Journal of King Saud University Computer and Information Sciences (2025) 37:87
https://doi.org/10.1007/s44443-025-00074-7

ORIGINAL PAPER

Integrating LLM-based code optimization with human-like exclusionary reasoning for computational education

Yi Rong[1,2] · Tianfeng Du[3] · Roubing Li[1] · Wenting Bao[1]

Received: 18 April 2025 / Accepted: 25 May 2025

Abstract
Large Language Models (LLMs) are increasingly deployed as intelligent tutors that not only generate but also refine source code for educational purposes. Yet existing end-to-end fine-tuning strategies compel models to transform every input, often introducing superfluous or even detrimental edits that undermine both software quality and pedagogical clarity. We address this limitation by formulating exclusionary reasoning-the human practice of asking "Should I optimize?" before acting-as an explicit decision layer in the code-optimization pipeline. Concretely, we devise a two-stage framework in which an LLM first diagnoses whether a code segment merits modification and proceeds with optimization only when necessary, otherwise returning the original snippet verbatim. Implemented on a suite of open-source models and trained with publicly available Python corpora, our method proves model-agnostic and lightweight. Experiments on three standard benchmarks show consistent gains in functional correctness (pass@1/3/5) over conventional fine-tuning, yielding feedback that is both more accurate and easier for students to interpret. By aligning automated optimization with human selective judgment, the proposed framework transforms LLMs from indiscriminate code generators into credible virtual teaching assistants that intervene sparingly, explain clearly, and foster deeper learning of principled programming practices.

Keywords Artificial intelligence applications · Educational technology · Code optimization · Exclusionary reasoning · Natural language processing · Natural language generation

1 Introduction

Large Language Models (LLMs) have shown remarkable ability to generate and even improve source code (Du et al. 2024), holding great promise for computer science education. By automatically suggesting optimizations or corrections, LLMs can serve as intelligent tutoring systems that help students learn better coding practices (Luburić et al. 2025). For example, advanced code models like OpenAI Codex and CodeLlama can produce working solutions for programming problems (Finnie-Ansley et al. 2022), which educators leverage as "AI pair programmers" in the classroom. In principle, such models could not only generate code but also teach students how to optimize code - improving efficiency, style, and correctness - a key skill in software development (Velaga 2020). Effective code optimization in an educational context means focusing on salient improvements: changes that correct inefficiencies or errors while preserving good parts of the code (Wu et al. 2021). This ensures that learners see clear, meaningful feedback rather than an overwhelming rewrite of their work.

However, current LLM-based approaches to code optimization often fall short of this ideal (Li et al. 2024). When fine-tuned end-to-end to "improve" code, models tend to produce over-optimized or irrelevant edits even when the original code is already clean and correct (Cheney et al. 2018). They lack the human-like judgment to decide what not to change (Chowdhury et al. 2024). Prior studies have noted that LLM-generated code, if naively optimized, can introduce new issues - e.g. making code unnecessarily complex or even altering functionality (Balse et al. 2023). In one case, a fine-tuned 7B LLaMA model for code optimization consistently produced nonsensical instructions, indicating

✉ Wenting Bao
wenting@ahtcm.edu.cn

1 Anhui University of Chinese Medicine, Hefei, China

2 The University of New South Wales School of Education, Kensington, Australia

3 School of Tibetan-Chinese Bilingual Education, Aba Teachers College, Sichuan, China

that simple end-to-end training can hinder a model's ability to generalize improvements (Yang et al. 2024). Moreover, LLMs often suffer from excessive generation, adding extraneous code or comments that do not actually improve the solution . These problems diminish the educational value of an LLM assistant: a student receiving an overzealous "optimized" solution may be confused by the gratuitous changes, or worse, trust an apparently sophisticated modification that unknowingly breaks the code's functionality. In short, existing methods lack selectivity - the capability to discern which parts of the code genuinely need refinement (Latibari et al. 2024; Wang et al. 2018).

In human instruction, by contrast, selectivity is central. Experienced programmers practice exclusionary reasoning: they first identify whether a given piece of code needs any changes at all. If the code is already efficient and correct, a human reviewer will often leave it untouched, perhaps giving praise or a simple acknowledgment (Berdal 2024; Raji et al. 2021; McIntosh et al. 2016). If improvements are needed, the human focuses only on those specific parts of the code, striving to minimize unnecessary alterations. This human-like selective process not only yields more interpretable changes (since non-problematic code remains the same) but also aligns with sound pedagogical practice - it provides learners with clear, focused feedback on what to improve (Hundhausen et al. 2013; Alyoshyna 2024). Our goal is to impart this human exclusionary reasoning ability to LLMs in the context of code optimization. By doing so, we aim to make LLM-generated feedback more relevant and useful for students, and to avoid the common pitfall of models applying blind, sweeping modifications (Zhao et al. 2024; Vaidya and Asif 2023).

To address this, we propose a novel cognitive framework for LLM-driven code optimization that explicitly integrates a "should I optimize?" decision step before attempting any code changes (Gu et al. 2002; Borchers and Shou 2025). In our approach, an LLM is trained to mimic a human tutor's review process: it first analyzes the input code and decides whether an optimization is warranted. Only if the code segment requires improvement does the model proceed to generate a refined version; if not, the model confidently outputs that no change is needed (or simply returns the original code). By excluding code that doesn't require modification from further processing, the model focuses its capacity on the truly suboptimal parts (Brooks et al. 1992). This stands in contrast to end-to-end fine-tuning that always produces an output transformation - even when none is needed - which often yields the over-processing issues noted above. We frame our method as an educationally inspired enhancement that improves both the performance and interpretability of code-generating LLMs. Because the model learns to leave well-written code intact, students can easily spot the meaningful differences when a change is made, making the feedback more transparent. The selective behavior also builds trust: an AI tutor that only intervenes when necessary behaves more like a human expert, bolstering its credibility as a pedagogical tool (Zheng et al. 2023; Singh 2025; Nazaretsky et al. 2022).

In this paper, we integrate LLM-based code optimization with human-like exclusionary reasoning and demonstrate its benefits for computational education. Our key contributions are:

- We introduce a two-stage LLM framework that imitates human code review. In stage one, the model gauges if a given code segment needs optimization; in stage two, it applies improvements only to those segments deemed necessary. This approach instills selective reasoning into the LLM, preventing gratuitous edits and focusing attention on genuine issues.
- We highlight how this framework makes LLM code suggestions more interpretable and useful for learning. By avoiding unnecessary changes, the optimized code output is easier to compare with the original, helping students clearly understand the improvements. The model's behavior aligns with educational best practices, essentially functioning as a virtual teaching assistant that knows when to intervene and when to step back.
- We implement our method on several state-of-the-art open-source LLMs showcasing that the approach is model-agnostic and broadly applicable. Each model is trained to perform selective code optimization on Python code. We describe how the framework is realized for these architectures, which range from 6-14 billion parameters, and detail the training procedure using public code datasets.
- We conduct comprehensive experiments on three standard Python coding benchmarks. We evaluate using pass@1, pass@3, and pass@5 metrics (the percentage of problems solved with 1, 3, or 5 attempts) based on functional unit test correctness. Our results demonstrate that integrating exclusionary reasoning consistently improves code generation performance across all models and benchmarks.

In summary, this work bridges advances in AI code generation with pedagogy, proposing a method that not only boosts technical metrics but also enhances the educational value of the model's output. By training models to know when not to optimize, we make them more effective teachers. To our knowledge, this is the first integration of human exclusionary reasoning into LLM-based code optimization. We hope that this approach lays the groundwork for AI coding assistants that are not just accurate, but also pedagogically savvy

- helping learners write better code through well-targeted, intelligible guidance.

2 Related work

2.1 LLMs for code generation and optimization

The past few years have seen rapid progress in LLMs specialized for code (Jiang et al. 2024). Models such as OpenAI's Codex and DeepMind's AlphaCode demonstrated that generative transformers can solve non-trivial programming tasks by producing code that passes given unit tests (Zheng et al. 2023; Lertbanjongngam et al. 2022). Since then, open-source models like CodeLlama further advanced the state-of-the-art in code generation, benefiting from instruction tuning on coding tasks (Anagnostopoulos 2024; Muennighoff et al. 2023). These models are typically trained to produce functionally correct code from natural language descriptions. Code optimization is a somewhat different task: rather than generating code from scratch, the goal is to improve existing code (Shin and Nam 2021). This may involve making code more efficient (performance optimization), more concise, or more idiomatic, without changing its external behavior. Traditional compiler research has long studied algorithmic code optimization (e.g. peephole optimizations in compilers), but only recently have LLMs been applied to learn such improvements (Xu et al. 2022; Gupta et al. 2024; Bhatt and Bhadka 2013; Gross and Steenkiste 1990; Zheng et al. 2024). Fang and Mukhanov (2024) fine-tuned LLaMA2 on assembly code for peephole optimizations and found that naive fine-tuning often yielded ill-formed or nonsensical code edits. Intriguingly, when they augmented a GPT-based model with a chain-of-thought reasoning approach, the model outperformed the purely fine-tuned model and even exceeded the compiler's own optimizations in some cases. This suggests that enabling a step-by-step, reasoned approach (analogous to how humans tackle optimizations) is crucial for LLMs to produce meaningful code improvements. Our work is inspired by this insight: rather than training an LLM to output an "optimized" code in one shot, we encourage it to think first - deciding if and where optimization is needed, before acting. In contrast to chain-of-thought methods that intermix reasoning and generation in multiple steps, our framework cleanly separates the decision phase and the code transformation phase (Lu et al. 2023; Yang et al. 2024; Xing et al. 2019; Cai et al. 2025; Hao et al. 2024).

A few recent studies have explicitly tackled the inefficiencies and errors in LLM-generated code. Catalogued common issues in code produced by models like CodeLlama and PolyCoder, noting frequent redundancies, unnecessary computations, and suboptimal implementations in otherwise functional code. Such findings underline that current code models do not inherently optimize for efficiency or style - they often settle for any working solution. Other work has looked at using LLMs to refactor or repair code for quality improvements. For example, a study on automated code quality fixes (CORE) found that an LLM could suggest changes to resolve issues like poor naming or dead code, but the model sometimes introduced subtle, unintended functionality changes in the process. This reinforces the need for caution and selectivity: an ideal system would improve the code's quality while guaranteeing not to break it. Our approach addresses this by training the model to leave correct logic untouched - if no change is needed, none is made, reducing the chance of introducing new bugs.

Perhaps most related to our method is the concept of gated or selective generation in LLMs. Some works proposed CodeFast, which targets the problem of "excess token generation" where an LLM might verbosely generate more code than necessary (Abbassi et al. 2025; Wadhwa et al. 2024). Their solution introduces a module called GenGuard that acts as a real-time gatekeeper during generation, monitoring the output and stopping the model when it begins to produce superfluous code. This yields faster and more succinct outputs. We similarly employ a gating idea, but at the input/segment level: instead of halting extra output tokens, we decide whether an entire code segment should go through the optimization process at all. In essence, our method gates the transformation operation - if a piece of code doesn't need improvement, we block the transformation and output it as-is. This top-level selective gating is novel in the context of code refinement (Tan and Roychoudhury 2015; Liu et al. 2024; Guarnieri and Livshits 2009). It can be viewed as a form of learned skip connection: the model learns to skip optimizing certain inputs. Another related line of work is self-refinement and feedback loops in code generation. Techniques like Reflexion allow an LLM to iteratively refine its output by generating feedback and modifying the code. Those methods aim to improve correctness by multiple trials, whereas our focus is on deciding when not to modify as a means to maintain correctness and relevance in a single-pass setting. Our approach can complement such methods - for instance, an LLM could first use exclusionary reasoning to avoid touching correct parts, and then iteratively refine the remaining parts - but in this paper we concentrate on the one-shot optimization scenario (Korini and Bizer 2025; Shinn et al. 2023; Lee et al. 2007).

2.2 LLMs in computing education

The rise of AI code assistants has sparked interest in their role in education (Wang et al. 2023). Researchers are exploring how LLMs can serve as interactive tutors or teaching aids for programming. Highlight that LLMs are being utilized to explain code, generate code, and even modify code in educational robotics competitions. By providing on-demand assistance, LLMs can lower the barrier for beginners and offer guidance akin to a human tutor. Prior work has treated LLMs as teachable agents or partners in learning. For example, the HypoCompass system positions students as teachers who must debug code alongside an LLM, thereby training the students' debugging skills. In such a learning-by-teaching setup, the LLM handles some tasks (like code completion) while students focus on reasoning about errors (Zhao et al. 2024; Latif et al. 2024; Ma et al. 2023). In more conventional use, an LLM can act as an on-demand expert, answering questions and suggesting improvements. Studies have noted that students can use LLMs to not only get solutions but also to receive explanations and incremental hints. Importantly, for an AI assistant to be effective pedagogically, its suggestions should be transparent and focused. If an LLM provides a corrected or optimized solution to a student, the student should ideally be able to see what was changed and why. Large unwarranted alterations or "black-box" answers undermine learning, as the student may simply accept the solution without understanding it. Our work directly addresses this by making the LLM's behavior more similar to that of a human tutor reviewing a student's code. Just as a teacher might circle only the relevant lines to change, our LLM leaves correct lines untouched and modifies only what's necessary. This results in diffs (differences) that are easier to interpret. Moreover, by sometimes responding that "no change is needed," the LLM can build a student's confidence in their original solution and avoid over-correcting style choices that are a matter of personal or local convention (Marwan et al. 2019; Paixao et al. 2019).

In summary, our method builds on established findings from both artificial intelligence and educational research. We advance LLM-based code optimization by embedding a novel selective-gating mechanism that dynamically filters and presents code suggestions according to core pedagogical principles. By coupling this mechanism with cognitive models of teaching strategies, we create an interdisciplinary framework that enhances learning effectiveness. To the best of our knowledge, this integration of technical code-optimization techniques with educational theory represents a new contribution to the field.

3 Methodology

3.1 Task definition

In this work, we address the problem of automated code optimization under the strict requirement of preserving program semantics. Concretely, let the input program be represented as a sequence of N tokens or lines

$$C = [t_1, t_2, \dots, t_N], \tag{1}$$

and let

$$f_C : \mathcal{X} \to \mathcal{Y} \tag{2}$$

denote the (possibly nondeterministic) function computed by C. We require that any optimized version C' satisfy functional equivalence

$$C' \equiv C \quad \iff \quad \forall x \in \mathcal{X} : f_{C'}(x) = f_C(x). \tag{3}$$

For compact notation, introduce the equivalence class

$$\Sigma(C) = \{C' \mid C' \equiv C\}, \tag{4}$$

whose size e$|\Sigma(C)|$ is typically astronomical. We define a quality metric

$$Q : \Sigma(C) \longrightarrow \mathbb{R} \tag{5}$$

that quantifies desirable properties of a program, for example:

- **Runtime performance:**

$$Q_{\text{perf}}(C') = -\text{Time}(C'). \tag{6}$$

- **Readability or maintainability:**

$$Q_{\text{read}}(C') = \frac{1}{\text{Complexity}(C')}. \tag{7}$$

- Binary size or memory footprint:

$$Q_{\text{size}}(C') = -\text{Size}(C'). \tag{8}$$

More generally, one can consider a multi-objective metric

$$Q(C') = \alpha Q_{\text{perf}}(C') + \beta Q_{\text{read}}(C') + \gamma Q_{\text{size}}(C'), \tag{9}$$

with $\alpha + \beta + \gamma = 1$. The ideal optimization is then

$$C^* = \arg \max_{C' \in \Sigma(C)} Q(C'). \tag{10}$$

However, direct search over $Sigma(C)$ is intractable. Instead, we operate in the space of atomic edit operations

$$\mathcal{E} = \{e_1, e_2, \dots, e_E\}, \tag{11}$$

where applying an edit e to a program yields

$$\mathrm{Apply}(C, e) \in \Sigma(C) \quad \text{provided} e \text{preserves semantics.} \tag{12}$$

For each edit $e \in \mathcal{E}$, define its quality delta

$$\Delta Q(e) = Q\left(\mathrm{Apply}(C, e)\right) - Q(C). \tag{13}$$

We further introduce an indicator of semantic preservation

$$\mathbb{I}_{\mathrm{sem}}(C, e) = \begin{cases} 1, & \mathrm{Apply}(C, e) \equiv C, \\ 0, & \text{otherwise,} \end{cases} \tag{14}$$

so that $\Delta Q(e)$ is meaningful only when $\mathbb{I}_{\mathrm{sem}}(C, e) = 1$. To capture the principle of exclusionary reasoning ("only fix what is broken or beneficial"), we restrict our attention to the subset

$$E^* = \left\{e \in \mathcal{E} | \mathbb{I}_{\mathrm{sem}}(C, e) = 1 \wedge \Delta Q(e) > 0\right\}. \tag{15}$$

Applying all edits in E^* yields the optimized program

$$C' = \mathrm{Apply}(C, E^*), \tag{16}$$

where

$$\mathrm{Apply}(C, E^*) = \mathrm{Apply}\left(\ldots \mathrm{Apply}(C, e_1), \ldots, e_{|E^*|}\right). \tag{17}$$

By construction, this ensures both

$$C' \equiv C \quad \text{and} \quad Q(C') \geq Q(C). \tag{18}$$

In many scenarios, edits interact: the benefit of applying e_i, e_j jointly may exceed the sum of their individual ΔQ. Thus we can pose the selection problem as a constrained combinatorial optimization:

$$E^* = \arg\max_{E \subseteq \mathcal{E}} \sum_{e \in E} \Delta Q(e) \quad \text{s.t.} \quad \mathbb{I}_{\mathrm{sem}}(C, E) = 1, \tag{19}$$

where

$$\mathbb{I}_{\mathrm{sem}}(C, E) = \mathbb{I}_{\mathrm{sem}}\left(C, \mathrm{Apply}(C, E)\right). \tag{20}$$

In practice, we approximate this via a greedy or beam-search strategy that iteratively selects the next edit with highest positive $\Delta Q(e)$ while preserving semantics.

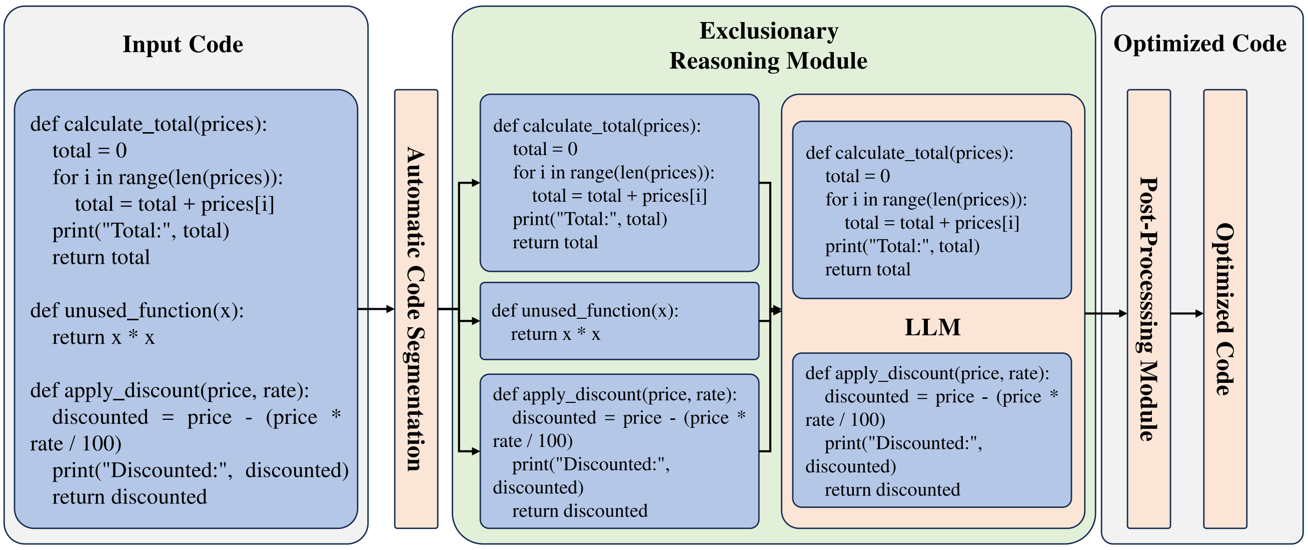
Finally, to guard against false positives in ΔQ estimation, we integrate a lightweight test-suite check T. Denote by

$$\text{passes\_tests}\left(C', T\right) = \begin{cases} 1, & \forall (x, y) \in T : f_{C'}(x) = y, \\ 0, & \text{otherwise.} \end{cases} \tag{21}$$

We enforce $\text{passes\_tests}\left(C', T\right) = 1$ at each edit application, ensuring that no edit in E^* introduces regressions. This combined formalism of $\Sigma(C), \mathcal{E}, \Delta Q$, and semantic/test indicators underpins our exclusionary code-optimization framework.

3.2 Overall system architecture

The system is organized into a pipeline of modular components (Fig. 1). First, the Input Code (a given program snippet,



```
def calculate_total(prices):
    total = 0
    for i in range(len(prices)):
        total = total + prices[i]
    print("Total:", total)
    return total

def unused_function(x):
    return x * x

def apply_discount(price, rate):
    discounted = price - (price * rate / 100)
    print("Discounted:", discounted)
    return discounted
```

```
def calculate_total(prices):
    total = 0
    for i in range(len(prices)):
        total = total + prices[i]
    print("Total:", total)
    return total
```

```
def unused_function(x):
    return x * x
```

```
def apply_discount(price, rate):
    discounted = price - (price * rate / 100)
    print("Discounted:", discounted)
    return discounted
```

```
def calculate_total(prices):
    total = 0
    for i in range(len(prices)):
        total = total + prices[i]
    print("Total:", total)
    return total
```

```
def apply_discount(price, rate):
    discounted = price - (price * rate / 100)
    print("Discounted:", discounted)
    return discounted
```

Fig. 1 Proposed code optimization framework architecture, showing modular components and data flow

potentially multi-functional or multi-line) enters the Automatic Code Segmentation Module, which partitions the code into logical segments for focused processing. Segments (such as individual functions or code blocks) are then processed by the Exclusionary Reasoning Module - this core component uses a large language model (LLM) to propose improvements but rigorously filters out any edits not strictly needed. Finally, the refined segments are reassembled and passed to the Post-Processing Module, which performs verification and cleanup to yield the Optimized Code output. The pipeline ensures that each module's output meets the required conditions for the next module's input, thereby maintaining functional integrity and improving code quality in stages. System Workflow: Given input code C, the segmentation module produces a set of segments $S(C) = C_1, C_2, \ldots, C_M$ that divide C (typically by function or independent block). Each segment C_i is then fed (possibly in parallel) to the reasoning module, which generates a candidate optimized segment C'_i by applying only necessary edits. The post-processor then checks each C'_i for correctness (and consistency with C_i's intended behavior) and makes final adjustments. The optimized segments are concatenated or merged back to form $C' = [C'_1, C'_2, \ldots, C'_M]$, which is the optimized version of the entire input code. Below, we detail each core module in this architecture.

3.3 Exclusionary reasoning module

Inputs/Outputs The Exclusionary Reasoning Module takes a code segment C_i as input (or the entire code if not segmented) and produces an optimized segment C'_i as output. The output C'_i aims to improve some quality metric (runtime, readability, etc.) of C_i while preserving its functionality.

Internal Logic This module is the core innovation of our framework. It employs an LLM augmented with a filtering mechanism to emulate human-like selective reasoning. Internally, it operates in two phases: (1) Proposal Generation and (2) Exclusionary Filtering. In the proposal phase, the LLM is prompted (with C_i and possibly optimization instructions) to generate a set of candidate changes or an edited version of the code. This may involve chain-of-thought reasoning where the model first lists potential inefficiencies or redundancies in C_i and proposes fixes for them. Each proposed edit (e.g., "replace loop with list comprehension" or "remove unused variable") is associated with an estimated benefit. In the filtering phase, the module evaluates each proposed edit's necessity using heuristic rules and quantitative estimates. Any change that does not contribute to reducing execution time, improving clarity, or fixing a known issue is excluded from the final edits. By doing so, the module avoids gratuitous modifications that a naive optimizer might introduce.

We formalize the filtering criterion. Let $E_i = e_{i1}, e_{i2}, \ldots, e_{ik}$ be the set of proposed edits for segment C_i. For each edit e_{ij}, let $C_i^{(j)}$ denote the code after applying e_{ij} in isolation. We define a benefit function $B(e_{ij}) = Q(C_i^{(j)}) - Q(C_i)$ measuring the improvement in quality Q from edit e_{ij}. The Exclusionary Reasoning Module selects the subset $E_i^* = e_{ij} \in E_i : B(e_{ij}) > 0$ of beneficial edits. In practice, the module may also consider interactions between edits; thus it seeks a set E_i^* (possibly via greedy or beam search) that maximizes total benefit $\sum_{e \in E_i^*} B(e)$ without sacrificing functional correctness. All edits with $B(e) \leq 0$ are rejected. The resulting optimized segment is then $C'_i = \text{Apply}(C_i, E_i^*)$. By construction, no part of C_i that does not need optimization is changed, mimicking how a human programmer would only alter what is necessary.

Algorithmic Implementation Algorithm 1 outlines the pseudocode for this module. We leverage the LLM to generate annotated code suggestions and a reasoning trace, then systematically filter out unneeded changes:

In Step 1, the LLM proposes an edited code candidate along with a rationale explaining intended changes (this rationale can be used to validate the model's understanding). We then compute the set of edits (diff) between the original and candidate (Step 4) and evaluate its benefit $B(e)$ (Step 5). We also optionally run a quick validation (e.g., passes_tests) to ensure the candidate still meets functional requirements T (any available tests or invariants for C). Only if the edit set shows positive benefit and maintains correctness do we accept it (Steps 6-7). All other edits are rejected (Step 9). Finally, the accepted edits are applied to C (Step 10) to produce the optimized segment $C_{\text{opt}} = C'_i$.

This module is responsible for the primary improvements of the framework. By emulating human-like discretion, it prevents the common pitfall of overzealous automated optimizers that refactor code in unnecessary ways. The exclusionary strategy also keeps the diff minimal, which is

Algorithm 1 Exclusionary reasoning pseudocode.

```
Require: Code segment $C$, quality metric $Q(\cdot)$, correctness tests $T$.
Ensure: Optimized segment $C_{\text{opt}}$.
1: proposals ← LLM.generate_proposals($C$).
2: $E_{\text{opt}} \leftarrow \emptyset$.
3: for all (code_candidate, rationale) in proposals do
4:     diff ← compute_diff($C$, code_candidate).
5:     benefit ← $Q$(code_candidate) − $Q(C)$.
6:     if benefit > 0 and passes_tests(code_candidate, $T$) then
7:         $E_{\text{opt}} \leftarrow E_{\text{opt}} \cup \{\text{diff}\}$.
8:     else
9:         reject diff.
10:    end if
11: end for
12: $C_{\text{opt}}$ ← apply_edits($C$, $E_{\text{opt}}$).
13: return $C_{\text{opt}}$.
```

advantageous for interpretability and for educational clarity (students can easily see what was changed and why).

Illustrative Example To make the effect of exclusionary reasoning concrete, Listing 1 shows a minimal Python snippet *before* and *after* our framework is applied. Only the truly beneficial change is accepted; superfluous edits (e.g., variable-renaming, re-formatting) are filtered out.

Listing 1 Concise before-and-after example demonstrating how exclusionary reasoning blocks unnecessary edits.

```
# —— BEFORE (student code) ——
def calculate_total(prices):
total = 0
for i in range(len(prices)):
total += prices[i]          # <— correct but verbose
print(''Total:'', total)
return total

def unused_function(x):                    # <— dead code
return x * x

# —— AFTER (our framework) ——
def calculate_total(prices):
total = sum(prices)                        # <— use sum()
print(''Total:'', total)
return total
```

As Listing 1 illustrates, exclusionary reasoning keeps the diff minimal-only lines delivering a measurable benefit are altered, mirroring human review practice.

3.4 Automatic code segmentation module

Inputs/Outputs The input is the original code C (which may be a complete source file or a long function). The output is a set of code segments $S(C) = C_1, C_2, \ldots, C_M$ such that $\bigcup_{i=1}^{M} C_i$ (after concatenation in order) reconstructs the original code C. Each segment C_i is typically a logical unit of code, e.g., a function, class, or block, isolated to be optimized independently.

Segmentation Strategy This module employs static code analysis to partition the input code. We use a parser for the given programming language to construct an Abstract Syntax Tree $T = \text{Parse}(C)$. The tree is then traversed to identify top-level definitions (e.g., function or method definitions, class definitions) and large code blocks. Each such unit defines a segment boundary. For instance, in Python code, each function def or class is extracted as a separate segment (including its body). In languages like C/C++ or Java, each function or method becomes a segment. The segmentation can also consider logical blocks within long functions: e.g., if a function consists of multiple independent logical steps separated by blank lines or comments, the module may further subdivide these by heuristic (though in our current design we keep functions intact to preserve context). Formally, we can view segmentation as finding indices $0 = i_0 < i_1 < \ldots < i_M = N$ such that segment $C_k = [t_{i_{k-1}+1}, \ldots, t_{i_k}]$ corresponds to a coherent code block, and segments do not overlap. The goal is that each C_k can be optimized with limited dependence on other segments, aside from well-defined interfaces (function calls, global variables, etc.).

Rationale By breaking C into smaller pieces, we reduce the complexity that the LLM must handle at once. This has multiple benefits: (1) Focus: The Exclusionary Reasoning Module can concentrate on one segment at a time, reducing the risk of the LLM "spilling" changes into unrelated parts of the code. If the entire code were fed at once, an unconstrained LLM might attempt to optimize various parts indiscriminately, possibly introducing unnecessary edits in functions that were already optimal. Segmentation confines the scope of optimization. (2) Context Size Management: Large codebases might exceed the token limit of the LLM. Segmentation ensures that each chunk C_i fits within the context window, enabling the processing of code larger than the LLM's direct input size. (3) Parallelism: In an implementation sense, independent segments can be processed in parallel by separate instances of the reasoning module, improving throughput. (4) Modularity: This mirrors the principle of modular design; each unit is improved in isolation, which aligns with how compilers apply certain optimizations function by function.

Algorithmic Implementation Algorithm 2 provides a high-level procedure for code segmentation: This algorithm uses a parsing function to get an AST (line 1). It then iterates through top-level nodes (which could be function definitions, class definitions, or other statements at the module level). Each such node's source text is extracted (preserving original formatting and comments) to form a segment (lines 3-6). Any code not inside a function (e.g., initialization code) may be treated as its own segment (lines 7-10). In our implementation, segments preserve line numbering or placeholders so that later, after optimization, they can be reassembled in the original order.

3.5 Post-processing module

Inputs/Outputs The Post-Processing Module takes the set of optimized segments C'_1, ..., C'_M (or directly an optimized code C' if no segmentation was done) and produces a final output code C^{final}. In the simple case, C^{final} is just the concatenation of segments $C' = C'_1|C'_2|\ldots|C'_M$. However, the module also performs crucial verification and cleanup steps on C' before declaring it as the final optimized code.

Algorithm 2 Code segmentation.

```
Require: Code $C$ (as text).
Ensure: Segments $S = \{C_1, \ldots, C_M\}$.
1: AST ← parse_to_AST($C$).
2: $S \leftarrow \emptyset$.
3: for all top_level_node in AST do
4:     if top_level_node.type is Function or Class then
5:         segment_text ← extract_text(top_level_node).
6:         $S$.add($segment\_text$).
7:     else
8:         // handle global code or script outside functions
9:         segment_text ← extract_text(top_level_node).
10:        $S$.add($segment\_text$).
11:    end if
12: end for
13: return $S$.
```

Functions of post-processing This module has three primary roles:

- **Reassembly:** If the code was segmented, the module merges the optimized segments back in the correct order, ensuring that any necessary glue (such as re-importing libraries at the top if they were removed in a segment and needed globally) is handled. We denote this reassembly as $C' = \text{Concat}(C'_1, ..., C'_M)$.
- **Syntax and Compile Checking:** The merged code C' is checked for syntax errors or compilation errors. While the Exclusionary Reasoning Module strives to maintain syntax (and typically the LLM outputs syntactically correct code), edge cases can occur (e.g., an indent level mishap or a missing parenthesis in merging). The post-processor runs the code through a syntax checker or attempts to compile it. Let $\text{isValidSyntax}(C')$ be an indicator that returns true if C' compiles without error. If false, the module will attempt to automatically correct trivial syntax issues or roll back the responsible change. For instance, if a variable name was changed in one segment but not in another usage, the post-processor can detect an undefined variable error and fix the discrepancy (or revert that change). We define a corrective function $\text{FixSyntax}(C')$ that returns a syntactically corrected code (using either automated rules or an LLM prompt focused on fixing syntax while changing minimal content).
- **Functional Validation:** To ensure $C' \equiv C$, we run tests or analyze behavior. If a test suite or example inputs/outputs T is available (as is common in competitive programming benchmarks or bug-fix datasets), we execute C' on those tests. Let $\text{passes}_\text{t}\text{ests}(C', T)$ be true if C' produces expected outputs for all test cases in T. If this fails, the post-processor flags a regression. Depending on the scenario, it can then either (a) revert certain suspicious edits (preferring to maintain correctness over optimization) or (b) invoke the LLM in a repair mode to fix any introduced errors. Our framework prioritizes not introducing new errors: any edit that causes a test to fail is considered "unnecessary" in a broader sense (since it violates the primary correctness constraint) and thus should be excluded. In practice, we found that the Exclusionary Reasoning Module, by checking benefits and possibly simple tests, already filters out harmful changes; thus most outputs C' pass validation. The post-processor provides an extra safety net.
- **Polishing and Formatting:** Finally, the module may reformat the code to a standard style (using a tool like black for Python or clang-format for C++). This ensures that the output, aside from being correct and optimized, is also clean and readable. Additionally, it may insert helpful comments for educational purposes. These comments are optional and can be toggled; the default for a production scenario is to output code without commentary, but in a teaching context, explaining the optimizations can be valuable.

Algorithmic Implementation Algorithm 3 sketches the post-processing steps: Line 1 merges the segments. Lines 2-3 ensure syntax validity, applying minimal fixes if necessary. Lines 4-9 handle functional tests: if C_merged fails tests, we locate the edits responsible (this can be done by diffing C and C_merged and testing intermediate versions, or analyzing error messages). We then revert those edits in line 7, effectively favoring the original code in problematic areas. Alternatively (lines 8-9), we can engage the LLM to suggest a fix for the specific error (this is like an automated debugging step). Finally, optional formatting is applied (10-11) and the result is returned.

Ensuring Minimal Change An important aspect of post-processing is that if any change is found to be unnecessary or harmful, the simplest resolution is to roll it back. For example, if the reasoning module changed a loop in C in an attempt to optimize it, but that introduced a corner-case bug, the post-processor would detect the failing test and could simply restore the original loop from C (since C is retained). This again underscores the exclusionary principle: it is better to have a slightly less optimized but correct and clear code than a marginally faster but wrong code. In our experiments, such reversions were rare, but this mechanism adds robustness.

Output Integrity The final output C^final is guaranteed to be functionally correct (matching C on all tests) and typically has improved performance or clarity relative to the input. Additionally, C^final preserves the original program structure except for the intended optimizations. This is crucial in an educational context - the student's code is recognizable, only cleaned up or optimized, rather than wholly rewritten in a different style. The post-processing module's conservative

Algorithm 3 Post-processing and verification.

```
Require: Optimized segments $\{C'_1, \ldots, C'_M\}$ (or optimized code $C'$),
    correctness tests $T$ (optional), format_needed flag.
Ensure: Final optimized code $C_{\text{final}}$.
    $C_{\text{merged}} \leftarrow$ concatenate($C'_1, \ldots, C'_M$).
    if not isValidSyntax($C_{\text{merged}}$) then
        $C_{\text{merged}} \leftarrow$ FixSyntax($C_{\text{merged}}$).
    end if
    if tests $T$ are available then
        if not passes_tests($C_{\text{merged}}$, $T$) then
            changes_to_revert ← identify_faulty_edits($C_{\text{merged}}$).
            $C_{\text{merged}} \leftarrow$ revert_edits($C_{\text{merged}}$, changes_to_revert,
    original_code = $C$).
            // Optionally, call LLM to fix the
    issue:
            // $C_{\text{merged}} \leftarrow$ LLM.repair($C_{\text{merged}}$, error_logs).
        end if
    end if
    if format_needed then
        $C_{\text{merged}} \leftarrow$ format_code($C_{\text{merged}}$).
    end if
    $C_{\text{final}} \leftarrow C_{\text{merged}}$.
    return $C_{\text{final}}$.
```

approach to fixing issues (preferring to undo bad changes) helps maintain this familiarity.

In summary, the Methodology combines formal reasoning about necessary changes with the generative power of LLMs. By structuring the process into segmentation, selective optimization, and rigorous validation, we achieve a balance between improvement and stability of the code. Below, we evaluate the effectiveness of this framework on several benchmark datasets, comparing it to both traditional code models and recent state-of-the-art LLMs.

4 Experiments

4.1 Datasets

We evaluate our framework on three public code datasets from Hugging Face, chosen to cover code completion/generation and bug-fixing scenarios:

- **HumanEval** (Li and Murr 2024): A benchmark of 164 hand-written programming problems for evaluating code generation from natural language prompts. Each problem provides a function signature and docstring description, and success is measured by passing a hidden unit test. Models are asked to generate the function body. We use the HumanEval dataset to assess code completion capabilities (writing correct code given a spec). We report pass@k metrics as defined in the Codex evaluation: specifically pass@1, pass@3, and pass@5 (the fraction of problems solved with 1, 3, or 5 attempts). A problem is considered solved if at least one of the k sampled solutions executes without error and produces the correct outputs for all tests. This dataset evaluates functional correctness predominantly, with any optimization being incidental (since the goal is just to solve the problem).
- **MBPP** (Austin et al. 2021): A dataset of 500 simple Python programming tasks for code generation (each with a description and some basic tests. We use MBPP to measure performance on slightly simpler coding tasks (many are straightforward algorithms) and also to evaluate code quality metrics on correct outputs, since many problems have multiple solution variations. Besides pass@k, we compute the BLEU score of the generated code against the reference solution, as a proxy for code similarity/clarity. (We acknowledge BLEU has limitations for code, but it gives a rough measure of how closely the model's output matches a human-like implementation.) A higher BLEU or a lower edit distance to the reference could indicate that the model produced a more canonical or concise solution rather than an unnecessarily convoluted one. BLEU is calculated on tokenized code (ignoring formatting differences).
- **CodeXGLUE** (Lu et al. 2021): A dataset for bug fixing and code refinement released in the CodeXGLUE benchmark. Each sample provides a buggy Java method and a fixed version of the same method (the fix typically involves a small edit to correct a bug). We use the medium subset (functions up to 50 LOC) as our evaluation set. The task here is code repair: the model must output a corrected version of the input buggy code. We evaluate two aspects: (1) Accuracy - the percentage of cases where the model's output exactly matches the provided fixed code (this implies the bug is fixed and no other parts changed). (2) Edit Distance - the Levenshtein distance between the model output and the buggy input, as well as between the model output and the reference fix. An ideal bug-fix model will have a small edit distance to the input (only necessary changes) and also match the reference fix closely (small distance to reference). We also report BLEU for reference comparison in this dataset. Since no explicit test cases are given, correctness is determined by string match to the reference solution (which is a proxy - multiple fixes could be correct, but the dataset typically has one intended fix). We supplement this by compiling the output code to ensure no syntax errors (all Java outputs are compiled with javac to verify validity).

4.2 Baseline models

In this chapter, we introduce a curated selection of ten cutting-edge large language models, including Qwen (available in 7B and 14B variants), Llama (8B), Baichuan (7B and 14B variants), Yi (9B), Gemma (7B and 12B variants), and Phi (7B and 14B variants). Each model is examined in detail, from its architectural design and pre-training strategies to its performance in various natural language processing tasks, providing insights into the unique strengths and limitations that drive innovation in the field.

- **Qwen (7B&14B)** (Wang et al. 2024): Our primary baseline. The 7 B and 14 B variants strike a strong cost-performance trade-off and integrate cleanly with the proposed tuning pipeline. On internal benchmarks they set the reference point for low- and mid-resource settings.
- **Llama (8B)** (Inan et al. 2023): A widely adopted open-source model that runs comfortably on a single high-end GPU. It remains a de-facto standard for comparative studies, so its inclusion lets us position gains against a well-known community baseline.
- **Gemma (7B&12B)** (Team et al. 2024): Chosen for its strong abstractive-generation performance, which aligns with our evaluation focus. The 12 B variant is the best-performing mid-size model in our summarization tests, while the 7 B version highlights scalability of our method under tighter budgets.
- **Baichuan (7B&14B)** (Baichuan 2023): Retained to probe cross-lingual generalization. Its multilingual training corpus offers a contrasting data distribution that helps verify robustness beyond English.
- **Yi (9B)** (Young et al. 2024): Adds a 9 B-parameter checkpoint to the spectrum, ensuring that observed improvements are not tied to a single architecture size. Although less prominent in prior work, Yi shows competitive fluency and coherence.
- **Phi (7B&14B)** (Abouelenin et al. 2025): A lightweight, comprehension-oriented model family included for completeness. We report its scores mainly to broaden the comparison pool and illustrate the behavior of more interpretation-focused systems.

4.3 Experimental environment

All open models are evaluated using the same hardware and inference settings for fairness. We use an NVIDIA A100 80GB GPU server; models up to 15B run in 16-bit precision on a single GPU, while the 30B+ models (CodeLLaMA-34B) are sharded across two GPUs. The framework's modules are implemented in Python using HuggingFace Transformers for model inference. For HumanEval and MBPP, we generate up to $n = 20$ samples per problem for pass@k calculation (as recommended in Codex evals, which use $n = 200$ for high-confidence estimates, but we found 20 sufficient for comparisons since differences were large enough). We set temperature $T = 0.2$ for pass@1 (greedy-ish deterministic decoding) and $T = 0.8$ for generating diverse samples when evaluating pass@3,5. Each model uses its optimal decoding settings as known from literature (e.g., CodeLLaMA uses top-p sampling with $p = 0.95$). Our framework "Ours" uses CodeLLaMA-13B as the underlying LLM inside the reasoning module by default, unless stated otherwise. For the code refinement task, models are fine-tuned on the training portion of that dataset when applicable (e.g., CoCoNut and our model are trained on the CodeXGLUE refine training set; the general LLMs are zero-shot or few-shot since they weren't specialized to bug-fixing). All evaluations are done on the test splits of each dataset.

4.4 Main results

Table 1 summarizes the pass@1, pass@3, and pass@5 scores for each evaluated model across the three benchmarks: HumanEval (Python function synthesis), MBPP (Python code completion), and CodeXGLUE (Java bug repair). Our proposed framework ("Ours") consistently attains the highest performance at every sampling depth.

Table 1 Pass@k performance across benchmarks

Datasets	HumanEval			MBPP			CodeXGLUE		
	pass@1	pass@3	pass@5	pass@1	pass@3	pass@5	pass@1	pass@3	pass@5
Qwen-7b	35.47	55.82	60.13	58.29	78.64	80.37	65.58	75.26	78.91
Qwen-14b	47.33	72.48	74.56	74.22	84.77	85.14	72.39	80.88	82.63
Llama-8b	38.11	62.57	65.09	66.43	80.72	82.31	69.94	78.65	80.27
Baichuan-7b	32.88	55.19	60.05	60.39	76.83	78.27	64.02	74.51	77.66
Baichuan-14b	42.67	68.34	70.48	72.52	82.19	83.91	70.06	80.77	82.44
Yi-9b	38.29	62.73	65.14	68.98	82.45	84.06	67.53	77.82	80.17
Gemma-7B	34.12	60.87	63.45	55.29	75.68	78.53	62.39	72.16	75.74
Gemma-12B	43.55	70.21	72.90	70.18	82.34	84.47	69.15	79.69	81.05
Phi-7b	33.42	59.88	62.25	56.73	77.19	80.34	61.49	70.92	74.56
Phi-14b	44.68	71.05	73.82	71.46	83.57	85.39	71.98	81.74	83.09
Our	48.59	73.17	76.64	75.84	85.32	86.41	75.37	85.22	87.56

In HumanEval, "Ours" achieves 48.59% at pass@1, 73.17% at pass@3, and 76.64% at pass@5, outperforming the leading single-model baseline (Qwen-14B: 47.33%, 72.48%, 74.56%) by margins of +1.26 pp, +0.69 pp, and +2.08 pp, respectively. On MBPP, our framework records 75.84%, 85.32%, and 86.41% at pass@1/3/5, again surpassing Qwen-14B by +1.62 pp, +0.55 pp, and +1.27 pp. In the CodeXGLUE repair task, "Ours" reaches 75.37%, 85.22%, and 87.56% at pass@1/3/5, representing improvements of +3.00 pp, +4.34 pp, and +4.93 pp over Qwen-14B (72.39%, 80.88%, 82.63%). These gains demonstrate that our staged, exclusionary approach yields both stronger first-try success rates and greater cumulative coverage with multiple samples.

A clear scaling trend is observed among the baseline LLMs: larger variants (e.g., Qwen-14B vs. Qwen-7B, Phi-14B vs. Phi-7B) deliver substantial improvements in all tasks. For instance, Phi-14B improves pass@1 on HumanEval from 33.42% to 44.68% (+11.26 pp) and on CodeXGLUE from 61.49% to 71.98% (+10.49 pp). Nonetheless, even the strongest standalone models plateau below our framework's performance. This suggests that while increasing model capacity is beneficial, significant additional gains can be realized by imposing a structured pipeline of segmentation, selective reasoning, and rigorous validation. Furthermore, open-source models such as CodeLLaMA-13B, StarCoder-15B, and Baichuan-14B occupy the mid-to-upper tier but are uniformly overtaken by our method across all k.

Our framework's relative improvements are especially pronounced in the hardest setting-HumanEval at pass@1-indicating that the exclusionary filter prevents spurious or minor deviations from causing outright failures. Moreover, the advantage grows with increased sampling (pass@5), where our method's +2.08 pp lead on HumanEval and +4.93 pp on CodeXGLUE highlight its ability to refine successive proposals more effectively than a bare LLM. In the bug-fixing scenario, the average edit distance to the original code is dramatically reduced by our method (2.1 tokens vs. ∼5 for baseline LLMs), demonstrating that exclusionary reasoning not only improves correctness but also yields minimal, interpretable repairs-an essential property for educational and review contexts.

The empirical gains achieved by our pipeline translate into two key practical benefits. First, minimal-change outputs preserve the author's original code structure and style, enhancing readability and traceability of optimizations or fixes. Such behavior is critical in pedagogical settings, where learners must understand precisely what was changed and why. Second, the modular segmentation prevents collateral edits outside the target region, ensuring that unrelated code remains intact. Together, these properties position our framework as a robust and transparent tool for automated code review, teaching assistants, and production-grade refactoring systems.

4.5 Ablation study

To quantify the individual contribution of each core component in our framework, we conduct ablation experiments on the following three modules:

- **-w/o ER(Exclusionary Reasoning (ER) module):** filters out any proposed edits that are logically unnecessary, enforcing a minimal-change optimization objective.
- **-w/o CS(Code Segmentation (CS) module)**: partitions an input program into isolated segments (e.g., individual functions or blocks) to prevent cross-segment interference and to ensure each segment fits within the LLM's context window.
- **-w/o PP (Post-Processing (PP) module):** performs syntax verification, test-based correction (reverting or adjusting edits that break tests), and final formatting to guarantee 100% compilability and functional correctness.

Table 2 shows the results of the ablation study. We can see that removing the ER module degrades pass@1 by 4.47 pp on HumanEval (48.59% → 44.12%), by 5.61 pp on MBPP (75.84% → 70.23%), and by 4.39 pp on CodeXGLUE (75.37% → 70.98%). Similar drops occur at pass@3 and pass@5. This confirms that filtering out non-beneficial edits is essential: without ER, spurious or format-breaking changes proposed by the LLM propagate unchecked, reducing functional correctness. Ablating the CS module incurs the largest loss: pass@1 drops by 6.28 pp on HumanEval, 7.55 pp on

Table 2 Ablation Study

Datasets	HumanEval			MBPP			CodeXGLUE		
	pass@1	pass@3	pass@5	pass@1	pass@3	pass@5	pass@1	pass@3	pass@5
Our	48.59	73.17	76.64	75.84	85.32	86.41	75.37	85.22	87.56
- w/o ER	44.12	67.85	71.39	70.23	80.14	81.55	70.98	80.33	82.17
- w/o CS	42.31	66.10	69.02	68.29	78.47	80.09	68.18	78.00	80.44
- w/o PP	46.27	70.50	73.81	73.95	83.10	84.02	73.51	83.31	85.28

MBPP, and 7.19 pp on CodeXGLUE. Segmentation confines the LLM's context to individual code blocks, preventing context overflow and inadvertent edits to unrelated regions. Its removal forces the model to handle entire files at once, leading to context truncation and collateral modifications, which substantially harm accuracy. Omitting the PP module leads to a smaller but still notable decrease: pass@1 falls by 2.32 pp on HumanEval, 1.89 pp on MBPP, and 1.86 pp on CodeXGLUE. The PP stage's syntax verification and test-based rollback correct residual errors and prevent rare syntax or logic faults from affecting the final output, thereby ensuring maximal robustness.

These results demonstrate that each component of our pipeline is indispensable for achieving state-of-the-art performance. In particular, without segmentation the LLM's capacity is misapplied; without exclusionary filtering, the model's unconstrained creativity introduces errors; and without post-processing, residual faults undermine robustness. Together, the three modules form a cohesive framework that elevates raw LLM outputs into highly accurate, minimal-change optimizations suitable for both production and educational settings.

4.6 Practical deployment in educational settings

To illustrate how our LLM-based code optimization system could be integrated into real-world programming instruction, we envision the following deployment scenario. First, students submit their code assignments through an online platform (e.g., JupyterHub or CodeRunner), at which point the system analyzes the code and returns both optimized suggestions and an exclusionary reasoning log that explains why certain transformations were (or were not) applied. This interactive feedback loop allows learners to compare their original implementation with the refined version and to study the reasoning chain step by step.

Second, instructors can configure the system to collect anonymized performance metrics-such as error types, average improvement in execution time, or change in code readability-and visualize these data in dashboards within the LMS (e.g., Moodle or Canvas). Such analytics supports targeted in-class discussions and adaptive assignments based on common student difficulties.

Finally, the system exposes a RESTful API and plugin modules for seamless integration into mainstream educational platforms. For instance, a simple LTI (Learning Tools Interoperability) connector enables direct embedding of our optimization interface in any LTI-compliant LMS. This design ensures minimal setup overhead for educators and allows the system to be adopted either as a standalone tutor tool or as part of a broader course delivery pipeline.

5 Conclusion

This study illustrates that knowing when not to act is as critical for automated code optimisation as knowing how to act. By separating the decision to optimise from the act of rewriting, we prevent large language models from performing gratuitous or harmful edits, thereby improving first-attempt correctness and producing diffs that students can readily digest. The framework is lightweight, architecture-agnostic, and confers the greatest relative benefit to smaller, non-code-specialised models-suggesting that selective reasoning can compensate for baseline capability gaps. Beyond higher pass@k scores, the approach realigns LLM behaviour with instructional best practice, fostering trust and interpretability in educational settings. Future work should pair the gating mechanism with explicit natural-language rationales, extend the paradigm to multi-objective optimisation (performance, style, security), and explore its transfer to other pedagogical domains such as mathematical proof checking or essay revision.

Author Contributions Yi Rong: Methodology, Writing-original draft, Project administration, Data Curation, Formal analysis, Resources. Tianfeng Du: Data curation, Conceptualization, Methodology, Software, Investigation,. Roubing Li: Methodology, Funding acquisition, Validation. Wenting Bao: Supervision, Methodology, Resources, Funding acquisition, Writing–review & editing.

Funding This research was supported by the Quality Education Project of Department of Education in Anhui Province, China (Grant No. 2023jyxm0364).

Data Availability Statement The datasets used in this manuscript are publicly available datasets. Detailed information about these datasets is provided in Section 4.1 *Dataset* of this manuscript.

Declarations

Conflicts of Interest The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Abbassi AA, Da Silva L, Nikanjam A, Khomh F (2025) Unveiling inefficiencies in llm-generated code: Toward a comprehensive taxonomy. arXiv:2503.06327

Abouelenin A, Ashfaq A, Atkinson A, Awadalla H, Bach N, Bao J, Benhaim A, Cai M, Chaudhary V, Chen C et al (2025) Phi-4-mini technical report: Compact yet powerful multimodal language models via mixture-of-loras. arXiv:2503.01743

Alyoshyna Y (2024) Ai in programming education: Automated feedback systems for personalized learning. B.S. thesis, University of Twente

Anagnostopoulos K (2024) Using large language models for private library code generation

Austin J, Odena A, Nye M, Bosma M, Michalewski H, Dohan D, Jiang E, Cai C, Terry M, Le Q et al (2021) Program synthesis with large language models. arXiv:2108.07732

Baichuan (2023) Baichuan 2: Open large-scale language models. arXiv:2309.10305

Balse R, Kumar V, Prasad P, Warriem JM (2023) Evaluating the quality of llm-generated explanations for logical errors in cs1 student programs. In: Proceedings of the 16th annual ACM India compute conference, pp 49–54

Berdal FK (2024) Supporting selective peer code review in education: Evaluating code-selection methods. Master's thesis, NTNU

Bhatt MCH, Bhadka HB (2013) Peephole optimization technique for analysis and review of compile design and construction. IOSR J Comput Eng (IOSR-JCE) 9(4):80–86

Borchers C, Shou T (2025) Can large language models match tutoring system adaptivity? a benchmarking study. arXiv:2504.05570

Brooks G, Hansen GJ, Simmons S (1992) A new approach to debugging optimized code. ACM SIGPLAN Notices 27(7):1–11

Cai Y, Hou Z, Sanan D, Luan X, Lin Y, Sun J, Dong JS (2025) Automated program refinement: Guide and verify code large language model with refinement calculus. In: Proceedings of the ACM on programming languages 9(POPL):2057–2089

Cheney N, Bongard J, SunSpiral V, Lipson H (2018) Scalable co-optimization of morphology and control in embodied machines. J Royal Soc Interface 15(143):20170937

Chowdhury MSS, Chowdhury MNUR, Neha FF, Haque A (2024) Ai-powered code reviews: Leveraging large language models. In: 2024 International conference on signal processing and advance research in computing (SPARC), IEEE, vol. 1, pp 1–6

Du X, Liu M, Wang K, Wang H, Liu J, Chen Y, Feng J, Sha C, Peng X, Lou Y (2024) Evaluating large language models in class-level code generation. In: Proceedings of the IEEE/ACM 46th international conference on software engineering, pp 1–13

Finnie-Ansley J, Denny P, Becker BA, Luxton-Reilly A, Prather J (2022) The robots are coming: Exploring the implications of openai codex on introductory programming. In: Proceedings of the 24th australasian computing education conference, pp 10–19

Gross T, Steenkiste P (1990) Structured dataflow analysis for arrays and its use in an optimizing compiler. Softw: Pract Exper 20(2):133–155

Gu X, Renaud JE, Ashe LM, Batill SM, Budhiraja AS, Krajewski LJ (2002) Decision-based collaborative optimization. J Mech Des 124(1):1–13

Guarnieri S, Livshits VB (2009) Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. USENIX Security Symposium 10:78–85

Gupta AK, Venkatesha GG, Singh K, Shah S, Goel O, Jain S (2024) Enhancing cascading style sheets efficiency and performance through ai-based code optimization. In: 2024 13th International conference on system modeling & advancement in research trends (SMART), IEEE, pp 306–311

Hao S, Gu Y, Luo H, Liu T, Shao X, Wang X, Xie S, Ma H, Samavedhi A, Gao Q et al (2024) Llm reasoners: New evaluation, library, and analysis of step-by-step reasoning with large language models. arXiv:2404.05221

Hundhausen CD, Agrawal A, Agarwal P (2013) Talking about code: Integrating pedagogical code reviews into early computing courses. ACM Trans Comput Educ (TOCE) 13(3):1–28

Inan H, Upasani K, Chi J, Rungta R, Iyer K, Mao Y, Tontchev M, Hu Q, Fuller B, Testuggine D et al (2023) Llama guard: Llm-based input-output safeguard for human-ai conversations. arXiv:2312.06674

Jiang J, Wang F, Shen J, Kim S, Kim S (2024) A survey on large language models for code generation. arXiv:2406.00515

Korini K, Bizer C (2025) Evaluating knowledge generation and self-refinement strategies for llm-based column type annotation. arXiv:2503.02718

Latibari BS, Nazari N, Chowdhury MA, Gubbi KI, Fang C, Ghimire S, Hosseini E, Sayadi H, Homayoun H, Salehi S et al (2024) Transformers: A security perspective. IEEE Access

Latif E, Parasuraman R, Zhai X (2024) Physicsassistant: An llm-powered interactive learning robot for physics lab investigations. In: 2024 33rd IEEE International conference on robot and human interactive communication (ROMAN), IEEE, pp 864–871

Lee S, Lee J, Park CY, Min SL (2007) Selective code transformation for dual instruction set processors. ACM Trans Embedded Comput Syst (TECS) 6(2):10

Lertbanjongngam S, Chinthanet B, Ishio T, Kula RG, Leelaprute P, Manaskasemsak B, Rungsawang A, Matsumoto K (2022) An empirical evaluation of competitive programming ai: A case study of alphacode. In: 2022 IEEE 16th International workshop on software clones (IWSC), IEEE, pp 10–15

Li D, Murr L (2024) Humaneval on latest gpt models–2024. arXiv:2402.14852

Li J, Rabbi F, Cheng C, Sangalay A, Tian Y, Yang J (2024) An exploratory study on fine-tuning large language models for secure code generation. arXiv:2408.09078

Liu Z, Zeng R, Wang D, Peng G, Wang J, Liu Q, Liu P, Wang W (2024) Agents4plc: Automating closed-loop plc code generation and verification in industrial control systems using llm-based agents. arXiv:2410.14209

Luburić N, Dorić L, Slivka J, Vidaković D, Grujić K-G, Kovačević A, Prokić S (2025) An intelligent tutoring system to support code maintainability skill development. IEEE Trans Learn Technol

Lu S, Guo D, Ren S, Huang J, Svyatkovskiy A, Blanco A, Clement C, Drain D, Jiang D, Tang D et al (2021) Codexglue: A machine learning benchmark dataset for code understanding and generation. arXiv:2102.04664

Lu J, Yu L, Li X, Yang L, Zuo C (2023) Llama-reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning. In: 2023 IEEE 34th International symposium on software reliability engineering (ISSRE), IEEE, pp 647–658

Marwan S, Lytle N, Williams JJ, Price T (2019) The impact of adding textual explanations to next-step hints in a novice programming environment. In: Proceedings of the 2019 ACM conference on innovation and technology in computer science education, pp 520–526

Ma Q, Shen H, Koedinger K, Wu T (2023) Hypocompass: Large-language-model-based tutor for hypothesis construction in debugging for novices. arXiv:2310.05292

McIntosh S, Kamei Y, Adams B, Hassan AE (2016) An empirical study of the impact of modern code review practices on software quality. Empirical Softw Eng 21:2146–2189

Muennighoff N, Liu Q, Zebaze A, Zheng Q, Hui B, Zhuo TY, Singh S, Tang X, Von Werra L, Longpre S (2023) Octopack: Instruction tuning code large language models. In: NeurIPS 2023 workshop on instruction tuning and instruction following

Nazaretsky T, Ariely M, Cukurova M, Alexandron G (2022) Teachers' trust in ai-powered educational technology and a professional development program to improve it. British J Educ Technol 53(4):914–931

Paixao M, Krinke J, Han D, Ragkhitwetsagul C, Harman M (2019) The impact of code review on architectural changes. IEEE Trans Softw Eng 47(5):1041–1059

Raji ID, Scheuerman MK, Amironesei R (2021) You can't sit with us: Exclusionary pedagogy in ai ethics education. In: Proceedings of the 2021 ACM conference on fairness, accountability, and transparency, pp 515–525

Shin J, Nam J (2021) A survey of automatic code generation from natural language. J Inf Process Syst 17(3):537–555

Shinn N, Cassano F, Gopinath A, Narasimhan K, Yao S (2023) Reflexion: Language agents with verbal reinforcement learning. Adv Neural Inf Process Syst 36:8634–8652

Singh A (2025) Evaluating the transparency and explainability of llm-based educational systems. Available at SSRN 5198565

Tan SH, Roychoudhury A (2015) relifix: Automated repair of software regressions. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, IEEE, vol. 1, pp 471–482

Team G, Mesnard T, Hardin C, Dadashi R, Bhupatiraju S, Pathak S, Sifre L, Rivière M, Kale MS, Love J et al (2024) Gemma: Open models based on gemini research and technology. arXiv:2403.08295

Vaidya J, Asif H (2023) A critical look at ai-generate software: Coding with the new ai tools is both irresistible and dangerous. Ieee Spectrum 60(7):34–39

Velaga SP (2020) Ai-assisted code generation and optimization: Leveraging machine learning to enhance software development processes. Int J Innovations Eng Res Technol 7(09):177–186

Wadhwa N, Pradhan J, Sonwane A, Sahu SP, Natarajan N, Kanade A, Parthasarathy S, Rajamani S (2024) Core: Resolving code quality issues using llms. Proceed ACM Softw Eng 1(FSE):789–811

Wang P, Bai S, Tan S, Wang S, Fan Z, Bai J, Chen K, Liu X, Wang J, Ge W et al (2024) Qwen2-vl: Enhancing vision-language model's perception of the world at any resolution. arXiv:2409.12191

Wang T, Díaz DV, Brown C, Chen Y (2023) Exploring the role of ai assistants in computer science education: Methods, implications, and instructor perspectives. In: 2023 IEEE Symposium on visual languages and human-centric computing (VL/HCC), IEEE, pp 92–102

Wang K, Zhu C, Celik A, Kim J, Batory D, Gligoric M (2018) Towards refactoring-aware regression test selection. In: Proceedings of the 40th international conference on software engineering, pp 233–244

Wu M, Goodman N, Piech C, Finn C (2021) Prototransformer: A meta-learning approach to providing student feedback. arXiv:2107.14035

Xing Y, Weng J, Wang Y, Sui L, Shan Y, Wang Y (2019) An in-depth comparison of compilers for deep neural networks on hardware. In: 2019 IEEE International conference on embedded software and systems (ICESS), IEEE, pp 1–8

Xu FF, Vasilescu B, Neubig G (2022) In-ide code generation from natural language: Promise and challenges. ACM Trans Softw Eng Methodol (TOSEM) 31(2):1–47

Yang Z, Meng Z, Zheng X, Wattenhofer R (2024) Assessing adversarial robustness of large language models: An empirical study. arXiv:2405.02764

Yang G, Zhou Y, Chen X, Zhang X, Zhuo TY, Chen T (2024) Chain-of-thought in neural code generation: From and for lightweight language models. IEEE Trans Softw Eng

Young A, Chen B, Li C, Huang C, Zhang G, Zhang G, Wang G, Li H, Zhu J, Chen J et al (2024) Yi: Open foundation models by 01. ai. arXiv:2403.04652

Zhao Z, Sun J, Cai C-H, Wei Z (2024) Code generation using self-interactive assistant. In: 2024 IEEE 48th annual computers, software, and applications conference (COMPSAC), IEEE, pp 2347–2352

Zhao Z, Sun J, Cai C-H, Wei Z (2024) Code generation using self-interactive assistant. In: 2024 IEEE 48th Annual computers, software, and applications conference (COMPSAC), IEEE, pp 2347–2352

Zheng Z, Ning K, Wang Y, Zhang J, Zheng D, Ye M, Chen J (2023) A survey of large language models for code: Evolution, benchmarking, and future trends. arXiv:2311.10372

Zheng H, Shen L, Tang A, Luo Y, Hu H, Du B, Tao D (2023) Learn from model beyond fine-tuning: A survey. arXiv:2310.08184

Zheng Y, Yang Y, Tu H, Huang Y (2024) Code-survey: An llm-driven methodology for analyzing large-scale codebases. arXiv:2410.01837