

A Systematic Study of Time Limit Exceeded Errors in Online Programming Assignments

JIALU ZHANG^{*†}, University of Waterloo, Canada
JIALIANG GU[†], George Mason University, USA
WANGMEIYU ZHANG, Chinese University of Hong Kong, Shenzhen, China
JOSÉ PABLO CAMBRONERO, Google, USA
JOHN KOLESAR, Yale University, USA
RUZICA PISKAC, Yale University, USA
DAMING LI, Independent Researcher, USA
HANYUAN SHI, Independent Researcher, China

Online programming platforms such as Codeforces and LeetCode attract millions of users seeking to learn to program or refine their skills for industry interviews. A major challenge for these users is the Time Limit Exceeded (TLE) error, triggered when a program exceeds the execution time bound. Although designed as a performance safeguard, TLE errors are difficult to resolve: error messages provide no diagnostic insight, platform support is minimal, and existing debugging tools offer little help. As a result, many users abandon their submissions after repeated TLE failures.

This paper presents the first large-scale empirical study of TLE errors in online programming. We manually analyzed 1000 Codeforces submissions with TLE errors, classified their root causes, and traced how users attempted to fix them. Our analysis shows that TLE errors often arise not only from inefficient algorithms but also from infinite loops, improper data structure use, and inefficient I/O, challenging the conventional view that TLEs are purely performance issues.

Guided by these findings, we introduce Nettle, the first automated repair tool specifically designed for TLE errors, and Nettle-Eval, the first framework for evaluating TLE repairs. Integrating LLMs with targeted automated feedback generated by the compiler and test cases, Nettle produces small, correct code edits that eliminate TLEs while preserving functionality. Evaluated on the same 1000 real-world cases, Nettle achieves a 98.5% fix rate, far exceeding the strongest LLM baseline, and all of its repairs pass both Nettle-Eval and the platform's official checker, confirming the reliability of our framework.

ACM Reference Format:

Jialu Zhang, Jialiang Gu, Wangmeiyu Zhang, José Pablo Cambronero, John Kolesar, Ruzica Piskac, Daming Li, and Hanyuan Shi. 2025. A Systematic Study of Time Limit Exceeded Errors in Online Programming Assignments. 1, 1 (October 2025), 21 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

^{*}Corresponding author.

[†]Equal contribution.

Authors' addresses: Jialu Zhang, University of Waterloo, Waterloo, Canada, jialu.zhang@uwaterloo.ca; Jialiang Gu, George Mason University, Fairfax, USA, jgu7@gmu.edu; Wangmeiyu Zhang, Chinese University of Hong Kong, Shenzhen, Shenzhen, China, 123090825@link.cuhk.edu.cn; José Pablo Cambronero, Google, Atlanta, USA, josepablocam@gmail.com; John Kolesar, Yale University, New Haven, USA, john.kolesar@yale.edu; Ruzica Piskac, Yale University, New Haven, USA, ruzica.piskac@yale.edu; Daming Li, Independent Researcher, Mountain View, USA, damingliyale22@gmail.com; Hanyuan Shi, Independent Researcher, Hangzhou, China, shihanyuan1995@gmail.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2025 Association for Computing Machinery.

XXXX-XXXX/2025/10-ART \$15.00

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 Introduction

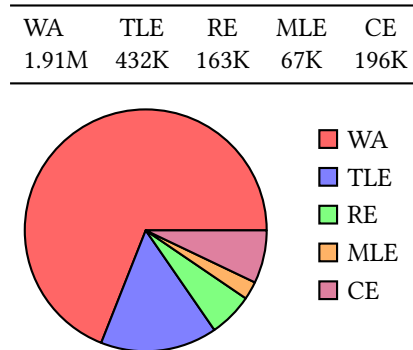
Online programming platforms such as Codeforces, LeetCode, and AtCoder play a crucial role in software development education and industry recruitment [4, 14]. These systems engage millions of users and evaluate code not only for functional correctness but also for computational efficiency. Each submission receives an automated verdict indicating the type of failure or success. Compared to the `Wrong Answer` (WA) verdict, where there is a bug in the program that causes incorrect outputs, the `Time Limit Exceeded` (TLE) verdict occurs when a submission exceeds the time budget (typically 1–2 seconds) despite producing no runtime or semantic errors. Table 1 shows that this error is particularly prevalent, appearing in 18% of the 2.7 million submissions we sampled from Codeforces, and is especially frustrating for platform users.

Despite its frequency and relevance to learning practical efficiency constraints, TLE remains underexamined in programming education research. Unlike `Wrong Answer` (WA) or `Runtime Error` (RE), which often provide outputs or stack traces, TLE offers only a one-line message, giving little guidance for debugging. Consequently, users are far less successful at resolving TLEs compared to other error types: while 31.7% of WA submissions are eventually corrected, only 18.9% of TLEs are fixed from our sampled 2.7 million submissions. Unlike WA or RE, there is no partial output or error localization. Users are left to speculate whether the error comes from asymptotic inefficiency, data structure misuse, an infinite loop, I/O bottlenecks, or some other cause. Worse still, the test cases are hidden and unreproducible, making root cause analysis nearly impossible. TLE failures affect both novices and experts, often leading to excessive iteration. The median number of resubmissions from the first TLE to eventual success exceeds seven from our sampled Codeforces submissions, translating into wasted user time, reduced motivation, and blocked learning outcomes.

Existing tools fail to address TLE errors. Profilers and static analyzers lack access to platform-specific inputs or runtime behavior[48]. Existing automated program repair (APR) techniques[15, 38] assume incorrect functionality and rely on failing test oracles, neither of which may be available for TLE. Similarly, recent advances in LLM-based code repair (e.g., CodeBERT[10], Codex[5]) provide generic suggestions that ignore performance characteristics. As a result, programmers often resort to guesswork or brute-force rewriting or simply abandon the problem. Community forums offer little help, often responding with generic advice to optimize the algorithm or even misleading advice that “a rule of thumb is that TLE errors are caused by inefficient algorithms.”¹

From a research perspective, TLE represents a blind spot in existing tools and techniques. Unlike functional faults, TLE arises from the interaction of algorithmic complexity, data-dependent execution paths, language-level I/O costs, and subtle implementation inefficiencies, which together complicate diagnosis and repair. Prior work has pursued efficiency improvements, such as program transformations to optimize algorithms [2] and container substitutions guided by complexity

Table 1. Error type distribution in 2.7M Codeforces submissions. WA = Wrong Answer, TLE = Time Limit Exceeded, RE = Runtime Error, MLE = Memory Limit Exceeded, CE = Compilation Error.



¹<https://stackoverflow.com/questions/59227574/how-to-avoid-tle-when-array-is-used-to-store-1-105-integer-entries>

analysis [45], but these efforts evaluate isolated optimizations rather than offering a systematic, empirical account of TLE as a distinct failure mode. This gap motivates our study.

We present the first large-scale empirical study of TLE errors in online programming platforms. We manually analyze 1,000 real-world TLE submissions from Codeforces, a popular online programming platform. We manually categorize their root causes, and we identify five recurring patterns, including inefficient algorithms, infinite loops, and improper I/O. These findings provide the first taxonomy of TLE failures at scale and expose key shortcomings for addressing these errors using current programming and tooling practices.

Building on these insights, we design and implement *Nettle*, the **N**ew **E**xaminer **T**argeting **T**LE errors. Unlike prior systems that pursue efficiency gains only in semantically equivalent programs, *Nettle* directly addresses TLE cases that manifest as timeouts and may also contain functional errors. Our approach leverages large language models to repair code iteratively, generate candidate fixes, test them, and refine subsequent attempts based on prior feedback. To achieve this, our tool consists of two tightly coupled stages, *reasoning* and *judging*, which interact in an iterative loop to converge on efficient and correct repairs.

To evaluate *Nettle*, we construct a benchmark of 1,000 real-world TLE submissions from Codeforces and implement *Nettle-Eval*, an evaluation framework designed to assess both repair correctness and time-efficiency under strict resource limits. On this benchmark, *Nettle* achieves a fix rate of 98.5%, substantially surpassing competitive baselines not only in repair success but also repair quality (small patch size). For example, *Nettle* reaches a 95.8% fix rate, far above the strongest LLM-based baseline at 88.3%, which approximates the best repair performance a user could achieve using general-purpose LLMs without tool-specific guidance. For repair quality, *Nettle*'s edits are the same size as the baselines or even smaller.

To validate the reliability of *Nettle-Eval* itself, we submitted all repaired programs back to Codeforces for re-verification against the platform's official ground truth. We found that every repair accepted by *Nettle-Eval* was also accepted by Codeforces, confirming the correctness of our local evaluation. In addition, we found one case where *Nettle-Eval* identified a `Wrong Answer` that Codeforces had marked as `Accepted`; after our report, the Codeforces maintainers confirmed the issue. This result further demonstrates that *Nettle-Eval* can reveal corner cases overlooked by the platform's tests. By design, *Nettle-Eval* adopts a conservative stance: because runtime environments and hardware conditions inevitably differ, it enforces stricter timing thresholds and flags borderline cases as TLE rather than accepting them. Together, these results demonstrate both the feasibility of automated TLE repair and the robustness of our evaluation framework, highlighting its potential to advance efficiency-aware program repair for online programming education and technical recruitment.

We make three core contributions:

- We present the first large-scale empirical study of Time Limit Exceeded (TLE) errors, analyzing 1,000 Codeforces submissions and identifying five recurring root causes along with their distinctive repair challenges.
- We introduce *Nettle*, the first automated repair tool explicitly targeting TLE errors, powered by large language models, together with *Nettle-Eval*, the first evaluation framework designed to validate both correctness and efficiency of TLE fixes systematically.
- We evaluate *Nettle* on 1,000 real-world TLE submissions and show that it achieves a fix rate of 98.5%, consistently outperforming strong LLM-based baselines both by being more accurate and by producing more compact repairs with faster convergence.

The remainder of the paper is structured as follows. Section 2 walks through multiple motivating examples of real TLEs. Section 3 provides a study of TLE patterns. Section 4 describes our approach

in detail. Section 5 provides experimental results on our dataset of Codeforces programs with TLE errors. Section 6 details further discussions and limitations. We discuss related work in Section 7. Finally, we conclude the paper in Section 8.

2 Motivation

2.1 What Are TLE Errors?

Time Limit Exceeded (TLE) errors are a distinctive failure mode in online programming problems. Unlike syntax or semantic errors, TLE indicates that a program exceeds the platform’s runtime constraints. This distinction makes TLE particularly elusive to diagnose and fix.

Figure 1 illustrates a real-world TLE example from Codeforces². The original solution uses Python’s `startswith` function on a sliced string to detect the pattern “AB”. Although functionally correct, repeated slicing incurs $O(n^2)$ overhead on large strings. A more efficient solution compares adjacent characters directly.

Despite its apparent simplicity, this fix is nontrivial to uncover. Traditional fault-localization techniques fail because both inefficient and efficient versions execute identical control paths. Profilers can reveal hotspots but offer no concrete repair strategy, and online platforms typically hide the TLE-triggering inputs. Most critically, resolving the issue requires language-specific expertise that many learners, especially novices, lack: string slicing in Python has a hidden $O(n^2)$ cost. These challenges illustrate why TLE errors persist as a uniquely difficult class of programming failures.

```
# TLE-inducing version
for i in range(len(s) - 1):
    if s[i:].startswith('AB'):
        abIndexes.append(i)

# Efficient fix
for i in range(len(s) - 1):
    if s[i] == 'A' and s[i + 1] == 'B':
        abIndexes.append(i)
```

Fig. 1. TLE-inducing vs. optimized string “AB” check

2.2 What Is Behind the “Time Limit Exceed” Error Message?

TLE Does Not Preclude Semantic Incorrectness. TLE is itself a vague error message. A common belief is that it indicates a semantically correct program that merely exceeds the time limit. To test this assumption, we randomly sampled and analyzed 1000 real-world TLE submissions from Codeforces by executing them locally under relaxed constraints (extended runtime) and with compiler optimizations, observing their outputs across varying input sizes.

As shown in Table 2, 516 programs produced correct outputs on small inputs but had TLE on larger inputs even under relaxed constraints. However, 176 programs still produced *incorrect* outputs even when given additional runtime, revealing underlying semantic bugs. In addition, 152 programs passed only after enabling advanced compiler optimizations in `pp3`, demonstrating that while such techniques offer modest performance gains, they fall far short of addressing the TLE problem in general.

Table 2. Outcome distribution for 1000 real-world TLE submissions from Codeforces by executing them locally under relaxed constraints.

Outcome	Count
Correct	152
Wrong Answer	176
TLE	516
Other Verdicts	156

²<https://codeforces.com/contest/550/submission/78561281>

2.3 Why Are TLE Errors Challenging to Fix?

Limitations of Local Stress Testing. Stress testing generates large random inputs locally to evaluate runtime performance. While useful in certain settings, this approach is often insufficient for diagnosing TLE errors, as many inefficient algorithms behave acceptably on random inputs but degrade on edge cases (e.g., quicksort on sorted arrays). Furthermore, generating valid large-scale inputs for complex problems, such as connected graphs with tens of thousands of edges, demands problem-specific knowledge and tooling.

Tooling Support Is Minimal. Traditional debugging and program repair tools are ill-suited for TLE errors. General-purpose debuggers (gdb), refactoring assistants (e.g., Refactory[17]), and APR tools (e.g., GenProg[22], Prophet[25], PyDex[54]) rely on observable incorrect outputs. For example, APR tools typically locate faults by comparing passing and failing test cases and performing semantic patching. This approach fails for TLE. Programs may produce the correct output but simply run too slowly, making them indistinguishable from correct programs under conventional fault localization.

In practice, users turn to forums like StackOverflow for help but find little actionable feedback:

- “My Java code is getting Time Limit Exceeded, whereas the same logic in C++ doesn’t.”[link]
- “Optimize Python code to read various inputs: Time Limit Exceeded.” [link]
- “How to solve timeout termination issue (Time Limit Exceeded) while using for loop in JavaScript?” [link]

Such questions often go unanswered or receive generic replies (“find an algorithm that doesn’t need to do 100 million things”³) without diagnosing the actual bottleneck. Without tooling that connects performance traces to code locations, users are left guessing.

For example, in this Stack Overflow post, a user provided a detailed problem description and asked:⁴

“How can I overcome a TLE error?”

The only response was a code dump without any explanation. When the original poster followed up with:

“I’m still getting TLE, but otherwise I would actually prefer to try and fix my original code and understand how to reduce the computational time.”

Nonetheless, no further reply was given. This case illustrates that TLE issues are often mishandled: an incorrect answer with no reasoning is worse than no answer at all, as it misleads learners and offers little educational value. Such examples highlight the urgent need for both an automated repair tool specifically designed for TLE errors and a trustworthy TLE validation framework.

3 Understanding TLE Errors

To understand the underlying causes of TLE errors, we manually analyzed real-world submissions from the Codeforces platform. We selected representative problems across difficulty levels and examined failing programs that triggered TLE or related verdicts, together with their successful repairs. Our analysis uncovered five recurring failure patterns, each reflecting a distinct inefficiency mechanism.

Inefficient Algorithm. The most frequent cause of TLE is excessive time complexity from brute-force enumeration.⁵ Example:

³<https://stackoverflow.com/questions/54496861/how-do-i-get-rid-of-the-tle-error-in-python>

⁴<https://stackoverflow.com/questions/73529959/how-can-i-overcome-this-tle>

⁵<https://codeforces.com/contest/1/submission/24609270>

```
for n in range(2, 100000000, 2):  
    if x * n >= a1:  
        print(n)  
        break
```

This linear scan performs billions of iterations. The correct approach would use mathematical rounding to compute the minimal multiplier in $O(1)$.

Inefficient Handling. Redundant calls to expensive operations (e.g., `strlen`, `memset`) or repeatedly recompute or accumulate quantities that could be derived directly.⁶ Example:

```
while plate_square < squares_square:  
    plate_square = plate_square + (int(plate_size) ** 2)  
    num_plates += 1
```

The code simulates tile accumulation by iterative addition rather than using direct division and ceiling operations, incurring unnecessary runtime overhead.

Improper I/O. Format mismatches or type errors that result in incorrect loop bounds or hanging input calls.⁷ Example:

```
n, m, a = input().split()  
print((n//a + (n%a != 0)) * (m//a + (m%a != 0)))
```

Since the input tokens remain strings, integer operations such as `//` and `%` fail or hang. Proper type conversion via `map(int, ...)` is essential before computation.

Infinite Loop. Logical flaws in loop conditions often lead to nontermination.⁸ Example:

```
while mayor!=0 or mayor<0:  
    mayor = mayor - c[2]
```

The use of logical `or` ensures the condition remains true even when `mayor` becomes negative. Combined with a zero or negative decrement, the loop never exits.

Memory Access. Out-of-bounds writes or reads may produce undefined behavior, sometimes manifesting as silent TLEs.⁹ Example:

```
int a[100];  
for (int i = 0; i < n; i++) {  
    cin >> a[i]; // overflow if n > 100  
}
```

When `n` exceeds array bounds, memory corruption interferes with I/O buffers or loop control, producing a silent hang rather than a segmentation fault.

Summary. TLE errors are multifaceted: they may stem from asymptotic inefficiency, unoptimized control flow, input mismanagement, or low-level memory misuse. These findings highlight that TLE is not purely an algorithmic failure but a spectrum of performance and correctness issues. Recognizing these fine-grained causes enables automated feedback systems to generate targeted, pedagogically meaningful guidance beyond simple runtime profiling.

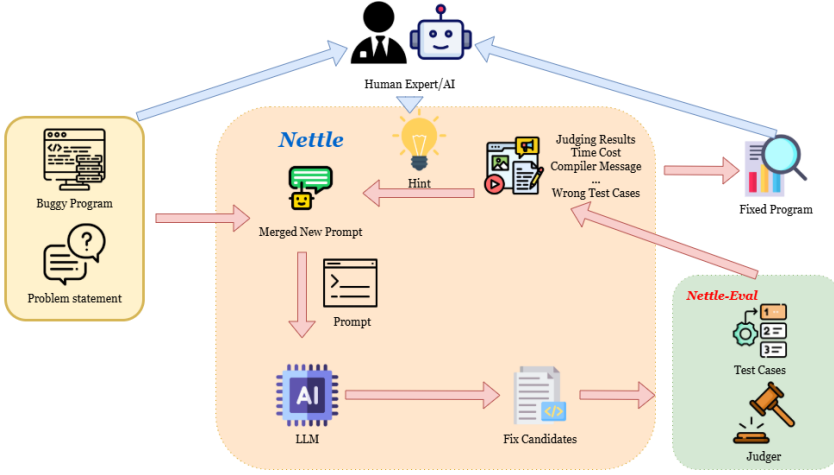


Fig. 2. Overview of the system pipeline. The reasoning stage (Nettle) constructs structured prompts with optional hints and generates fix candidates via the LLM, while the judging stage (Nettle-Eval) executes them under resource limits and provides feedback (e.g., TLE, WA, test case) to guide further reasoning.

4 Nettle's System Design

As shown in Figure 2, we propose an automated repair framework to address *time-limit exceeded* (TLE) errors in programming assignments. The system takes as input an incorrect program submission S for a given problem P , a set of visible input–output test cases, and a judge (Oracle). In the classical setting, a TLE program S may produce an output on certain sample inputs (not necessarily correct), but fails to terminate within the time limit on other inputs. Our framework aims to transform S into a corrected program S' , such that it (1) terminates within the time bounds enforced by the evaluator (provided by the problem description), (2) produces the expected outputs on all visible sample tests, and (3) generalizes to unseen inputs by both avoiding TLE and matching the expected outputs. To achieve this, our tool consists of two tightly coupled stages—*reasoning* and *judging*—which interact iteratively.

In the reasoning stage, an LLM-driven engine proposes k candidate repairs using prompts that integrate multiple sources of context. In the judging stage, a customized sandbox executes each candidate under strict resource limits to assess correctness and performance. The evaluator then produces structured feedback (e.g., which tests passed or failed, execution time, memory usage, error type) and returns it to the reasoning stage to guide the next iteration. Among the candidates, the one passing the most tests is retained while the others are discarded, effectively implementing a beam search with width one. The process repeats until a candidate passes all visible tests within the limits or the maximum iteration budget is reached. Inspired by how human programmers iteratively debug performance issues, the system alternates between reasoning, modification, and validation, with early termination ensuring efficiency when a valid patch is found before reaching the iteration limit.

⁶<https://codeforces.com/contest/1/submission/24330765>

⁷<https://codeforces.com/contest/1/submission/23397241>

⁸<https://codeforces.com/contest/1/submission/22168665>

⁹<https://codeforces.com/contest/490/submission/179836963>

4.1 Reasoning Stage

The reasoning module generates candidate repairs for TLE submissions. It aggregates context into a structured prompt for the LLM, including (1) the problem description Q , (2) the buggy code S , (3) diagnostic information (e.g., static analysis warnings, compiler messages, runtime profiles), (4) optional hints, and (5) prior repair history. The prompt formatter assembles these into a coherent query. For example, a prompt may include the problem description, the student's code (with comments removed), and an instruction such as: `Task: Fix the TLE errors in the code so that it finishes within the time limit.`

Two types of hints are supported. A *general* hint corresponds to an error class; for instance, for a TLE error: “The code may contain an infinite loop, an inefficient algorithm, or slow input/output.” General hints are produced automatically by *Nettle*, derived from the model's intermediate reasoning steps. A *problem-specific* hint, in contrast, targets a particular assignment. For example, in Codeforces 102B¹⁰, a suitable hint is “Use string input instead of integer input in Python.” To ensure consistency and pedagogical value, we manually author these problem-specific hints and reuse them across all submissions for the same problem.

Once the prompt is prepared, the LLM (candidate synthesizer) generates k candidates per iteration ($C_t = \{P_{t,1}, \dots, P_{t,k}\}$), sampled with different parameters (temperature, top- k). This allows exploration of different optimization strategies: one candidate may reuse the precomputed values, another may use a more efficient data structure. The LLM is encouraged to make minimal edits (via instructions like “fix the bug” rather than “rewrite the code”), preserving functionality. Each response is logged together with prompts and feedback, forming a memory across iterations. Feedback from the judging stage is merged into subsequent prompts, guiding the model away from repeated mistakes and towards convergence.

4.2 Judging Stage

One key challenge in addressing TLE is ensuring that a repaired program can pass the rigorous correctness and performance checks imposed by online platforms such as Codeforces. To tackle this, we introduce the first evaluation framework, *Nettle-Eval*, centered on TLE resolution. *Nettle-Eval* serves as a correctness-and-performance oracle: it executes candidate patches against a set of representative inputs and uses verdicts such as `Accepted`, `Wrong Answer`, or `Time Limit Exceeded` to guide further repair in the reasoning stage.

The judging stage is implemented as a Docker sandbox running under strict resource limits. Candidates are compiled (or interpreted) and executed on the test suite with CPU time limit T and memory limit M . Watchdogs enforce timeouts: if execution exceeds $T \cdot f$ seconds (with f a factor, typically 3–5), the process is killed and CPU time recorded. Memory is constrained to M MB, with overuse labeled `Memory Limit Exceeded`. The sandbox is deterministic, side-effect free, and reproducible. Moreover, this isolation prevents possible harmful code generated by LLMs from affecting the host system.

The judge produces a structured verdict with standard contest labels: `AC` (Accepted), `WA` (Wrong Answer), `RE` (Runtime Error), `CE` (Compilation Error), `TLE` (Time Limit Exceeded), and `MLE` (Memory Limit Exceeded). Diagnostics include failing test indices (e.g., “test 7 timed out”), execution time, memory usage, and captured error traces. Functional correctness is assessed by exact output comparison. *Nettle-Eval* supports an optional feature called *special judges* that can check for more complex properties, such as inclusion in multivalued sets. For example, Codeforces 584A

¹⁰<https://codeforces.com/problemset/problem/102/B>

¹¹ requires any n -digit number divisible by t . We implement a custom checker as follows to support multiple valid outputs:

Listing 1. Special judge for Codeforces 584A.

```
def _584A(file1, file2, file3):
    def trans(s):
        s = list(map(lambda s: s.rstrip(), s.split('\n')))
        if s[-1] == '':
            s = s[:-1]
        return s

    with open(file1) as f:
        f1 = trans(f.read())
    with open(file2) as f:
        f2 = trans(f.read())
    if not len(f1) == 1:
        return True
    f1 = f1[0]
    f2 = f2[0]
    if int(f2) == -1:
        return f1 != f2

    with open(file3) as f:
        line = f.readline()
        n, t = map(int, line.split())

    if not f1.isdigit():
        return True
    if len(f1) != n:
        return True
    if f1[0] == '0':
        return True
    s1 = int(f1)
    if s1 % t == 0:
        return False
    else:
        return True
```

After passing the judging stage, the output is collected, analyzed, and encoded into natural-language prompts for the next reasoning round. For example:

After TLE (on visible cases):

Time Limit Exceeded. The program may contain an infinite loop or an inefficient algorithm.

After WA (on visible cases):

Wrong Answer. For input X, expected Y but got Z. Check the logic and ensure correctness.

¹¹<https://codeforces.com/problemset/problem/584/A>

After RE (on visible cases):

Runtime Error. The program crashed (e.g., null pointer).
Fix the underlying runtime issue.

Therefore, the system functions as an iterative loop: the reasoning stage proposes candidate patches, the judging stage executes and evaluates them, and the candidate passing the largest number of tests is retained for the next round (ties are broken randomly). Feedback from the evaluator then guides the next iteration. The process continues until a patch S' passes all tests within the resource limits or the maximum depth budget is exhausted. Empirically, successful repairs are typically found within only a few iterations. By systematically combining LLM-based synthesis with rigorous sandbox evaluation, our framework ensures that only patches that *truly resolve TLE while preserving correctness* are accepted, closely mirroring how human programmers iteratively debug performance issues.

To sum up, this judger design ensures that a repaired program passes a rigorous checking that emulates that of an online platform. We have wrapped and released our judger as a standalone gym, providing a dedicated testing framework to support future research on TLE.

5 Evaluation

We evaluate *Nettle* by answering the following research questions:

- **RQ1: Repair Effectiveness.** How effective is *Nettle* in repairing real-world TLE errors compared to state-of-the-art LLM-based repair baselines, in terms of success rate and repair size?
- **RQ2: Role of Contextual Information.** How does the availability of different contextual signals (e.g., general error reasons, problem-specific hints) influence the repair performance of *Nettle*?
- **RQ3: Effectiveness of *Nettle-Eval*.** How effective is *Nettle-Eval* in assessing repair candidates within *Nettle*? In particular, is the proposed change a genuine repair of the original code, or merely a plausible but incorrect repair?

5.1 Implementation Setup

We implemented *Nettle* in Python, leveraging open-source libraries for code analysis and repair. The system is designed to address both syntactic and semantic TLE errors in a unified framework.

For candidate ranking, we used log-probability scoring from the language model component (WizardCoder, fine-tuned from OpenAI’s CodeLlama). In each iteration, the model generates 10 candidates, which are then evaluated against the visible test suite. Among these, the candidate passing the largest number of tests is retained while others are discarded, ensuring that only the strongest repair is carried forward to the next iteration. This iterative process continues until either a candidate passes all visible tests or the maximum iteration budget is exhausted.

We experimented with Qwen2.5-Coder-32B as our main engine, and compared it against Llama-3 (33B Instruct) and WizardCoder, representing a state-of-the-art LLM baselines.

Experiments were run on Ubuntu 22.04, Intel i9 CPU, 64 GB RAM, and dual Nvidia P40 GPUs.

5.2 Benchmarks

We constructed a new benchmark of 1,000 real-world TLE submissions drawn from Codeforces. We selected 10 problems (difficulty 800–1000) with the highest number of submissions and no image-based descriptions. For each problem:

- (1) We randomly sampled 50 submissions that were eventually repaired by the user (with a ground truth).

Table 3. Distribution of TLE root causes in our dataset.

Error Type	Count
Inefficient algorithm	473
Inefficient Handling	218
Improper I/O	164
Runtime errors (e.g., Endless Loop, Memory Access)	84
Completely wrong / irrelevant code	13
Other (e.g., input format stalls)	132

(2) We randomly sampled 50 submissions that users eventually abandoned (challenging cases).

Notably, many of these submissions exhibit not only efficiency bottlenecks but also additional semantic issues (e.g., hidden wrong-answer or runtime errors that appear once performance is improved). This makes the benchmark more realistic and challenging, and it allows us to demonstrate that *Nettle* can repair both *pure* TLE cases and *mixed* TLE+functional errors.

This yields a balanced dataset of 500 *eventually repaired* and 500 *never repaired* programs. All benchmarks, code, and Codeforces submission history will be made available during the artifact evaluation phase, to be unblinded upon paper acceptance. To show the dataset statistics, in Table 3, we manually categorized each submission into the root-cause categories described in Section 3.

5.3 RQ1: Comparison with Baselines

As a baseline, we simulate the repair process that a typical user would perform by directly prompting a large language model (LLM) to fix the code, without using any additional repair tools. This baseline isolates the model’s intrinsic ability to generate correct repairs.

Evaluation Metrics. We adopt an evaluation protocol that closely emulates the Codeforces judging process. The test suite for each problem is partitioned into two subsets: an initial 10% used during repair and a held-out 90% used for final validation. During the repair process, candidates are only executed against the 10% subset. If a candidate fails on any of these tests, it is immediately discarded. Once the repair loop produces a candidate that passes all visible tests, this candidate is then evaluated on the remaining 90%.

We consider a repair successful if the final candidate passes all held-out cases with the correct expected outputs and without TLE errors. This staged setup allows early pruning of invalid repairs while still providing rigorous validation against unseen inputs. In our experiments, we verified that 100% of the fixes flagged as correct by this protocol were indeed correct.

Results. Across the benchmark, *Nettle* consistently surpasses all baselines, regardless of the underlying LLM (Table 4). With Qwen2.5-Coder, *Nettle* attains a 98.5% repair success rate with a small average edit distance, demonstrating both accuracy and effectiveness. To ensure fairness, all systems followed the same repair budget: ten candidates were generated per iteration, the best candidate was retained at each step, and the process was repeated for up to five iterations.

5.4 RQ2: Role of Contextual Information

We next investigate how providing contextual information affects repair outcomes. We consider two forms of hints:

- **General hint** (e.g., “This code likely times out due to inefficient input handling”).
- **Problem-specific hint** (e.g., “For Problem 102B, convert inputs to strings to avoid repeated parsing overheads”).

Table 4. Repair success rates across system variants (10 problems $\times$ 100 submissions each).

System	Repair Rate	Average Edit Distance
baseline (wizardcoder)	44.0%	59.1
baseline (Llama-3)	73.0%	61.2
baseline (Qwen2.5-Coder)	88.3%	58.5
Nettle (WizardCoder)	95.7%	44.8
Nettle (Llama-3)	94.5%	48.9
Nettle (Qwen2.5-Coder)	98.5%	52.6

Table 5. Impact of hints on repair rates by hint type and model .

Problem	No Hints			General hint			Problem-Specific Hint		
	Llama-3	Wizardcoder	Qwen2.5-Coder	Llama-3	Wizardcoder	Qwen2.5-Coder	Llama-3	Wizardcoder	Qwen2.5-Coder
339B	44%	8%	80%	83%	99%	100%	84%	99%	100%
379A	76%	28%	75%	97%	94%	92%	98%	94%	96%
486A	7%	6%	90%	76%	94%	98%	87%	95%	100%
490A	79%	23%	83%	97%	97%	98%	97%	97%	99%
584A	78%	42%	92%	96%	100%	100%	97%	100%	100%
617A	93%	74%	98%	100%	100%	100%	100%	100%	100%
791A	94%	78%	99%	99%	99%	100%	100%	100%	100%
977A	97%	85%	95%	97%	99%	99%	98%	99%	99%
1A	99%	76%	100%	100%	100%	100%	100%	100%	100%
102B	63%	20%	71%	100%	75%	98%	100%	96%	97%

Table 5 shows that hints significantly improve repair rates, especially for harder problems (e.g., 102B¹²). Lightweight contextual cues substantially boost repair success, in some cases more than doubling accuracy on difficult problems (e.g., 102B). Problem-specific hints yield the largest improvements but need to be designed only once per problem, making them a low-overhead addition. We also observe that such hints benefit general-purpose models like Llama more than code-specialized models like WizardCoder, highlighting that contextual signals can compensate for weaker domain specialization.

Finally, we isolate the effect of the number of candidates per iteration (fan-out, C) and the total number of repair iterations (D). Table 6 shows that increasing either factor alone yields modest gains, but combining both yields the largest improvements. Breadth (candidate diversity) and depth (iterative refinement) are complementary; the best performance emerges when both are maximized.

Why Existing Semantics Repair Technique Cannot Fix TLE. Time Limit Exceeded (TLE) errors fundamentally differ from the bugs that semantics-based repair systems such as Refactory[17] and Clara[15] are designed to handle. These tools target short, single-function programs and rely on local control-flow mutations while ignoring input/output, but TLE errors almost always arise from *global inefficiency* in I/O handling or algorithmic design. In our dataset, 20–30% of TLE cases come directly from input bottlenecks—for example, in Codeforces 102B¹³, `n = int(input())` times out on very large numbers, and the correct fix requires string-based parsing, which these tools cannot even access because they do not repair I/O code. More critically, TLE fixes often require algorithmic redesign that produces code structurally unrelated to the buggy version: in Codeforces 490A¹⁴, the buggy greedy-loop solution and the correct array-based grouping solution share almost

¹²<https://codeforces.com/problemset/problem/102/B>

¹³<https://codeforces.com/problemset/problem/102/B>

¹⁴<https://codeforces.com/problemset/problem/490/A>

Table 6. Effect of candidate pool size (C) and repair depth (D) on repair rates across different code repair models. C denotes the number of candidate fixes generated per round (i.e., pool size), while D denotes the number of iterative repair rounds (i.e., repair depth). The analysis focuses on challenging repair instances to highlight the benefit of increasing C and D . Bold values indicate superior performance.

Configuration	Wizardcoder	LLama-3	Qwen2.5-Coder
$C=1, D=1$	44.0%	73.0%	88.3%
$C=1, D=5$	58.0%	87.9%	89.0%
$C=5, D=1$	76.0%	78.4%	90.5%
$C=5, D=5$	95.7%	94.5%	98.5%

no control-flow similarity, making transplantation impossible. Finally, because Refactory judges only output equivalence, it may wrongly accept programs that succeed on small cases but still gets TLE on large inputs, and slow TLE executions shrink the search budget from thousands of iterations to just one. In short, TLE repair requires reasoning about *time complexity* and *input performance*, which semantics-based repair is incapable of addressing.

5.5 RQ3: Effectiveness of Nettle-Eval

To assess the reliability of our evaluation framework, we compared the outcomes of Nettle-Eval (Qwen2.5-Coder-32B) with the official Codeforces verdicts on a sample of 1000 submissions. Remarkably, the two verdicts matched in all but 15 cases. Even more encouragingly, these mismatches consistently reflect that Nettle-Eval is *stricter* than Codeforces: the 10 programs flagged by our framework as failing were all accepted by Codeforces, yet we found no cases where Codeforces rejected a program that our framework accepted. Closer inspection reveals the following breakdown of the 10 mismatches. Of these, one was judged `Wrong Answer` (WA) by our local evaluator, and nine were judged TLE. All 10 of these submissions were nevertheless labeled `Accepted` (AC) by Codeforces. This asymmetry suggests that our framework exposes corner cases overlooked by the platform’s public test data. In particular, the extra TLE verdicts can be attributed to natural variance in runtime measurement across environments, while the additional WA cases imply that our test inputs are stronger or more diverse than those used in Codeforces, uncovering errors that would otherwise go unnoticed. Notably, the WA case was successfully reported to Codeforces, further validating the accuracy and value of Nettle-Eval. A concrete example illustrates this case. Consider the following repaired candidate for a digit-summation task ¹⁵:

```
def sum_of_digits(s):
    return sum(int(digit) for digit in s)

def count_operations_to_single_digit(n_str):
    count = 0
    while len(n_str) > 1:
        n_str = str(sum_of_digits(n_str))
        count += 1
    return count

# Read the input as a string
n_str = input().strip()
```

¹⁵<https://codeforces.com/problemset/problem/102/B>

```

if 0 <= int(n_str) <= 10**100000:
    result = count_operations_to_single_digit(n_str)
    print(result)
else:
    print("Input out of range")

```

While this code appears correct, the use of `int(n_str)` to check numeric bounds introduces a hidden scalability bottleneck. The call converts the entire string into an integer—an operation that becomes prohibitively expensive or even infeasible when the input length approaches the declared upper bound (up to 10^{100000} digits). On Codeforces, where inputs are typically smaller, this issue remains undetected and the program is marked as AC. Under *Nettle-Eval*, however, the larger and more realistic input distribution triggers a stall or timeout, leading to a TLE. This example highlights how subtle implementation choices can interact with runtime conditions to produce divergent verdicts, and how *Nettle-Eval* uses its broader input coverage to successfully expose such latent inefficiency bugs. Overall, these results demonstrate that *Nettle-Eval* is highly effective: it matches Codeforces in more than 97% of cases, and in the few disagreements, our framework errs on the side of rigor. Far from being a weakness, this stricter stance highlights an important advantage: *Nettle-Eval* can surface hidden flaws and performance issues that would otherwise be misclassified as correct. This property makes *Nettle-Eval* particularly well-suited for educational settings, where learners benefit more from precise and conservative feedback than from overly permissive acceptance.

6 Discussion

We conclude by discussing qualitative aspects of repair quality that quantitative metrics alone cannot capture.

6.1 Repairs vs. Full Rewrites

A key nuance in evaluating automated repair is distinguishing *incremental fixes* from *full rewrites*. Consider the following case in Figure 3.

<pre> n=int(input()) c=0 while n>9: a=n s=0 while a!=0: m=a%10 s+=m a=a//10 n=s c+=1 print(c) </pre>	<pre> n=input() c=0 while len(n)!=1: n=[int(i) for i in n] n=str(sum(n)) c+=1 print(c) </pre>
---	--

Fig. 3. The user’s “fix” is a complete rewrite: the original solution uses integer division and modulus with nested loops, while the new code abandons this strategy and re-implements the task using string processing.

<pre> # Original buggy code (TLE) mna=[int(i) for i in input().split(' ')] m=mna[0] n=mna[1] a=mna[2] auxm=0 auxn=0 kn=0 km=0 while auxm<m: auxm=auxm+a km=km+1 while auxn<n: auxn=auxn+a kn=kn+1 out=km*kn print(out) </pre>	<pre> # Tool-generated fix (O(1)) mna=[int(i) for i in input().split(' ')] m=mna[0] n=mna[1] a=mna[2] km=m//a kn=n//a if m%a!=0: km=km+1 if n%a!=0: kn=kn+1 out=km*kn print(out) </pre>
---	---

Fig. 4. Our tool transforms two inefficient linear loops ($O(m/a + n/a)$) into a direct ceiling formula using integer division and conditional increment ($O(1)$).

The original program¹⁶ attempted digit extraction with integer division and modulus, but the nested loops led to inefficiency and ultimately a TLE verdict. The user's fix¹⁷, however, did not refine this approach; it discarded it entirely by switching to string conversion and list comprehensions.

Such rewrites are fundamentally distinct from true repairs. Our system is designed to preserve the original problem-solving intent and propose actionable corrections (e.g., reducing redundant computation, adjusting loop bounds). By contrast, the user's new submission represents a complete replacement. It should not be interpreted as a limitation of *Nettle*, but rather as an instance of programmers abandoning one strategy and starting over with another, a common behavior in online programming.

This distinction underscores the pedagogical value of automated repair: incremental edits scaffold learning and help programmers refine existing solutions, whereas wholesale rewrites obscure intent and provide little insight into the repairability of the original code.

6.2 High-Quality Repair Example (Problem 1A, Submission 10700220)

The example¹⁸ in Figure 4 highlights the quality of our system's repairs. The buggy submission relied on linear accumulation loops, leading to $O(m/a + n/a)$ complexity and repeated TLE. Our tool replaced this with integer division plus conditional rounding up, the mathematically standard ceiling formula. This change guarantees correctness, since it matches the official editorial solution and handles both divisible and non-divisible boundaries. It improves complexity by reducing the algorithm to constant time, fully avoiding timeout risks. It enhances readability, as the repaired code directly reflects the intended "rows $\times$ columns" geometry. Finally, it improves robustness by eliminating reliance on intermediate accumulations, thereby reducing the chance of implementation

¹⁶<https://codeforces.com/contest/102/submission/40030614>

¹⁷<https://codeforces.com/contest/102/submission/40032105>

¹⁸<https://codeforces.com/contest/1/submission/10700220>

```
n, m = list(map(int,
    input().split()))
task = list(map(int,
    input().split()))
count = 0
cur = 1 # Start from
    house 1
for i in task:
    while cur != i:
        if cur == n: #
            wrap to 1
            cur = 1
        else:
            cur += 1
        count += 1
print(count)
```

(a) Attempt 1: Original buggy code (TLE, $O(nm)$)

```
n, m = list(map(int,
    input().split()))
task = list(map(int,
    input().split()))
count = 0
cur = 1
for i in task:
    dist_clockwise = (i
        - cur) % n
    dist_c_clockwise = (
        cur - i) % n
    # Error: may move
        counterclockwise
    if dist_clockwise <=
        dist_c_clockwise
        :
        count +=
            dist_clockwise

    else:
        count +=
            dist_c_clockwise
```

```
cur = i
print(count)
```

(b) Attempt 2: WA (counterclockwise shortcut)

```
n, m = list(map(int,
    input().split()))
task = list(map(int,
    input().split()))
count = 0
cur = 1 # initial
    position is house 1
for i in task:
    if cur <= i:
        dist = i - cur
        # forward
    else:
        dist = n - cur +
            i # wrap
            around
        count += dist
        cur = i
print(count)
```

(c) Attempt 3: AC (clockwise-only distance)

Fig. 5. Repair pipeline for Problem 339B: (a) TLE due to simulation, (b) WA from invalid counterclockwise moves, (c) AC by encoding the correct clockwise distance.

errors or overflow. Together, these properties demonstrate that our system generates fixes that are not only valid, but also optimal, elegant, and pedagogically aligned with canonical human solutions.

6.3 Iterative Repair Example (Problem 339B, Submission 35093821)

The example¹⁹ in Figure 5 illustrates how *Nettle* enables iterative refinement of repairs. Starting from a TLE submission, we produced an intermediate repair that improved efficiency but violated the semantics (WA), before converging on a correct solution (AC). This trajectory highlights the value of repair pipelines: debugging is rarely a one-shot process, and effective evaluation supports progressive improvement.

The progression from Attempt 1 → Attempt 2 → Attempt 3 shows how *Nettle-Eval* not only distinguishes genuine repairs from overfitting rewrites, but also supports an *iterative design*. Rather than rejecting intermediate fixes outright, the framework helps programmers understand why a solution fails (efficiency vs. semantics) and guides them toward principled, optimal solutions.

¹⁹<https://codeforces.com/contest/339/submission/35093821>

7 Related Work

Time-Limit-Exceeded (TLE) Errors. The software engineering and programming languages communities have long sought to build tools that assist learners in diagnosing and correcting errors in online programming environments. Prior work spans a wide range of tasks, including error localization [7, 11, 53], automated repair [43, 47, 55], and intelligent feedback generation [15, 17, 23, 26, 43, 44, 46]. However, the vast majority of these efforts target programs with observable *wrong-answer* outcomes, where the output diverges from a reference solution under the same inputs. In contrast, *time-limit-exceeded (TLE)* errors [16] represent a fundamentally different failure mode: the program is computationally inefficient, and thus far more challenging for learners to diagnose or repair. While a few studies have explored performance-related bug repair [30, 31], they focus on narrowly defined patterns such as too many loops or repeated computations. These approaches cannot address the broader algorithmic inefficiencies and improper I/O handling that dominate real-world TLE cases.

Automated Feedback Generation. Over the past decade, the problem of automatically generating feedback for incorrect programming assignments has attracted substantial attention from the software engineering and computing education communities [6, 23, 32, 38, 44, 47, 49, 51, 57]. Early systems [18, 43] required manual intervention from instructors to guide feedback creation. More recent approaches [15, 17, 26, 46] achieve automation by leveraging large repositories of student submissions to generate feedback. Despite these advances, existing tools are ill-suited for handling programs that fail due to TLE errors. Some techniques [26, 50] operate only on syntactic faults, whereas programs having TLE errors are syntactically correct. Others, including SemCluster and Refactory [17, 33], focus on semantic errors and rely heavily on dynamic analysis. Dynamic instrumentation becomes ineffective when every test input results in a timeout, leaving no successful execution to observe. More fundamentally, most feedback generators equate correctness with input-output equivalence and ignore runtime efficiency. Consequently, they cannot diagnose or repair programs that are functionally correct but computationally inefficient, the top class of problems targeted in this work.

LLM-based Automated Program Repair. Automated program repair (APR) has been extensively studied over the past decade. Early research primarily explored *heuristic-based* approaches [12, 13, 21, 22, 36] and *semantics-based* techniques [1, 28, 29], which rely on mutation heuristics or symbolic reasoning to synthesize patches. More recently, *learning-based* methods [8, 24, 25, 35, 40, 50] have become prominent by learning repair patterns from historical patches, yielding higher fix quality and generalization. With the advent of large language models, *LLM-driven* repair systems [3, 9, 19, 37, 39, 41, 42, 52, 56] have rapidly emerged, demonstrating strong capabilities in generating human-like patches across diverse languages and domains. Despite these advances, existing APR techniques are suited for isolated bugs in large codebases. In contrast, TLE errors on online programming platforms often require algorithm-level redesign rather than localized edits. This distinction makes TLE repair a substantially different and underexplored challenge for LLM-based systems.

Online Programming Assignments. Recent research on online programming platforms has focused on enhancing the overall learner experience through multiple directions. First, natural-language-based feedback systems generate personalized explanations and hints for student code [34]. Second, automated test-case generation frameworks seek to construct comprehensive test suites, thereby enhancing the validity and reliability of automated grading [44]. Third, several studies explore how to design more effective and engaging programming assignments [27]. Finally, recent empirical work investigates how automated feedback affects learning outcomes and user behavior [20]. Our work is complementary to these efforts: it demonstrates that the challenging

problem of repairing time-limit-exceeded (TLE) submissions on online programming platforms can be effectively addressed through `Nettle`.

8 Conclusion

This work provides a systematic understanding of Time Limit Exceeded (TLE) errors in online programming platforms and demonstrates the feasibility of automated support for their diagnosis and repair. Through a large-scale empirical study of 1,000 Codeforces submissions, we identify key root causes, including not only algorithmic inefficiency but also semantic errors such as infinite loops, poor data structure choices, and inefficient I/O. These findings challenge prevailing assumptions about TLE and uncover concrete opportunities for performance-aware feedback.

Building on this analysis, we introduce `Nettle` and `Nettle-Eval`, which together enable automated repair of TLE errors at scale. Our evaluation shows that targeted, LLM-driven feedback can outperform generic repair models in both accuracy and repair quality. These results point to a promising direction for future tools that reduce trial-and-error, improve the programming experience, and foster deeper algorithmic insight—especially for learners navigating complex problem-solving tasks.

References

- [1] Umair Z. Ahmed, Zhiyu Fan, Jooyong Yi, Omar I. Al-Bataineh, and Abhik Roychoudhury. 2022. Verifix: Verified Repair of Programming Assignments. *ACM Trans. Softw. Eng. Methodol.* 31, 4, Article 74 (jul 2022), 31 pages. <https://doi.org/10.1145/3510418>
- [2] Sagnik Anupam, Alexander Shypula, and Osbert Bastani. 2025. LLM Program Optimization via Retrieval Augmented Search. arXiv:2501.18916 [cs.LG] <https://arxiv.org/abs/2501.18916>
- [3] Rohan Bavishi, Harshit Joshi, José Cambronero, Anna Fariha, Sumit Gulwani, Vu Le, Ivan Radiček, and Ashish Tiwari. 2022. Neurosymbolic Repair for Low-Code Formula Languages. 6, OOPSLA2, Article 164 (oct 2022), 30 pages. <https://doi.org/10.1145/3563327>
- [4] ICPC UCLA ACM Chapter. 2023. *State of Competitive Programming: ICPC Participation Trends*. <https://icpc.uclaacm.com/icpc> Accessed: 2023-11-18.
- [5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgén Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. <https://doi.org/10.48550/ARXIV.2107.03374>
- [6] Loris D’Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qclose: Program Repair with Quantitative Objectives. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9780)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 383–401. https://doi.org/10.1007/978-3-319-41540-6_21
- [7] Vidroha Debroy and W. Eric Wong. 2010. Using Mutation to Automatically Suggest Fixes for Faulty Programs. In *2010 Third International Conference on Software Testing, Verification and Validation*. 65–74. <https://doi.org/10.1109/ICST.2010.66>
- [8] 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 *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. <https://openreview.net/forum?id=SJeqs6EFvB>
- [9] Elizabeth Dinella, Todd Mytkowicz, Alexey Svyatkovskiy, Christian Bird, Mayur Naik, and Shuvendu K. Lahiri. 2021. DeepMerge: Learning to Merge Programs. arXiv:2105.07569 [cs.SE] <https://arxiv.org/abs/2105.07569>
- [10] 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*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 1536–1547. <https://doi.org/10.18653/v1/2020.findings-emnlp.139>

- [11] Elena L. Glassman, Aaron Lin, Carrie J. Cai, and Robert C. Miller. 2016. Learnersourcing Personalized Hints. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing, CSCW 2016, San Francisco, CA, USA, February 27 - March 2, 2016*, Darren Gergle, Meredith Ringel Morris, Pernille Bjørn, and Joseph A. Konstan (Eds.). ACM, 1624–1634. <https://doi.org/10.1145/2818048.2820011>
- [12] Claire Goues, Stephanie Forrest, and Westley Weimer. 2013. Current Challenges in Automatic Software Repair. *Software Quality Journal* 21, 3 (sep 2013), 421–443. <https://doi.org/10.1007/s11219-013-9208-0>
- [13] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62, 12 (nov 2019), 56–65. <https://doi.org/10.1145/3318162>
- [14] National Research Group. 2023. Research Report on Computer Competitions in Chinese Universities (2012-2022). *Springer Research* 5 (2023), 1–15. <https://link.springer.com/content/pdf/10.1007/s44217-023-00034-1.pdf> Accessed: 2023-11-18.
- [15] Sumit Gulwani, Ivan Radicek, and Florian Zuleger. 2018. Automated Clustering and Program Repair for Introductory Programming Assignments. *SIGPLAN Not.* 53, 4 (June 2018), 465–480. <https://doi.org/10.1145/3296979.3192387>
- [16] S. Halim and F. Halim. 2013. *Competitive Programming 3: The New Lower Bound of Programming Contests*. Number v. 3. Lulu.com. <https://books.google.com/books?id=vUc-nwEACAAJ>
- [17] Y. Hu, U. Z. Ahmed, S. Mechtaev, B. Leong, and A. Roychoudhury. 2019. Re-Factoring Based Program Repair Applied to Programming Assignments. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 388–398.
- [18] Shalini Kaleeswaran, Anirudh Santhiar, Aditya Kanade, and Sumit Gulwani. 2016. Semi-Supervised Verified Feedback Generation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (Seattle, WA, USA) (FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 739–750. <https://doi.org/10.1145/2950290.2950363>
- [19] Nima Karimipour, Michael Pradel, Martin Kellogg, and Manu Sridharan. 2025. LLM-Based Repair of Static Nullability Errors. arXiv:2507.20674 [cs.SE] <https://arxiv.org/abs/2507.20674>
- [20] Natalie Kiesler. 2022. An Exploratory Analysis of Feedback Types Used in Online Coding Exercises. arXiv:2206.03077 [cs.HC] <https://arxiv.org/abs/2206.03077>
- [21] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-Written Patches. In *Proceedings of the 2013 International Conference on Software Engineering (San Francisco, CA, USA) (ICSE '13)*. IEEE Press, 802–811.
- [22] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72. <https://doi.org/10.1109/TSE.2011.104>
- [23] Leping Li, Hui Liu, Kejun Li, Yanjie Jiang, and Rui Sun. 2022. Generating Concise Patches for Newly Released Programming Assignments. *IEEE Transactions on Software Engineering* (2022), 1–1. <https://doi.org/10.1109/TSE.2022.3153522>
- [24] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic Inference of Code Transforms for Patch Generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (Paderborn, Germany) (ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 727–739. <https://doi.org/10.1145/3106237.3106253>
- [25] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (St. Petersburg, FL, USA) (POPL '16)*. Association for Computing Machinery, New York, NY, USA, 298–312. <https://doi.org/10.1145/2837614.2837617>
- [26] Yunlong Lu, Na Meng, and Wenxin Li. 2021. FAPR: Fast and Accurate Program Repair for Introductory Programming Courses. *CoRR* abs/2107.06550 (2021). arXiv:2107.06550 <https://arxiv.org/abs/2107.06550>
- [27] Bradley McDanel and Ed Novak. 2025. Designing LLM-Resistant Programming Assignments: Insights and Strategies for CS Educators. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1 (Pittsburgh, PA, USA) (SIGCSE 2025)*. Association for Computing Machinery, New York, NY, USA, 756–762. <https://doi.org/10.1145/3641554.3701872>
- [28] Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2018. Semantic program repair using a reference implementation. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 129–139. <https://doi.org/10.1145/3180155.3180247>
- [29] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 691–701. <https://doi.org/10.1145/2884781.2884807>
- [30] Khanh Nguyen and Guoqing Xu. 2013. Cachetor: Detecting Cacheable Data to Remove Bloat. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (Saint Petersburg, Russia) (ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 268–278. <https://doi.org/10.1145/2491411.2491416>

- [31] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. Caramel: Detecting and Fixing Performance Problems That Have Non-Intrusive Fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) (*ICSE '15*). IEEE Press, 902–912.
- [32] Benjamin Paassen, Barbara Hammer, Thomas W. Price, Tiffany Barnes, Sebastian Gross, and Niels Pinkwart. 2018. The Continuous Hint Factory - Providing Hints in Vast and Sparsely Populated Edit Distance Spaces. *Journal of Educational Data Mining* 10, 1 (Jun. 2018), 1–35. <https://doi.org/10.5281/zenodo.3554697>
- [33] David M. Perry, Dohyeon Kim, Roopsha Samanta, and Xiangyu Zhang. 2019. SemCluster: Clustering of Imperative Programming Assignments Based on Quantitative Semantic Features. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) (*PLDI 2019*). Association for Computing Machinery, New York, NY, USA, 860–873. <https://doi.org/10.1145/3314221.3314629>
- [34] Tung Phung, José Cambronero, Sumit Gulwani, Tobias Kohn, Rupak Majumdar, Adish Singla, and Gustavo Soares. 2023. Generating High-Precision Feedback for Programming Syntax Errors using Large Language Models. arXiv:2302.04662 [cs.PL] <https://arxiv.org/abs/2302.04662>
- [35] Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. 2016. Sk_p: A Neural Program Corrector for MOOCs. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity* (Amsterdam, Netherlands) (*SPLASH Companion 2016*). Association for Computing Machinery, New York, NY, USA, 39–40. <https://doi.org/10.1145/2984043.2989222>
- [36] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziyang Dai, and Chengsong Wang. 2014. The Strength of Random Search on Automated Program Repair. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) (*ICSE 2014*). Association for Computing Machinery, New York, NY, USA, 254–265. <https://doi.org/10.1145/2568225.2568254>
- [37] Jianxing Qin, Alexander Du, Danfeng Zhang, Matthew Lentz, and Danyang Zhuo. 2025. Can Large Language Models Verify System Software? A Case Study Using FSCQ as a Benchmark. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems* (Banff, AB, Canada) (*HotOS '25*). Association for Computing Machinery, New York, NY, USA, 34–41. <https://doi.org/10.1145/3713082.3730382>
- [38] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 404–415.
- [39] Pat Rondon, Renyao Wei, José Cambronero, Jürgen Cito, Aaron Sun, Siddhant Sanyam, Michele Tufano, and Satish Chandra. 2025. Evaluating Agent-based Program Repair at Google. arXiv:2501.07531 [cs.SE] <https://arxiv.org/abs/2501.07531>
- [40] Mark Santolucito, Jialu Zhang, Ennan Zhai, Jürgen Cito, and Ruzica Piskac. 2022. Learning CI Configuration Correctness for Early Build Feedback. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 1006–1017. <https://doi.org/10.1109/SANER53432.2022.00118>
- [41] Yuchen Shao, Yuheng Huang, Jiawei Shen, Lei Ma, Ting Su, and Chengcheng Wan. 2025. Are LLMs Correctly Integrated into Software Systems? arXiv:2407.05138 [cs.SE] <https://arxiv.org/abs/2407.05138>
- [42] Yuan Si, Daming Li, Hanyuan Shi, and Jialu Zhang. 2025. ViScratch: Using Large Language Models and Gameplay Videos for Automated Feedback in Scratch. arXiv:2509.11065 [cs.SE] <https://arxiv.org/abs/2509.11065>
- [43] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated Feedback Generation for Introductory Programming Assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) (*PLDI 13*). Association for Computing Machinery, New York, NY, USA, 15–26. <https://doi.org/10.1145/2491956.2462195>
- [44] Dowon Song, Woosuk Lee, and Hakjoo Oh. 2021. *Context-Aware and Data-Driven Feedback Generation for Programming Assignments*. Association for Computing Machinery, New York, NY, USA, 328–340. <https://doi.org/10.1145/3468264.3468598>
- [45] Chengpeng Wang, Peisen Yao, Wensheng Tang, Qingkai Shi, and Charles Zhang. 2022. Complexity-guided container replacement synthesis. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 68 (April 2022), 31 pages. <https://doi.org/10.1145/3527312>
- [46] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Search, Align, and Repair: Data-Driven Feedback Generation for Introductory Programming Exercises (*PLDI 2018*). Association for Computing Machinery, New York, NY, USA, 481–495. <https://doi.org/10.1145/3192366.3192384>
- [47] Ke Wang, Zhendong Su, and Rishabh Singh. 2018. Dynamic Neural Program Embeddings for Program Repair. In *International Conference on Learning Representations*. <https://openreview.net/forum?id=BJuWrGW0Z>
- [48] Guoqing Xu. 2013. Resurrector: a tunable object lifetime profiling technique for optimizing real-world programs. *SIGPLAN Not.* 48, 10 (Oct. 2013), 111–130. <https://doi.org/10.1145/2544173.2509512>
- [49] Feng Yao, Zilong Wang, Liyuan Liu, Junxia Cui, Li Zhong, Xiaohan Fu, Haohui Mai, Vish Krishnan, Jianfeng Gao, and Jingbo Shang. 2025. Training Language Models to Generate Quality Code with Program Analysis Feedback.

- arXiv:2505.22704 [cs.CL] <https://arxiv.org/abs/2505.22704>
- [50] Michihiro Yasunaga and Percy Liang. 2021. Break-It-Fix-It: Unsupervised Learning for Program Repair. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 11941–11952. <http://proceedings.mlr.press/v139/yasunaga21a.html>
 - [51] Jooyong Yi, Umair Z. Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. 2017. A Feasibility Study of Using Automated Program Repair for Introductory Programming Assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (Paderborn, Germany) (ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 740–751. <https://doi.org/10.1145/3106237.3106262>
 - [52] Yiming Zeng, Wanhao Yu, Zexin Li, Tao Ren, Yu Ma, Jinghan Cao, Xiyan Chen, and Tingting Yu. 2025. Bridging the Editing Gap in LLMs: FineEdit for Precise and Targeted Text Modifications. arXiv:2502.13358 [cs.CL] <https://arxiv.org/abs/2502.13358>
 - [53] Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2017. SHErrLoc: A Static Holistic Error Locator. *ACM Trans. Program. Lang. Syst.* 39, 4, Article 18 (Aug. 2017), 47 pages. <https://doi.org/10.1145/3121137>
 - [54] Jialu Zhang, José Pablo Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. 2024. PyDex: Repairing Bugs in Introductory Python Assignments using LLMs. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 133 (April 2024), 25 pages. <https://doi.org/10.1145/3649850>
 - [55] Jialu Zhang, De Li, John Charles Kolesar, Hanyuan Shi, and Ruzica Piskac. 2023. Automated Feedback Generation for Competition-Level Code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (Rochester, MI, USA) (ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 13, 13 pages. <https://doi.org/10.1145/3551349.3560425>
 - [56] Jialu Zhang, Todd Mytkowicz, Mike Kaufman, Ruzica Piskac, and Shuvendu K. Lahiri. 2022. Using Pre-Trained Language Models to Resolve Textual and Semantic Merge Conflicts (Experience Paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (Virtual, South Korea) (ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 77–88. <https://doi.org/10.1145/3533767.3534396>
 - [57] Kurtis Zimmerman and Chandan R. Rupakheti. 2015. An Automated Framework for Recommending Program Elements to Novices. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (Lincoln, Nebraska) (ASE '15)*. IEEE Press, 283–288. <https://doi.org/10.1109/ASE.2015.54>