	3.44 (↑ 8.09%)	3.39 (↑ 6.61%)
SBCS	69.85	72.12 (↑ 3.25%)	71.59 (↑ 2.49%)	69.82	72.12 (↑ 3.29%)	71.82 (↑ 2.86%)	70.73	72.81 (↑ 2.94%)	73.15 (↑ 3.42%)

↑ denotes performance improvement of CodEn against base models

Table 5. Experimental results on bug fixing (BF).

Dataset	Base Model	Exact Match			BLEU-4		
		Base	+CodEn$_{GPT3.5}$	+CodEn$_{GPT4o}$	Base	+CodEn$_{GPT3.5}$	+CodEn$_{GPT4o}$
TFix	**CodeT5**	30.94%	36.36% (↑ 17.51%)	36.28% (↑ 17.26%)	68.79	71.53 (↑ 3.98%)	68.19 (↓ -0.87%)
	UniXcoder	26.35%	34.42% (↑ 30.64%)	36.10% (↑ 36.99%)	64.74	70.95 (↑ 9.59%)	68.95 (↑ 6.49%)
	CodeGen	29.43%	35.34% (↑ 20.09%)	36.64% (↑ 24.52%)	65.23	70.49 (↑ 8.06%)	67.73 (↑ 3.83%)
BFP_{small}	**CodeT5**	3.36%	10.47% (↑ 211.73%)	6.58% (↑ 95.92%)	57.56	70.86 (↑ 23.11%)	61.64 (↑ 7.10%)
	UniXcoder	3.32%	10.90% (↑ 227.84%)	7.46% (↑ 124.23%)	53.65	68.40 (↑ 27.49%)	63.26 (↑ 17.92%)
	CodeGen	3.77%	10.06% (↑ 166.82%)	6.53% (↑ 73.18%)	56.26	69.97 (↑ 24.37%)	61.27 (↑ 8.92%)

↑ denotes performance improvement of CodEn against base models
↓ denotes performance decrease of CodEn against base models

In Summary. Overall, CodEn demonstrates consistently beneficial across assertion generation, code summarization, and bug fixing tasks, significantly elevating baseline models' performance with both ChatGPT-3.5 and ChatGPT-4o. However, ChatGPT-4o's performance gains are context-sensitive, highlighting that advanced LLM variants are not universally superior. Task characteristics, dataset abstractions, and the desired granularity of outputs play critical roles in determining the optimal choice of LLM integration.

5.2 RQ2: Model Agnosticism of CodEn

Motivation and Approach. In RQ1, we use ChatGPT-3.5-turbo as the backbone model to evaluate the effectiveness of CodEn compared to the baseline model. The experimental results demonstrate that CodEn significantly improves the generation quality of the baseline model, regardless of downstream tasks and metrics. In this RQ, we further investigate the model agnosticism of CodEn to determine whether it remains effective when using different LLMs. Specifically, in addition to ChatGPT-3.5-turbo, we employ two open-source LLMs: (1) DeepSeek-Coder-V2-Lite-Instruct [93] and (2) Meta-Llama-3.1-8B-Instruct [19]. We follow the same experimental setup as in RQ1, ensuring that the number of code demonstrations k and the beam search m settings remain consistent. The

Table 6. Performance Comparison of CodEn with different LLMs.

Base Model	LLM	Task-Metric		
		AG-SAM	**CS-BLEU**	**BF-EM**
CodeT5	CodEn$_{DSeek}$	64.97% (↑ 15.05%)	38.57 (↑ 11.90%)	33.66% (↑ 8.80%)
	CodEn$_{Llama3}$	63.51% (↑ 12.47%)	35.64 (↑ 4.89%)	34.03% (↑ 10.00%)
UniXcoder	CodEn$_{DSeek}$	66.71% (↑ 11.93%)	39.13 (↑ 16.85%)	32.72% (↑ 24.17%)
	CodEn$_{Llama3}$	65.64% (↑ 10.14%)	35.96 (↑ 7.36%)	32.70% (↑ 24.10%)
CodeGen	CodEn$_{DSeek}$	66.73% (↑ 21.01%)	39.87 (↑ 4.65%)	32.98% (↑ 12.07%)
	CodEn$_{Llama3}$	65.16% (↑ 18.16%)	39.46 (↑ 3.56%)	33.72% (↑ 14.59%)

↑ denotes performance improvement of CodEn variants against baseline models.

results of CodEn using these two LLMs are referred to as CodEn$_{DSeek}$ and CodEn$_{Llama3}$, respectively, and are compared to the baseline model. Considering space constraints, we report only key metrics in this RQ: SAM for assertion generation, BLEU for code summarization, and Exact Match (EM) for the bug fixing task. For the Bug Fixing task, we focus exclusively on the TFix dataset in subsequent experiments, as it provides a more diverse range of error types and a larger sample size than BFP_{small} dataset.

Results. Table 6 presents the performance results of CodEn on each downstream task using different backbone LLMs. The results show that CodEn consistently outperforms baseline models across all tasks, demonstrating its effectiveness regardless of the LLM used. For the assertion generation task, CodEn achieves SAM improvements ranging from 10.14% to 21.01%. Specifically, CodEn$_{DSeek}$ achieves the highest SAM improvement of 21.01% on CodeGen, while CodEn$_{Llama3}$ shows a slightly lower improvement, with the highest being 18.16%. In code summarization, BLEU scores improve by up to 16.85% with CodEn$_{DSeek}$ and 7.36% with CodEn$_{Llama3}$. For the bug fixing task, accuracy improves by up to 24.17% with CodEn$_{DSeek}$ and 24.10% with CodEn$_{Llama3}$. Among the two open-source LLMs, CodEn$_{DSeek}$ generally shows higher performance improvements across most tasks compared to CodEn$_{Llama3}$, indicating its strong capability in repairing low-quality outputs. Overall, CodEn is model-agnostic and can effectively boost the generation quality of various deep code models.

5.3 RQ3: Integration Advantages of CodEn

Motivation and Approach. Attentive readers may wonder if the performance improvements of CodEn stem from the significant advantages of LLMs over target models in downstream tasks. In response, we provide the performance of using LLMs alone on the entire test dataset. We devise two baseline types, LLM$_{gen}$ and LLM$_{sel}$, based on generation and selection prompts, respectively. Each prompt is guided by the same code demonstrations used in CodEn. As with RQ2, we report only key metrics on each downstream task.

Table 7. The performance of CodEn *vs.* LLMs. Values highlighted in red indicate the best. All results are under the same in-context learning setting.

Base	LLM	AG	CS	BF	Base	LLM	AG	CS	BF	Base	LLM	AG	CS	BF	p-value (E)
CodeT5	DeepSeek$_{gen}$	64.07%	34.87	30.45%	UniXcoder	DeepSeek$_{gen}$	64.07%	34.87	30.45%	CodeGen	DeepSeek$_{gen}$	64.07%	34.87	30.45%	** (M)
	DeepSeek$_{sel}$	62.07%	36.89	30.40%		DeepSeek$_{sel}$	64.10%	37.95	29.99%		DeepSeek$_{sel}$	62.83%	**40.98**	29.74%	* (S)
	CodEn$_{DSeek}$	**64.97%**	**38.57**	**33.66%**		CodEn$_{DSeek}$	**66.71%**	**39.13**	**32.72%**		CodEn$_{DSeek}$	**66.73%**	39.87	**32.98%**	–
	Llama3$_{gen}$	58.09%	24.74	30.42%		Llama3$_{gen}$	58.09%	24.74	30.42%		Llama3$_{gen}$	58.09%	24.74	30.42%	** (L)
	Llama3$_{sel}$	60.91%	34.63	30.91%		Llama3$_{sel}$	62.10%	34.91	29.46%		Llama3$_{sel}$	61.20%	**39.73**	30.45%	* (S)
	CodEn$_{Llama3}$	**63.51%**	**35.64**	**34.03%**		CodEn$_{Llama3}$	**65.64%**	**35.96**	**32.70%**		CodEn$_{Llama3}$	**65.16%**	39.46	**33.72%**	–
	GPT3.5$_{gen}$	64.42%	**37.34**	30.41%		GPT3.5$_{gen}$	64.42%	37.34	30.41%		GPT3.5$_{gen}$	64.42%	37.34	30.41%	* (S)
	GPT3.5$_{sel}$	59.65%	35.47	32.67%		GPT3.5$_{sel}$	61.81%	35.68	29.94%		GPT3.5t_{sel}	60.41%	**41.13**	30.85%	* (S)
	CodEn$_{GPT3.5}$	**65.31%**	37.15	**36.36%**		CodEn$_{GPT3.5}$	**66.83%**	**37.44**	**34.42%**		CodEn$_{GPT3.5}$	**67.08%**	40.28	**35.34%**	–

[1.] *** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$, – $p > 0.05$. [2.] L, M, S and N represent Large, Medium, Small and Negligible effect size (E) according to Cliff's delta.
[2.] The number of code demonstrations is task-specific: 5-shot for AS and CS, and 15-shot for BF.

Results. Table 7 illustrates how CodEn effectively leverages the strengths of both target models and LLMs, demonstrating superior performance in assertion generation and bug fixing tasks. To confirm the statistical significance of these observed improvements, we perform a Wilcoxon signed-rank test [81] at a 95% significance level. Additionally, the non-parametric Cliff's delta effect size is used to evaluate the magnitude of the difference[1]. The results show that CodEn significantly outperforms all standalone LLM baselines, with p-values indicating statistical significance ($p < 0.05$) across all cases. Effect size analysis further supports this conclusion, with CodEn achieving at least a small (S) effect size or greater compared to LLMs. However, in the code summarization task,

[1]We use the following mapping for the values of the delta that are less than 0.147, between 0.147 and 0.33, between 0.33 and 0.474 and above 0.474 as "*Negligible (N)*", "*Small (S)*", "*Medium (M)*", "*Large (L)*" effect size, respectively [16]

CodEn shows competitive performance but slightly lags behind some standalone LLM variants. For example, $GPT3.5_{gen}$ outperforms $\text{CodEn}_{GPT3.5}$ when integrated with CodeT5. This discrepancy may be attributed to the ability of these standalone LLM variants to further refine outputs for high-quality inputs. Therefore, we further conduct additional statistical tests specifically for the code summarization task. The results show that CodEn still achieves significant improvements over LLM_{gen} with a p-value of 0.006 and a large effect size (Cliff's delta = 0.778). Despite trailing behind LLM_{sel} in some cases, CodEn still maintains a significant performance advantage, with a p-value of 0.048 and a small effect size. Overall, CodEn is particularly beneficial in scenarios where target models excel at handling high-quality samples, allowing LLMs to focus on more challenging samples.

5.4 RQ4: Comparative Evaluation

Motivation and Approach. While CodEn employs targeted collaboration between a lightweight PCM and an LLM to enhance performance, there are alternative strategies to improve the model effectiveness. One widely adopted approach is fine-tuning, where an LLM is retrained on task-specific datasets to internalize domain-specific knowledge. However, fine-tuning is computationally intensive, requiring substantial resources and labeled data, making it impractical for on-the-fly performance improvement after model deployment. Once a model is deployed, any performance enhancement would necessitate additional retraining and redeployment cycles, increasing costs and deployment delays. Another emerging paradigm is agent-based frameworks, in which multiple LLMs collaborate iteratively to refine outputs dynamically. Although adaptable, such iterative interactions often introduce significant inference latency and increased computational overhead. The comparative trade-offs between these approaches remain unclear, particularly in terms of both effectiveness (*e.g.,* output quality) and efficiency (*e.g.,* training overhead and inference cost). Consequently, we investigate whether CodEn can achieve performance comparable to or exceeding that of fine-tuned and agent-based methods while maintaining a lower computational footprint. We conduct a comparative study involving two distinct techniques:

① *Fine-Tuned LLM.* We adopt Parameter-Efficient Fine-Tuning (PEFT), specifically utilizing Low-Rank Adaptation (LoRA) [35], to adapt two LLMs—CodeLlama-7b-hf and Llama-3.1-8B—to our downstream tasks. LoRA selectively updates a minimal subset of model parameters, integrating low-rank matrices into attention modules while freezing most pretrained weights. Following established guidelines [20, 35], we set the low-rank dimension $\gamma = 8$ and the scaling factor $\alpha = 16$ for stability. We refer to these fine-tuned models as $CodeLlama_{lora}$ and $Llama3.1_{lora}$. Notably, we explore the performance of CodEn with Llama-3.1-8B-Instruct in RQ2, so we intentionally chose Llama-3.1-8B here to ensure a comparable baseline for fine-tuning.

① *Agent-Based Collaboration.* We further design an agent-based pipeline inspired by the collaborative paradigm in CodEn, implemented as two specialized agents. The first agent, $Agent_{detect}$, identifies outputs deemed "low-quality" and requiring refinement. The second agent, $Agent_{repair}$, revises outputs flagged by $Agent_{detect}$ using an LLM. Specifically, upon receiving an incoming input, an initial candidate result is generated by an LLM. $Agent_{detect}$ evaluates its quality: acceptable outputs are returned immediately, whereas low-quality outputs trigger iterative revisions by $Agent_{repair}$, capped at five iterations. To maintain consistent evaluation conditions, each agent invocation receives the same number of in-context demonstrations as provided to CodEn. In alignment with RQ2, we explore two LLMs in this agent-based setup: DeepSeek-Coder-V2-Lite-Instruct [93] and Meta-Llama-3.1-8B-Instruct [19], denoted as $DeepSeek_{agent}$ and $Llama3.1_{agent}$, respectively.

In terms of effectiveness, we report the same evaluation metrics as those used in RQ2. For efficiency, we separately measure the training time and inference overhead associated with each performance enhancement paradigm. In particular, since both CodEn and the agent-based method operate without model retraining, we additionally report the total number of token consumed during LLM invocation to quantify the inference cost associated with LLM usage. In this RQ, we instantiate CodEn using CodeT5 as the base PCM and integrate it with DeepSeek-Coder-V2-Lite-Instruct and Meta-Llama-3.1-8B-Instruct for low-quality output repair, respectively.

Table 8. Effectiveness comparison across enhancement paradigms. Values highlighted in red and blue indicate the best and second best.

Method	Task-Metric		
	AG-SAM	CS-BLEU	BF-EM
CodeLlama$_{unfine-tuned}$	0.00%	1.89	0.00%
Llama3.1$_{unfine-tuned}$	0.00%	0.59	0.00%
CodeLlama$_{lora}$	65.17%	18.05	22.90%
Llama3.1$_{lora}$	63.19%	30.94	3.91%
DeepSeek$_{agent}$	44.34%	31.45	16.83%
Llama3.1$_{agent}$	53.55%	16.60	27.58%
CodEn$_{Llama3}$	63.51%	35.64	34.03%
CodEn$_{DSeek}$	64.97%	38.57	33.66%

Effectiveness comparison. Table 8 summarizes the effectiveness of various performance enhancement paradigms, fine-tuned models, agent-based collaboration, and CodEn, across the three downstream tasks. Each task is represented by its core evaluation metric: SAM for assertion generation, BLEU for code summarization, and Exact Match for bug fixing. For completeness, we also include the results of unfine-tuned LLMs, which establish the lower bound of performance. These unfine-tuned models exhibit very limited capability on downstream tasks (*e.g.,* 0.00% Exact Match on bug fixing and near-zero BLEU on code summarization). In contrast, fine-tuning yields substantial improvements across all tasks. ***For the assertion generation task***, the fine-tuned model *CodeLlama$_{lora}$* achieves the highest SAM of 65.17%, closely followed by CodEn$_{DSeek}$ at 64.97%. This indicates that parameter-efficient fine-tuning via LoRA effectively customizes LLMs for assertion generation, leading to optimal semantic alignment. By comparison, agent-based methods show considerably weaker performance: *DeepSeek$_{agent}$* obtains a SAM of only 44.34%, and *Llama*3.1$_{agent}$ reaches 53.55%. Nevertheless, CodEn closely matches the best fine-tuned model's effectiveness without the computational overhead of retraining, highlighting its practical utility. ***In the code summarization task***, CodEn demonstrates clear superiority, outperforming both fine-tuning and agent-based alternatives. Specifically, CodEn$_{DSeek}$ achieves the highest BLEU score of 38.57, substantially surpassing both fine-tuned models, such as *Llama*3.1$_{lora}$ (30.94) and *CodeLlama$_{lora}$* (18.05), as well as the best-performing agent-based method, *DeepSeek$_{agent}$* (31.45). ***In the bug fixing task,*** CodEn maintains its effectiveness advantage, with CodEn$_{Llama3}$ achieving an Exact Match score of 34.03% and closely followed by CodEn$_{DSeek}$ at 33.66%. Both scores notably exceed those obtained through fine-tuning (*CodeLlama$_{lora}$* at 22.90%) and the best agent-based alternative (*Llama*3.1$_{agent}$ at 27.58%). This indicates that CodEn's targeted identification and selective output enhancement strategy consistently provide stronger performance benefits compared to

Table 9. Efficiency comparison across enhancement paradigms.

Method	AG		CS		BF	
	Train	Infer	Train	Infer	Train	Infer
CodeLlama$_{lora}$	66,675.00	0.46	125,874.00	0.32	44,341.00	0.49
Llama3.1$_{lora}$	60,480.00	0.47	136,735.00	0.33	32,217.00	0.38
DeepSeek$_{agent}$	–	3.04	–	0.63	–	8.48
Llama3.1$_{agent}$	–	1.15	–	1.11	–	0.85
CodEn$_{Llama3}$	37.24	0.85	175.26	0.74	34.37	1.13
CodEn$_{DSeek}$	37.24	0.76	175.26	0.69	34.37	1.00

Table 10. Token consumed by CodEn *vs.* Agent-based methods.

Method	AG	CS	BF
DeepSeek$_{agent}$	5,822.92	1,338.82	9,531.08
Llama3.1$_{agent}$	2,538.29	4,497.20	3,160.89
CodEn$_{Llama3}$	2,362.17	2,098.10	5,881.61
CodEn$_{DSeek}$	2,461.09	2,308.34	6,688.85

other enhancement paradigms. Overall, these comparative findings clearly demonstrate that CodEn provides robust and consistently superior performance across diverse downstream tasks. Although fine-tuning can achieve strong performance in narrowly scoped scenarios (*e.g.,* assertion generation), it does not generalize well across tasks and incurs significant retraining overhead. Similarly, while agent-based methods offer dynamic adaptability, their iterative refinement processes may compromise effectiveness and struggle to achieve consistently high performance.

Efficiency comparison. Tables 9 and 10 present detailed efficiency comparison between the three enhancement paradigms. Efficiency is measured in terms of training overhead, inference latency, and token consumption during LLM invocation. ***Regarding training overhead***, fine-tuning methods impose significant computational costs across all tasks. For instance, fine-tuning CodeLlama requires 66,675s for assertion generation, 125,874s for code summarization, and 44,341s for bug fixing. Similarly, fine-tuning Llama3.1 takes substantial training time: 60,480s for assertion generation, 136,735s for code summarization, and 32,217s for bug fixing. Conversely, while CodEn involves training a quality classifier, it employs a simpler and faster gradient tree boosting (GTB) model. This strategy drastically reduces training time to approximately 37 seconds for assertion generation, 175 seconds for code summarization, and 34 seconds for bug fixing, making CodEn significantly more efficient in terms of training cost. ***In terms of inference latency***, fine-tuned models provide the fastest predictions, typically under one second across all tasks. In contrast, agent-based methods exhibit substantially higher latency due to iterative agent invocations, with $DeepSeek_{agent}$ taking up to 8.48 seconds in the bug fixing task. CodEn, on the other hand, demonstrates moderate inference overhead—slightly slower than fine-tuned models but considerably faster than agent-based methods. For example, CodEn_{DSeek} achieves inference times of 0.76s, 0.69s, and 1.00s for assertion generation, code summarization, and bug fixing, respectively. Notably, although CodEn exhibits slightly higher inference latency compared to fine-tuned models, the effectiveness results (as shown in Table 8) reveal that CodEn consistently achieves similar or much better generation quality across tasks. For example, in the code summarization task, CodEn_{DSeek} achieves a BLEU score of 38.57, markedly outperforming the fine-tuned Llama3.1 BLEU of 30.94, despite its slightly higher inference latency. Similarly, in bug fixing, CodEn_{Llama3} attains an Exact Match score of 34.03%, significantly exceeding the 22.90% achieved by the fine-tuned CodeLlama model. ***When examining token consumption during LLM invocation***, CodEn again demonstrates notable efficiency advantages over agent-based methods. Specifically, in the bug fixing task, $DeepSeek_{agent}$ consumes 9,531 tokens, whereas CodEn_{DSeek} uses only 6,688 tokens. A similar pattern emerges in assertion generation, where CodEn_{Llama3} requires 2,362 tokens compared to $DeepSeek_{agent}$'s 5,822 tokens, underscoring significantly reduced computational resource usage.

In summary. The comparative analysis demonstrates that CodEn offers an optimal balance between effectiveness and efficiency. It achieves notably superior or competitive performance compared to fine-tuned methods while incurring significantly lower training costs and maintaining acceptable inference latency. Moreover, CodEn substantially outperforms agent-based methods in terms of inference latency, token consumption, and generation quality, highlighting its efficiency advantage in scenarios where iterative agent invocations would impose prohibitive overheads. These advantages highlight CodEn's practical suitability for real-world deployment scenarios requiring rapid, effective, and resource-conscious on-the-fly model enhancement.

5.5 RQ5: Performance in Identifying Challenging Inputs

Motivation and Approach. Accurately identifying challenging inputs without ground truth is critical for real-time model performance enhancement. Our approach leverages an ensemble learning framework integrating multiple task-agnostic metrics to address the fundamental challenge. This task shares conceptual similarities with test selection [33, 64, 75] in software engineering, where the goal is identifying inputs that trigger failures, and active learning [60] in machine learning, which prioritizes informative yet uncertain samples. Therefore, RQ5

Table 11. The identification performance (%) for each configuration. Values highlighted in red indicate the best.

Dataset	Model	Precision				Recall				F1-score			
		KC	Gini	BADGE	CodEn	KC	Gini	BADGE	CodEn	KC	Gini	BADGE	CodEn
AG	CodeT5	51.15	71.65	70.27	73.38	52.22	73.16	71.74	74.96	51.68	72.40	71.00	74.16
	CodeGen	49.24	66.17	65.24	67.42	66.75	89.69	88.43	91.48	56.68	76.16	75.08	77.62
	UniXcoder	45.82	71.02	69.54	73.22	47.48	73.59	72.05	75.86	46.64	72.29	70.77	74.52
CS	CodeT5	80.98	87.91	86.64	88.48	89.73	97.42	96.01	98.05	85.13	92.42	91.09	93.02
	CodeGen	75.10	90.95	87.16	92.71	75.40	91.31	87.51	93.08	75.25	91.13	87.33	92.89
	UniXcoder	81.31	88.76	87.01	89.39	88.13	96.20	94.31	96.89	84.58	92.33	90.51	92.99
BF	CodeT5	67.20	74.52	73.68	75.01	85.82	95.17	94.09	95.79	75.37	83.59	82.64	84.13
	CodeGen	68.01	78.56	77.75	79.80	79.08	91.35	90.40	92.19	73.13	84.48	83.60	85.55
	UniXcoder	73.55	78.54	77.08	81.41	84.00	89.69	88.02	92.97	78.43	83.75	82.19	86.81
Average	——	65.82	78.68	77.15	80.09	74.29	88.62	86.95	90.14	69.65	83.17	81.58	84.63
Statistical Results	p-value	**	**	**	——	**	**	**	——	**	**	**	——
	cliff's delta	0.56	0.16	0.23	——	0.73	0.28	0.33	——	0.63	0.26	0.28	——
	Effect-size	*L*	*S*	*S*	——	*L*	*S*	*M*	——	*L*	*S*	*S*	——

[1] *** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$, – $p > 0.05$.
[2] *L*, *M*, *S* and *N* represent Large, Medium, Small and Negligible effect size according to Cliff's delta.

aims to compare CodEn with established methods from test selection and active learning in terms of identification performance of challenging inputs, including precision, recall, and F1-score. Given that test selection and active learning methods usually require selecting a predetermined number of samples, we align the evaluation by selecting an equal number of inputs identified by our method as challenging. We compare CodEn with 3 baseline approaches as follows.

① *Active Learning Method.* **K-Center** (KC) [59] selects inputs that maximize coverage of the feature space, effectively choosing samples that are farthest from already chosen points. **BADGE** [7] combines uncertainty and diversity by selecting inputs whose gradients are ambiguous yet spread out in representation space, thus targeting both uncertain and diverse samples.

② *Test Selection Method.* Originally proposed for test input selection in classification tasks, **DeepGini** [22] measures prediction uncertainty based on the distribution of class probabilities. To adapt it for generative tasks, we follow a token-level strategy: for each generated sequence, we compute the DeepGini score for every token based on the model's predicted probability distribution over the vocabulary. Specifically, for a predicted token distribution p, DeepGini is defined as $1 - \sum_i p_i^2$, where p_i denotes the probability of the i-th token. We then take the average DeepGini score across all tokens in the output sequence to represent the overall uncertainty of the generation. Higher averaged scores indicate greater uncertainty, and inputs with the highest scores are selected as challenging candidates.

Results. Table 11 presents a comparative assessment of CodEn against baseline methods KC, DeepGini, and BADGE in identifying challenging inputs, across three key metrics: Precision, Recall, and F1-score. Overall, CodEn consistently outperforms all baseline approaches across three downstream tasks and multiple models (CodeT5, CodeGen, UniXcoder), as highlighted by the top performance values. On average, the Precision, Recall, and F1-score achieved by CodEn are 80.09%, 90.14%, and 84.63%, respectively, substantially surpassing baseline performances. Statistical tests further corroborate the significance of these improvements, with wilcoxon signed-rank tests confirming that the observed differences are significant at the 95% confidence level ($p < 0.01$). Additionally, Cliff's delta effect sizes range predominantly from small to large, reinforcing the practical relevance of these findings. Analyzing individual baselines, DeepGini emerges as the second-best method, displaying relatively stable performance across Precision and F1-score metrics. In contrast, KC consistently underperforms in

Recall and F1-score, indicating limitations inherent to its feature-space coverage strategy in accurately identifying the most challenging inputs. Furthermore, the results suggest that while BADGE effectively balances uncertainty and diversity, it still falls short compared to the ensemble-based framework utilized by CodEn. Although Table 18 shows that CodEn's time overhead remains comparable to or slightly below some baselines, Table 11 presents a clear advantage in identification metrics. In other words, CodEn retains competitive or faster inference time while delivering more precise and robust detection of challenging inputs.

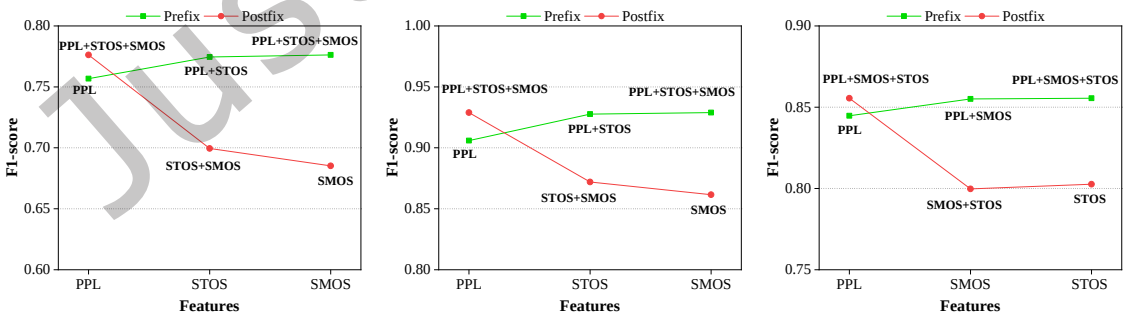
In conclusion, the experimental evidence highlights the effectiveness of CodEn in detecting challenging inputs. Its robust and generalizable performance across different tasks and models positions it as a compelling alternative to existing strategies, with strong potential for test selection and real-time quality assurance fields.

5.6 RQ6: Ablation Experiment

In this section, we explore the contributions of the challenging input identification, the weighting mechanism, and output repairing mechanisms proposed in CodEn. Due to page limitations, we focus on CodeGen as the base model since it shows the best or second-best performance in RQ1.

5.6.1 Impact of Challenging Input Identification. **Motivation and Approach.** We employ the F1-score to evaluate the impact of different criteria on the identification algorithm's performance. We use Recursive Feature Elimination (RFE) [32] in conjunction with gradient tree boosting to rank features. RFE systematically removes the least important features based on the classifier's performance until only one feature remains. The order of removal determines the ranking of feature importance. Given the feature ranking $\{f_1, f_2, ..., f_n\}$ (ordered from most to least important), we train multiple gradient tree boosting models using two strategies: 1) *Prefix strategy* trains with features $\{f_1, f_2, ..., f_i\}$ to evaluate the contribution of the most crucial features. 2) *Postfix strategy* trains with $\{f_i, f_{i+1}, ..., f_n\}$ to assess the contribution of the less significant features. To further understand the individual contribution of each output quality assessment criterion, we conduct an additional ablation experiment. Specifically, we train separate variants of the quality classifier using only one criterion at a time. This allows us to assess the standalone effectiveness of each metric in identifying challenging inputs.

Results. Figure 6 presents the F1-scores for different combinations of quality evaluation criteria across various tasks. The green line signifies the prefix strategy, beginning with the most important feature on the left. For instance, in Figure 6(a), the leftmost point on the green line represents the F1-score obtained using only the PPL criterion. Conversely, the red line represents the postfix strategy, with the rightmost point indicating the F1-score when training involves only the least important feature, in this case, SMOS. The results reveal that incorporating more quality evaluation criteria generally enhances the ability to identify challenging inputs in most tasks. A



(a) Assertion Generation. (b) Code Summarization. (c) Bug Fixing.

Fig. 6. The F1-scores for different training features, with features ranked according to their importance.

Table 12. The F1-score of challenging input identification using individual metric.

Variants	Task		
	AG	CS	BF
PPL	75.67%	90.60%	84.47%
STOS	69.66%	87.08%	80.26%
SMOS	68.52%	86.16%	80.43%
CodEn	**77.62%**	**92.89%**	**85.55%**

comparative analysis underscores the significant contribution of the PPL criterion. The impact of other features varies depending on the task and model characteristics.

We further report the F1-score achieved when using each single metric (PPL, STOS, SMOS) as well as their integrated use within CodEn, as shown in Table 12. PPL typically shows stronger F1-scores than STOS or SMOS, especially in assertion generation and code summarization. For example, in assertion generation, PPL reaches an F1-score of 75.67%, while STOS and SMOS record 69.66% and 68.52%, respectively. Similar trends appear in the code summarization and bug fixing tasks, which suggests that perplexity, as a direct indicator of the model's generation uncertainty, can effectively flag inputs prone to low-quality outputs. Furthermore, combining all metrics outperforms any single metric alone. CodEn integrates PPL, STOS, and SMOS through an ensemble classifier, attaining the highest F1-scores across assertion generation (77.62%), code summarization (92.89%), and bug fixing (85.55%). Compared to PPL alone, the relative improvements are +2.58% in assertion generation, +2.53% in code summarization, and +1.27% in bug fixing. Although these gaps might seem modest in some tasks, they underscore the value of leveraging diverse signals—covering generation uncertainty (PPL), lexical similarity (STOS), and semantic consistency (SMOS). By fusing these perspectives, CodEn detects challenging inputs more robustly than any single criterion, suggesting each metric captures complementary features of low-quality generation. Overall, the results confirm that individual metrics can be effective but incomplete, and that CodEn consistently provides the best coverage in identifying low-quality outputs. This finding aligns with our feature ranking study.

5.6.2 Impact of Weighting Mechanism. **Motivation and Approach.** In CodEn, both STOS and SMOS leverage a weighting mechanism to dynamically adjust the contribution of each retrieved reference based on its similarity to the input. This design emphasizes references closely matching the target input, potentially enhancing the precision of challenging input identification. However, it remains unclear whether this mechanism can improve detection accuracy or merely introduces unnecessary complexity. We thus pose RQ6.2 to investigate how removing this weighting factor affects CodEn's capability in identifying challenging inputs. We compare two configurations in detail: (1) $STOS_{weighted}$, the default setting in CodEn, where the similarity of the input and its neighbor ($\text{JacSim}(x, x_i)$) weights the output similarity ($\text{JacSim}(y, y_i)$); and (2) $STOS_{no\_weight}$, a simplified variant that omits this weighting factor, thus relying solely on output similarity $\text{JacSim}(y, y_i)$. We then evaluate each version in terms of precision, recall, and F1-score for challenging input detection across different tasks. Although SMOS uses a similar mechanism, we focus our detailed analysis on STOS to avoid repetition, given both exhibit consistent trends.

Results. Table 13 summarizes the performance of $STOS_{no\_weight}$ and $STOS_{weighted}$ across the three evaluated tasks. The results reveal minimal differences between the two variants across all tasks. For instance, the F1-scores for assertion generation are nearly identical at 77.60% (*no_weight*) versus 77.62% (*weighted*). A similar trend is evident in the code summarization task (92.79% vs. 92.89%) and the bug fixing task (85.41% vs. 85.55%). Despite these modest differences, the weighted version consistently achieves slightly higher F1-scores.

Table 13. Impact of the weighting mechanism on STOS across three tasks.

	$STOS_{no\_weight}$			$STOS_{weighted}$		
Task	**Prec.**	**Rec.**	**F1**	**Prec.**	**Rec.**	**F1**
AG	67.18	91.84	77.60	67.42	91.48	77.62
CS	92.71	92.87	92.79	92.71	93.08	92.89
BF	79.51	92.26	85.41	79.80	92.19	85.55

Overall, these findings indicate that the weighting mechanism does not substantially alter CodEn's performance. Nonetheless, the consistent improvements provided by the weighting strategy may justify the minor computational overhead required to calculate and maintain input-output similarity scores. Consequently, adopting the weighted variant may still be beneficial in scenarios where precision in challenging input identification is particularly critical.

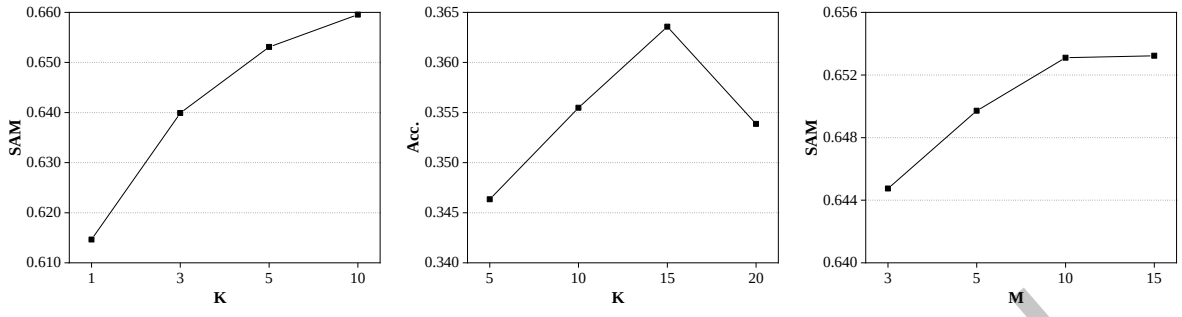
5.6.3 Impact of Low-quality Output Repair. **Motivation and Approach.** We explore the proposed adaptive selection mechanism that combines the generation and selection prompt paradigms. We compare CodEn$_{GPT3.5}$ against three of its variants: 1) CodEn$_{GPT3.5}$-w/o sel: This variant repairs low-quality outputs solely using the generation prompt. 2) CodEn$_{GPT3.5}$-w/o gen: Conversely, this variant relies exclusively on the selection prompt. 3) CodEn$_{GPT3.5}$-w/o adaptive: This variant does not employ the adaptive selection strategy; instead, it randomly selects between outputs generated by the generation and selection prompts as the final repaired result.

Table 14. The effectiveness of prompt paradigms of CodEn

Variants	**Task-Metric**		
	AG-SAM	**CS-BLEU**	**BF-EM**
CodEn$_{GPT3.5}$-w/o sel	66.05%	**40.31**	32.45%
CodEn$_{GPT3.5}$-w/o gen	61.78%	39.77	32.41%
CodEn$_{GPT3.5}$-w/o adaptive	64.99%	39.80	30.83%
CodEn$_{GPT3.5}$	**67.08%**	40.28	**35.34%**

Results. Table 14 compares the performance of our complete framework, CodEn$_{GPT3.5}$, with three ablated variants that remove either the selection prompt (*-w/o sel*), the generation prompt (*-w/o gen*), or the adaptive selection strategy (*-w/o adaptive*). For assertion generation task, our full approach achieves the highest SAM score at 67.08%, outperforming both the single-prompt variants (*-w/o sel* at 66.05% and *-w/o gen* at 61.78%), as well as the random-selection variant (*-w/o adaptive* at 64.99%). A similar pattern emerges for bug fixing, where the full framework attains 35.34% of exact match, substantially above all ablated versions. These gains highlight the synergy derived from enabling both prompts and adaptively choosing the better repair result rather than randomly selecting it.

For code summarization, CodEn$_{GPT3.5}$ delivers a BLEU score of 40.28, which is close to the single-prompt (*-w/o sel*) variant at 40.31. While the difference is small, additional analysis indicates that the generation prompt tends to produce longer comments compared to those generated from the selection prompt, potentially resulting in higher perplexity scores. This discrepancy may bias CodEn towards predictions from the selection prompt. Despite that bias, we find the generation prompt is still more effective for a subset of challenging inputs, thereby justifying the need for a mechanism that adaptively chooses between generation and selection results. Overall, the combination of generation and selection prompts consistently enhances the performance of CodEn across



(a) The exploration of k setting in AG. (b) The exploration of k setting in BF. (c) The exploration of m setting in AG.

Fig. 7. Parameter analysis on k and m.

different tasks. However, the relative importance of each prompt type can vary depending on the task. The adaptive selection effectively balances these prompts, leading to optimal overall performance.

5.7 RQ7: Parameter Analysis

5.7.1 Selection of k and m. **Motivation and Approach.** We investigate the impact of two hyperparameters in CodEn: k and m. By default, k (the number of demonstrations) is set to 5 for the assertion generation and code summarization tasks, 15 for the bug fixing task, and m (beam size) is uniformly set to 10. Due to page limitations and the fact that our initial experiments on the code summarization task show trends similar to those observed in the assertion generation task, we focus our detailed analysis on the assertion generation and bug fixing tasks. Specifically, we present the impact of varying k and m on CodeT5's performance during assertion generation, as well as the results for different k values in the bug fixing task. We experiment with $k = \{1, 3, 5, 10\}$ and $m = \{3, 5, 10, 15\}$ for the assertion generation task and $k = \{5, 10, 15, 20\}$ for the bug fixing task. When adjusting one hyperparameter, the other is maintained at its default setting. We use ChatGPT-3.5-turbo as the backbone model of CodEn.

Results. The experimental results illustrated in Figure 7 display the impact of the hyperparameters k and m on CodEn's performance in the assertion generation and bug fixing tasks. For the assertion generation task, the data indicates a positive correlation between increasing values of k and m and the SAM metric. Specifically, the sharp increase in SAM from 1 to 5 suggests that additional demonstrations provide more context and diverse perspectives, significantly benefiting CodEn in generating more accurate assertions. Similarly, increasing m from 3 to 15 leads to an upward trend in SAM, indicating that a broader search space facilitates better selection of the optimal output. For the bug fixing task, the exploration of different k values reveals a peak performance at $k = 15$. However, a further increase to $k = 20$ results in a slight decline in accuracy, which indicates that beyond a certain point, additional demonstrations may introduce noise or unnecessary complexity that could hinder performance. Although larger values of k and m could further enhance performance in some cases, they also increase computational demands and processing time. Therefore, we balance performance with efficiency by opting for $k = 5$ and $m = 10$ for the assertion generation and code summarization tasks and $k = 15$ for the bug fixing task.

5.7.2 Selection of Size of Most Similar Subset. **Motivation and Approach.** In our three-step retrieval process within STOS, we approximate the top-k neighbors by initially selecting the 500 most similar PCM training samples as a candidate subset. While this default choice offers a practical balance between efficiency and accuracy, the optimal subset size remains an open hyperparameter question. On one hand, a smaller candidate set could further

expedite retrieval but risk omitting critical neighbors, thereby reducing accuracy. Conversely, enlarging the candidate subset might capture more relevant samples but increase computational overhead, adversely affecting runtime efficiency. Thus, we systematically explore the impact of different subset sizes {20, 100, 500, 1000} on both retrieval accuracy and runtime performance. To quantify the effects of subset size, we compare each approximate retrieval variant against an exact nearest-neighbor baseline (denoted as *Exact-NN*) that exhaustively computes the top-5 neighbors for each test sample using Jaccard similarity. For each subset size, we select the top-5 approximate neighbors from the candidate subset based on their Jaccard scores. We then evaluate retrieval accuracy by calculating the overlap ratio between the approximate neighbors and the exact top-5 set. Additionally, we measure retrieval times for each subset size to characterize their computational efficiency.

Table 15. Impact of candidate subset size on retrieval accuracy in STOS.

Size	Task		
	AG	CS	BF
20	73.79%	63.95%	26.49%
100	87.85%	79.42%	40.48%
1000	90.79%	81.87%	47.23%
500	90.49%	81.64%	46.00%

Table 16. Impact of candidate subset size on retrieval efficiency (s) in STOS.

Size	Task		
	AG	CS	BF
Exact-NN	17193.57	5376.96	4493.13
20	1687.19	708.68	601.84
100	1774.84	724.50	604.60
1000	2295.79	907.86	666.19
500	2025.43	824.33	629.41

Results. Tables 15 and 16 summarize the trade-off between retrieval accuracy and efficiency as we vary the candidate subset size {20, 100, 500, 1000}, along with an exact nearest-neighbor retrieval baseline (*Exact-NN*). Two main observations emerge. First, increasing the subset size substantially improves retrieval accuracy across all tasks. For instance, in assertion generation, the overlap ratio jumps from 73.79% at size 20 to 87.85% at size 100, and further reaches around 90% when using 500 or 1000 candidates. A similar pattern appears in bug fixing, where the overlap grows from 26.49% at size 20 to roughly 46% or more at sizes 500 and 1000. In each case, expanding the subset allows the system to retain more relevant neighbors, thus increasing the chance of matching the exact top-5 nearest neighbors. Notably, the difference between 500 and 1000 candidates is relatively small (0.30% in assertion generation and 1.23% in bug fixing), indicating that beyond a certain threshold, further enlarging the candidate set yields diminishing returns. Second, a larger subset comes at higher computational cost but remains far more efficient than exhaustive retrieval. For AG, *Exact-NN* requires around 17K seconds, whereas approximate retrieval with 500 or 1000 candidates takes about 2K seconds—an order of magnitude improvement. Similar trends hold for code summarization and bug fixing. Although size 20 yields the shortest runtime, its accuracy drops substantially (*e.g.,* to 73.79% for AG and 26.49% for bug fixing). Meanwhile, moving from size 500 to size 1000 raises runtime from 2025.43s to 2295.79s in assertion generation (an increase of about 13%), yet the corresponding accuracy gain is marginal (from 90.49% to 90.79%). Thus, 500 candidates strikes a reasonable balance, preserving near-optimal accuracy while containing retrieval overhead.

5.7.3 Selection of Classification Algorithm. **Motivation and Approach.** In our framework, a quality classifier (Section 3.2) predicts whether a given input is likely to yield low-quality outputs, even without ground truth. We initially adopt a gradient tree boosting (GBDT) model to learn these predictions. However, it remains unclear how GBDT compares with alternative classification methods—such as SVM, Logistic Regression (LR), or Random Forest (RF)—under the same training conditions. To address this gap, we investigate whether different algorithms can achieve higher performance in identifying challenging inputs. We re-train the quality classifier on the same labeled dataset (QC training set) using four different algorithms: GBDT, SVM, LR, and RF. Each model receives

Table 17. F1-score (%) of Quality Classifier using Different Algorithms across Tasks.

Model	AG				CS				BF			
	SVM	LR	RF	GBDT	SVM	LR	RF	GBDT	SVM	LR	RF	GBDT
CodeT5	70.43%	68.83%	71.29%	**74.16%**	**93.18%**	92.88%	91.50%	93.02%	80.53%	78.73%	83.58%	**84.13%**
CodeGen	**77.68%**	73.27%	76.05%	77.62%	92.80%	92.06%	91.86%	**92.89%**	81.86%	80.77%	84.57%	**85.55%**
Unixcoder	73.11%	71.86%	72.10%	**74.52%**	92.94%	92.48%	91.36%	**92.99%**	83.36%	82.49%	85.80%	**86.81%**

Table 18. Average Inference time (s) of CodEn variants with different input selection strategies.

Method	AG	CS	BF
CodEn$_{kc}$ + DeepSeek	0.75	0.66	0.99
CodEn$_{kc}$ + Llama3	0.84	0.71	1.12
CodEn$_{gini}$ + DeepSeek	0.70	0.63	0.99
CodEn$_{gini}$ + Llama3	0.80	0.67	1.12
CodEn$_{badge}$ + DeepSeek	0.77	0.68	1.07
CodEn$_{badge}$ + Llama3	0.86	0.73	1.20
CodEn + DeepSeek	0.76	0.69	1.00
CodEn + Llama3	0.85	0.74	1.13

the same feature set, *i.e.,* perplexity, syntax and semantic similarities. We then evaluate every trained classifier in terms of F1-score on the test set.

Results. Table 17 compares the F1-scores of four classification algorithms in identifying challenging inputs across three code models (CodeT5, CodeGen, and UniXcoder) and three tasks (assertion generation, code summarization, bug fixing). Overall, GBDT generally yields the highest F1-scores across most tasks and models. For instance, in bug fixing, GBDT achieves 84.13% with CodeT5, 85.55% with CodeGen, and 86.81% with UniXcoder, consistently outperforming the other classification algorithms. A similar pattern is evident in assertion generation, where GBDT attains around 74.16% and 74.52% for CodeT5 and UniXcoder, respectively, marginally higher than competing methods. Finally, in code summarization, GBDT again leads each baseline over most models. These consistent gains suggest that GBDT's combination of robust feature handling and flexible ensemble structure is well-suited for detecting inputs likely to trigger low-quality outputs.

6 DISCUSSION

6.1 Time Overhead of CodEn Under Different Test Selection and Active Learning Methods

Motivation and Approach. In this section, we aim to understand how integrating different test selection or active learning strategies affects inference time. Specifically, we investigate whether CodEn's overall runtime changes significantly when we adopt DeepGini, K-Center, or BADGE, each representing a distinct approach to identifying challenging inputs. We construct three variants of CodEn, namely CodEn$_{gini}$, CodEn$_{kc}$, and CodEn$_{badge}$, by replacing CodEn's default challenging input classifier with one of these established approaches. Given the potential network latency from ChatGPT API calls, each variant is paired with either DeepSeek or Llama3 to repair low-quality outputs generated by CodeT5. We then measure and compare the average inference time per standardized test sample for each variant, including both the time taken to identify challenging inputs and the additional overhead introduced by LLM-based repairs.

From the Table 18, we can obtain the following observations: 1) Minor variation in the inference time. All variants remain under or around 1.0s for code summarization and assertion generation, and below 1.3s for bug fixing—well beneath the 2s threshold commonly considered acceptable [2] for an interactive setting [45]. Notably, the default classifier shows negligible overhead differences ($\leq$ 0.09s) compared to these advanced test selection or active learning methods. The primary time overhead of CodEn arises from identifying the nearest neighbors for each input. To mitigate this, we use an approximate nearest neighbor search strategy for SMOS metric, leveraging vectorization and GPU-accelerated parallel computation. The nearest neighbors identified by SMOS are reused for subsequent code demonstrations, avoiding a secondary search. This small latency gap reinforces that our proposed method delivers higher F1-scores in challenging input detection (Section 5.5) without imposing a significant penalty on inference speed.

Overall, these findings confirm that our proposed challenging-inputs classifier does not significantly increase inference time relative to other methods. Given the substantial performance improvements delivered by CodEn, we recommend it as an effective method to dynamically enhance generation quality. Moreover, we suggest using the smallest model in the family as the backbone to further minimize any additional overhead from LLM-based repairs, thereby balancing efficiency and generation quality in real-world workflows.

6.2 What Makes CodEn Work?

CodEn can identify challenging inputs. One of CodEn's key strengths lies in its adeptness at identifying challenging inputs. This is achieved through a meticulous process that combines multiple output evaluation criteria. To validate this, we conduct an analysis on the ATLAS test set, computing the edit distances between the results generated by CodeT5 and the ground truth outputs. The average edit distance across all generated outputs is 12.95, notably lower than the average distance of the dataset selected by CodEn, which stands at 23.10. Similar trends are observed in the code summarization results, with CodEn exhibiting an average edit distance of 69.45 compared to the average of 56.66 for all generated outputs.

```
// Code from the test set:
public boolean exportPkcs8(){
  return jrbPkcs8.isSelected();
}
```

```
Ground truth summary:
Has the user chosen to export to PKCS #8?
Summary generated by CodeT5:
Export PKCS #8 data?
Summary generated based on Selection Prompt (PPL:1.02):
Has the user chosen to export PKCS #8 data?
Summary generated based on Generation Prompt(PPL:1.26):
Returns whether PKCS8 should be exported
```

(a) Final repaired output based on selection prompt

```
// Code from the test set:
void onSendTaskCompleted(RecordBuffer<R> buffer){
  logger.trace("{}:{} Send Completed",name(),buffer);
  activeSendTasks.decrementAndGet();
}
```

```
Ground truth summary:
This method should not raise any exceptions.
Summary generated by CodeT5:
Called when a send task has completed
Summary generated based on Selection Prompt (PPL:1.05):
Called when a send task has completed.
Summary generated based on Generation Prompt(PPL:1.01):
This method should not raise any exceptions.
```

(b) Final repaired output based on generation prompt

Fig. 8. Case studies on the code summarization task.

CodEn can better repair low-quality output based on a dual-prompt strategy To further assess the practical implications of CodEn, we conduct an evaluation centered on the assertion generation task, which is crucial to improve the efficiency of the software validation and debugging processes. In this evaluation, we randomly select 100 test samples from the ATLAS dataset. For each sample, we compare three sets of generated assertions: (1) those produced by the base model, (2) those generated by CodEn, and (3) those generated by Meta-Llama-3.1-8B-Instruct in a zero-shot setting. To mitigate potential subjectivity bias, we anonymize the sources of

[2]https://www.websitebuilderexpert.com/building-websites/website-load-time-statistics/

the generated assertions and randomly shuffle the three sets before presenting them to evaluators. It is important to note that, in some cases, CodEn's output aligns with that of the base model, indicating that no modifications are necessary. To ensure a focused evaluation of CodEn's effectiveness, we specifically select samples where its outputs differ from those of the base model. The evaluation is conducted by seven volunteers with more than three years of Java programming experience, comprising three Ph.D. student and four master's students. Each evaluator rates the quality of the generated assertions based on their accuracy and alignment with the intended test sample. The key evaluation question posed is: *To what extent does the generated assertion accurately match the test sample?* Ratings are provided on a 5-point scale, where 1 indicates the lowest accuracy and 5 represents the highest accuracy. On average, each evaluator spends approximately 34 minutes completing the evaluation. The results reveal notable differences in performance across the three methods. The base model achieves an average score of 3.18 out of 5, reflecting moderate alignment with the intended behavior, though requiring manual corrections in some cases. CodEn-generated assertions receive an average score of 3.91, demonstrating significant improvements in both accuracy and relevance. In contrast, the zero-shot results from LLM score an average of 2.07, indicating that while the model occasionally generates reasonable assertions, they are often less precise and domain-specific than those produced by CodEn. The findings offer key insights from a user-centric perspective. First, CodEn consistently outperforms the base model, particularly in terms of the semantic alignment of generated assertions with expected behaviors. Additionally, CodEn shows substantial improvements over the zero-shot LLM, particularly in scenarios requiring domain-specific knowledge that the LLM struggles to capture accurately. From a usability standpoint, participants appreciate the clarity and relevance of CodEn-generated assertions, which reduce the time required for manual validation during debugging. Specifically, the average time spent per assertion is 12% lower for CodEn compared to the base model, primarily due to its higher accuracy and relevance. Furthermore, participants indicate that CodEn's results provide a more reliable foundation for further test case refinement, ultimately reducing debugging time and improving developer satisfaction.

The effectiveness of CodEn lies in its adaptive use of both generation and selection prompts to repair low-quality outputs. Figure 8 presents two examples that illustrate the different advantages of the selection and generation prompts. The selection prompt utilizes information provided by the target model, enabling it to generate project-specific identifiers, such as **`PKCS #8`** in Figure 8(a). However, the selection prompt can only return incorrect output when the expected result is not in the candidate set, as shown in Figure 8(b). In such cases, the generation prompt becomes essential to produce accurate results. By combining these two prompts, CodEn ensures robust performance across various tasks, effectively balancing the generation of new content with the selection of the best existing options. This dual-prompt strategy is key to enhancing the overall quality and reliability of the outputs produced by CodEn.

6.3 Potential Insights to the Research Community

Beyond enhancing performance metrics, CodEn provides valuable implications and insights for both academic research and practical applications. We discuss these contributions across four key aspects:

1) Novel Ensemble-Based Identification of Challenging Inputs. CodEn introduces a novel ensemble-based approach for identifying challenging inputs. This method effectively isolates those inputs that lead to low-quality outputs, even without access to ground truth. Conceptually, this method aligns closely with the test selection problem in software testing. Traditional testing of deep neural networks typically involves generating extensive test datasets and manually labeling samples to uncover erroneous model behaviors before deployment. Such labeling is expensive and time-consuming, particularly in domains requiring specialized expertise (*e.g.,* protein structure prediction). Moreover, only a small fraction of the test set typically triggers model failures. In contrast, our method can effectively isolate these critical challenging inputs without ground truth, and our experimental results demonstrate that it achieves significantly higher identification precision compared to SOTA

test selection and active learning methods (as shown in Table 11). Consequently, CodEn provides researchers and practitioners with a novel means of reducing testing costs and improving testing efficiency. Additionally, it can select challenging inputs to retrain the target model, thereby improving the robustness of target models.

2) On-the-Fly Output Repair Without Retraining. CodEn proposes a practical on-the-fly output repair framework that dynamically enhances the performance of deep code models without the need for retraining. This plug-and-play approach overcomes the limitations of traditional static training paradigms. CodEn significantly shortens the model update and adaptation cycle, which is particularly valuable in continuous integration and continuous deployment (CI/CD) scenarios. By reducing both deployment costs and manual intervention, such a dynamic framework paves the way for more agile and responsive software practices.

3) Importance of Domain-Specific Deep Code Models. Our work emphasizes that, in certain tasks, fine-tuned, domain-specific PCMs can outperform larger, general-purpose LLMs. Our research reveals that performance is not strictly proportional to model size; in fact, smaller models often offer clear advantages in terms of computational efficiency, such as reduced inference latency and lower computational cost. This insight is particularly relevant for environments with limited resources or stringent real-time requirements. Practitioners can thus achieve superior results by carefully selecting and fine-tuning smaller models, rather than defaulting to the largest available alternatives.

4) Complementary Integration of Multiple Models. Our study demonstrates that no single model consistently achieves optimal performance across diverse input cases. By accurately identifying challenging inputs, CodEn strategically assigns simpler cases to the base model and reserves the more complex instances for more capable models. This dynamic allocation strategy ensures that each model contributes its strengths to the overall task, thereby achieving an optimal combination of performance. Notably, our approach does not necessarily depend on collaboration between PCMs and LLMs. The proposed framework of "identifying challenging inputs and repairing low-quality outputs" can also facilitate collaboration among models of comparable size, thereby improving overall generation quality. To substantiate this claim, we adapt the CodEn framework to a scenario involving collaboration between PCMs of similar scale. Specifically, we adopt CodeGen as the target model and leverage a separately fine-tuned CodeT5 model to improve its performance across downstream tasks. Both CodeGen and CodeT5 are fine-tuned on the same downstream tasks in advance. Unlike LLMs, CodeT5 lacks emergent capabilities; thus, we adopt a lightweight adaptation strategy within CodEn. In detail, we use the quality classifier from CodEn to identify challenging inputs and subsequently leverage the fine-tuned CodeT5 model to regenerate the outputs for these inputs. We evaluate the performance improvements achieved by this model collaboration strategy, with results presented in Table 19.

Table 19. Performance gains from using CodeT5 to repair CodeGen's challenging inputs via CodEn.

Variants	Task		
	AG	CS	BF
CodeGen	55.15%	35.69	29.54%
CodeGen+CodeT5	59.25% (↑ 7.45%)	36.52 (↑ 2.33%)	31.57% (↑ 6.86%)

As shown in Table 19, across all downstream tasks, integrating CodeT5 consistently improves CodeGen's output quality on challenging inputs. Specifically, for assertion generation, the SAM metric increases from 55.15% to 59.25%, marking an increase of 7.45%. In bug fixing, the Exact Match metric increases from 29.54% to 31.57%, representing a gain of 6.86%. These results underscore the flexibility and generalizability of CodEn, demonstrating that the core approach of identifying challenging inputs and selectively repairing low-quality outputs can effectively combine the complementary strengths of similarly-sized models. Collectively, our findings suggest the broader applicability of our proposed framework in real-world software engineering tasks.

6.4 Threats to Validity

The primary threat to internal validity stems from the implementation of CodEn and the experimental scripts. To mitigate this threat, rigorous testing and code review processes are performed, and artifacts are made available for replication and practical use [1].

The threat to external validity primarily stems from the subjects used. We assess CodEn across three prominent code-related generation tasks: assertion generation, code summarization, and bug fixing. Future work will include additional tasks to evaluate CodEn thoroughly. In the bug-fixing task, we employ the TFix dataset for JavaScript bug fixing rather than a more traditional benchmark such as Defects4J. While Defects4J is widely used and includes real bugs from open-source projects, it covers fewer data points overall. In contrast, TFix provides over 100K ESLint-based code patches, enabling large-scale evaluation of on-the-fly repair. Moreover, TFix encompasses 52 distinct error types, offering a broad range of bug categories. This diversity is particularly valuable for assessing whether both CodEn and our baseline models can generate correct patches across many different error scenarios. Furthermore, TFix is widely used in program repair and code generation studies, supporting reproducibility and comparability with existing work [38, 51, 76, 94]. Additionally, we supplement TFix with Tufano *et al.* 's Java dataset to mitigate language-specific bias and examine the generality of our method. Nevertheless, our choices of TFix and BFP_{small} may not capture all the nuances of real-world production bugs or runtime dependencies, posing a potential external validity threat. Future work could investigate how CodEn performs on benchmarks with fully executable test suites (*e.g.,* Defects4J) or in-the-wild code repairs. Although our dataset is exclusively based on Javascript and Java, our approach is language-agnostic and adaptable to other programming languages. While we utilize ChatGPT for this implementation, it does not limit the generalizability of our method. CodEn is compatible with any LLM that supports generation and selection prompts.

The main construct threat to validity comes from the evaluation metrics. In line with prior research in static program repair, we evaluate patches primarily through textual metrics (Exact Match, and BLEU-4), where exact match is computed by strict string equality matching. We acknowledge that some automated program repair studies employ dynamic or compilation-based evaluations, where a candidate fix must pass unit tests. Our decision to rely on text-based metrics stems from TFix's design—bug fixes are derived from ESLint warnings rather than runtime tests, making textual patch alignment (including subword overlap) a suitable proxy for correctness. Obviously, this approach may overlook semantically equivalent but textually different solutions. As a result, our comparison presents the lowest bound of effectiveness from CodEn. Nevertheless, exact match and BLEU-4 are widely adopted as an evaluation metric by other learning-based source code processing techniques [14, 17, 46, 56] and automated program repair studies [15, 25, 47, 48, 76, 78, 90]. Incorporating test-based validation would likely provide deeper insights into patch correctness and is an avenue we plan to explore in future work.

7 RELATED WORK

To enhance the performance of deep code models, two primary approaches are commonly used: augmenting data for fine-tuning and designing advanced neural networks for retraining. In the former approach, studies like Yefet *et al.* [83] and Zhang *et al.* [87] propose constructing adversarial examples to fine-tune models. Yu *et al.* [85] and Allamanis *et al.* [5] introduce program transformation rules to augment code data, enhancing model generalization. Wang *et al.* [74] use curriculum learning to optimize fine-tuning and reduce overfitting, improving robustness. Concerning the latter approach, Buie *et al.* [12] integrate a tree-based convolutional neural network into a capsule network to improve learning from abstract syntax trees. Recent studies leverage contrastive learning with novel architectures and loss functions to boost performance [44, 77]. However, these techniques are not suitable for on-the-fly enhancement of deployed models, which is the primary goal of CodEn. Unlike these methods, CodEn does not involve retraining or fine-tuning the target model. Instead, it improves performance by identifying challenging inputs and repairing low-quality outputs online. Tian *et al.* [67] present CodeDenoise, an innovative

technique for enhancing the performance of deployed deep code models through on-the-fly input denoising. However, CodeDenoise is unsuitable for generative tasks. Due to the autoregressive nature of output generation, CodeDenoise's direct modification of the inputs makes the resulting output changes unpredictable. Additionally, the expected results often contain project-specific tokens within input. Directly modifying the input might potentially alter these tokens, leading to incorrect output. In contrast, CodEn introduces an on-the-fly output repair mechanism specifically tailored for generative tasks. This allows CodEn to complement CodeDenoise, filling a significant gap in the current literature.

8 CONCLUSION AND FUTURE WORK

In this paper, we explore the on-the-fly enhancement of generation quality in deployed deep code models, without altering the model's parameters. We introduce CodEn, a novel framework with two key components: challenging input identification and an online repair module for low-quality outputs. Extensive experiments across three code-related generation tasks show that CodEn can be seamlessly integrated with various pre-trained code models, leading to substantial performance improvements. For example, in the assertion generation task, CodEn enhances the SAM (Semantic Accuracy Match) by 12.83% to 21.38%. Future work will explore the performance of CodEn under other downstream tasks. Moreover, this paper focuses on model collaboration between the deep code model and the large language model (LLM). Subsequent work will centre on the model collaboration between LLMs.

9 DATA AVAILABILITY

Our replication package, including source code and comprehensive results, is available at [1].

10 ACKNOWLEDGEMENTS

We appreciate the insightful insights provided by anonymous reviewers to improve the quality of the paper. This work was supported in part by the National Natural Science Foundation of China (No. 62372071), the Chongqing Technology Innovation and Application Development Project (No. CSTB2022TIAD-STX0007 and No. CSTB2023TIAD-STX0025), and in part by the Fundamental Research Funds for the Central Universities under Grant 2023CDJKYJH013.

REFERENCES

[1] 2024. https://anonymous.4open.science/r/CodeEn-C418/
[2] 2024. Interpretation of bleu score. https://cloud.google.com/translate/automl/docs/evaluate.
[3] 2024. PyTorch. https://pytorch.org/.
[4] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653* (2020).
[5] Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. 2021. Self-supervised bug detection and repair. *Advances in Neural Information Processing Systems* 34 (2021), 27865–27876.
[6] Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *2013 10th working conference on mining software repositories (MSR)*. IEEE, 207–216.
[7] Jordan T Ash, Chicheng Zhang, Akshay Krishnamurthy, John Langford, and Alekh Agarwal. 2019. Deep batch active learning by diverse, uncertain gradient lower bounds. *arXiv preprint arXiv:1906.03671* (2019).
[8] Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*. 65–72.
[9] Yejin Bang, Samuel Cahyawijaya, Nayeon Lee, Wenliang Dai, Dan Su, Bryan Wilie, Holy Lovenia, Ziwei Ji, Tiezheng Yu, Willy Chung, et al. 2023. A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity. *arXiv preprint arXiv:2302.04023* (2023).

[10] Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. 2021. Tfix: Learning to fix coding errors with a text-to-text transformer. In *International Conference on Machine Learning*. PMLR, 780–791.
[11] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
[12] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. Treecaps: Tree-based capsule networks for source code processing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 30–38.
[13] Qiuyuan Chen, Xin Xia, Han Hu, David Lo, and Shanping Li. 2021. Why my code summarization model does not work: Code comment improvement with category prediction. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–29.
[14] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1943–1959.
[15] Yiu Wai Chow, Luca Di Grazia, and Michael Pradel. 2024. Pyty: Repairing static type errors in python. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
[16] Norman Cliff. 2014. *Ordinal methods for behavioral data analysis*. Psychology Press.
[17] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *International conference on learning representations (ICLR)*.
[18] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Tianyu Liu, et al. 2022. A survey on in-context learning. *arXiv preprint arXiv:2301.00234* (2022).
[19] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).
[20] Lishui Fan, Jiakun Liu, Zhongxin Liu, David Lo, Xin Xia, and Shanping Li. 2024. Exploring the capabilities of llms for code change related tasks. *ACM Transactions on Software Engineering and Methodology* (2024).
[21] Zhiyu Fan, Haifeng Ruan, Sergey Mechtaev, and Abhik Roychoudhury. 2024. Oracle-guided Program Selection from Large Language Models. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 628–640.
[22] Yang Feng, Qingkai Shi, Xinyu Gao, Jun Wan, Chunrong Fang, and Zhenyu Chen. 2020. Deepgini: prioritizing massive tests to enhance the robustness of deep neural networks. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 177–188.
[23] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020 (Findings of ACL, Vol. EMNLP 2020)*. Association for Computational Linguistics, 1536–1547.
[24] Jerome H Friedman. 2001. Greedy function approximation: a gradient boosting machine. *Annals of statistics* (2001), 1189–1232.
[25] Michael Fu, Van Nguyen, Chakkrit Tantithamthavorn, Dinh Phung, and Trung Le. 2024. Vision transformer inspired automated vulnerability repair. *ACM Transactions on Software Engineering and Methodology* 33, 3 (2024), 1–29.
[26] David Furcy and Sven Koenig. 2005. Limited discrepancy beam search. In *IJCAI*. 125–131.
[27] Shuzheng Gao, Cuiyun Gao, Yulan He, Jichuan Zeng, Lunyiu Nie, Xin Xia, and Michael Lyu. 2023. Code structure–guided transformer for source code summarization. *ACM Transactions on Software Engineering and Methodology* 32, 1 (2023), 1–32.
[28] Shuzheng Gao, Wenxin Mao, Cuiyun Gao, Li Li, Xing Hu, Xin Xia, and Michael R Lyu. 2024. Learning in the Wild: Towards Leveraging Unlabeled Data for Effectively Tuning Pre-trained Code Models. *arXiv preprint arXiv:2401.01060* (2024).
[29] Shuzheng Gao, Hongyu Zhang, Cuiyun Gao, and Chaozheng Wang. 2023. Keeping pace with ever-increasing data: Towards continual learning of code intelligence models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 30–42.
[30] Zi Gong, Cuiyun Gao, Yasheng Wang, Wenchao Gu, Yun Peng, and Zenglin Xu. 2022. Source code summarization with structural relative position guided transformer. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 13–24.
[31] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850* (2022).
[32] Isabelle Guyon, Jason Weston, Stephen Barnhill, and Vladimir Vapnik. 2002. Gene selection for cancer classification using support vector machines. *Machine learning* 46 (2002), 389–422.
[33] Yao Hao, Zhiqiu Huang, Hongjing Guo, and Guohua Shen. 2023. Test input selection for deep neural network enhancement based on multiple-objective optimization. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 534–545.
[34] Sakib Haque, Zachary Eberhart, Aakash Bansal, and Collin McMillan. 2022. Semantic similarity metrics for evaluating source code summarization. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. 36–47.
[35] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2022. Lora: Low-rank adaptation of large language models. *ICLR* 1, 2 (2022), 3.

[36] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing source code with transferred api knowledge. (2018).
[37] Fred Jelinek, Robert L Mercer, Lalit R Bahl, and James K Baker. 1977. Perplexity—a measure of the difficulty of speech recognition tasks. *The Journal of the Acoustical Society of America* 62, S1 (1977), S63–S63.
[38] Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. 2023. Repair is nearly generation: Multilingual program repair with llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 5131–5140.
[39] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE transactions on software engineering* 28, 7 (2002), 654–670.
[40] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems* 30 (2017).
[41] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. 2005. An empirical study of code clone genealogies. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. 187–196.
[42] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.
[43] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM computing surveys* 55, 9 (2023), 1–35.
[44] Shangqing Liu, Bozhi Wu, Xiaofei Xie, Guozhu Meng, and Yang Liu. 2023. Contrabert: Enhancing code pre-trained models via contrastive learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2476–2487.
[45] Steve Lohr. 2012. For impatient web users, an eye blink is just too long to wait. *The New York Times* (2012), A1–L.
[46] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 101–114.
[47] Antonio Mastropaolo, Nathan Cooper, David Nader Palacio, Simone Scalabrino, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2022. Using transfer learning for code-related tasks. *IEEE Transactions on Software Engineering* 49, 4 (2022), 1580–1598.
[48] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 336–347.
[49] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. 2018. Regularizing and Optimizing LSTM Language Models. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=SyyGPP0TZ
[50] Fangwen Mu, Xiao Chen, Lin Shi, Song Wang, and Qing Wang. 2022. Automatic comment generation via multi-pass deliberation. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.
[51] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-based prompt selection for code-related few-shot learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2450–2462.
[52] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).
[53] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A conversational paradigm for program synthesis. *arXiv preprint arXiv:2203.13474* 30 (2022).
[54] Changan Niu, Chuanyi Li, Vincent Ng, Dongxiao Chen, Jidong Ge, and Bin Luo. 2023. An empirical comparison of pre-trained models of source code. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2136–2148.
[55] Yotam Perlitz, Ariel Gera, Michal Shmueli-Scheuer, Dafna Sheinwald, Noam Slonim, and Liat Ein-Dor. 2023. Active learning for natural language generation. *arXiv preprint arXiv:2305.15040* (2023).
[56] Michael Pradel and Koushik Sen. 2018. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–25.
[57] Amandeep Kaur Sandhu and Ranbir Singh Batth. 2021. Software reuse analytics using integrated random forest and gradient boosting machine learning algorithm. *Software: Practice and Experience* 51, 4 (2021), 735–747.
[58] Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. 2008. *Introduction to information retrieval*. Vol. 39. Cambridge University Press Cambridge.
[59] Ozan Sener and Silvio Savarese. 2017. Active learning for convolutional neural networks: A core-set approach. *arXiv preprint arXiv:1708.00489* (2017).
[60] Burr Settles. 2009. Active learning literature survey. (2009).
[61] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2020. On-the-fly adaptation of source code models. In *NeurIPS 2020 Workshop on Computer-Assisted Programming*.
[62] Weisong Sun, Chunrong Fang, Yudu You, Yun Miao, Yi Liu, Yuekang Li, Gelei Deng, Shenghan Huang, Yuchen Chen, Quanjun Zhang, et al. 2023. Automatic code summarization via chatgpt: How far are we? *arXiv preprint arXiv:2305.12865* (2023).

[63] Weifeng Sun, Hongyan Li, Meng Yan, Yan Lei, and Hongyu Zhang. 2023. Revisiting and Improving Retrieval-Augmented Deep Assertion Generation. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1123–1135.

[64] Weifeng Sun, Meng Yan, Zhongxin Liu, and David Lo. 2023. Robust Test Selection for Deep Neural Networks. *IEEE Transactions on Software Engineering* 49, 12 (2023), 5250–5278.

[65] T.T. Tanimoto. 1958. *An Elementary Mathematical Theory of Classification and Prediction*. International Business Machines Corporation. https://books.google.com.hk/books?id=yp34HAAACAAJ

[66] Chris Thunes. 2019. Javalang. https://github.com/c2nes/javalang.

[67] Zhao Tian, Junjie Chen, and Xiangyu Zhang. 2023. On-the-fly Improving Performance of Deep Code Models via Input Denoising. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 560–572.

[68] Zhao Tian, Junjie Chen, Qihao Zhu, Junjie Yang, and Lingming Zhang. 2022. Learning to construct better mutation faults. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[69] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. Learning how to mutate source code from bug-fixes. In *2019 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 301–312.

[70] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 4 (2019), 1–29.

[71] Ramakrishna Vedantam, C Lawrence Zitnick, and Devi Parikh. 2015. Cider: Consensus-based image description evaluation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4566–4575.

[72] Bei Wang, Meng Yan, Zhongxin Liu, Ling Xu, Xin Xia, Xiaohong Zhang, and Dan Yang. 2021. Quality assurance for automated commit message generation. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 260–271.

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

[74] Deze Wang, Zhouyang Jia, Shanshan Li, Yue Yu, Yun Xiong, Wei Dong, and Xiangke Liao. 2022. Bridging pre-trained models and downstream tasks for source code understanding. In *Proceedings of the 44th International Conference on Software Engineering*. 287–298.

[75] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* (2024).

[76] Weishi Wang, Yue Wang, Shafiq Joty, and Steven CH Hoi. 2023. Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 146–158.

[77] Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. 2021. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation. *arXiv preprint arXiv:2108.04556* (2021).

[78] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. *CoRR* abs/2109.00859 (2021). arXiv:2109.00859

[79] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On Learning Meaningful Assert Statements for Unit Test Cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1398–1409. https://doi.org/10.1145/3377811.3380429

[80] Bolin Wei, Yongmin Li, Ge Li, Xin Xia, and Zhi Jin. 2020. Retrieve and refine: exemplar-based neural comment generation. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 349–360.

[81] Frank Wilcoxon. 1992. *Individual comparisons by ranking methods*. Springer.

[82] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1482–1494.

[83] Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[84] Hao Yu, Yiling Lou, Ke Sun, Dezhi Ran, Tao Xie, Dan Hao, Ying Li, Ge Li, and Qianxiang Wang. 2022. Automated assertion generation via information retrieval and its integration with deep learning. In *Proceedings of the 44th International Conference on Software Engineering*. 163–174.

[85] Shiwen Yu, Ting Wang, and Ji Wang. 2022. Data augmentation by program transformation. *Journal of Systems and Software* 190 (2022), 111304.

[86] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. 2014. Recurrent Neural Network Regularization. *CoRR* abs/1409.2329 (2014). arXiv:1409.2329 http://arxiv.org/abs/1409.2329

[87] Huangzhao Zhang, Zhiyi Fu, Ge Li, Lei Ma, Zhehao Zhao, Hua'an Yang, Yizhe Sun, Yang Liu, and Zhi Jin. 2022. Towards robustness of deep program processing models—detection, estimation, and enhancement. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1–40.

[88] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1385–1397.
[89] Quanjun Zhang, Chunrong Fang, Tongke Zhang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2023. Gamma: Revisiting template-based automated program repair via mask prediction. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 535–547.
[90] Ting Zhang, Ivana Clairine Irsan, Ferdian Thung, David Lo, Asankhaya Sharma, and Lingxiao Jiang. 2023. Evaluating pre-trained language models for repairing api misuses. *arXiv preprint arXiv:2310.16390* (2023).
[91] Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024. Opencodeinterpreter: Integrating code generation with execution and refinement. *arXiv preprint arXiv:2402.14658* (2024).
[92] Zhi Zhou, Xu Chen, En Li, Liekang Zeng, Ke Luo, and Junshan Zhang. 2019. Edge intelligence: Paving the last mile of artificial intelligence with edge computing. *Proc. IEEE* 107, 8 (2019), 1738–1762.
[93] Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence. *arXiv preprint arXiv:2406.11931* (2024).
[94] Armin Zirak and Hadi Hemmati. 2024. Improving automated program repair with domain adaptation. *ACM Transactions on Software Engineering and Methodology* 33, 3 (2024), 1–43.

Received 29 November 2024; revised 19 August 2025; accepted 26 August 2025