GI-Agent Search-Based LLM Agent for Code Optimization with Genetic Improvement

Donghyun Lee, William B. Langdon and Justyna Petke
lukeleeai@gmail.com,w.langdon@cs.ucl.ac.uk,j.petke@ucl.ac.uk
Computer Science, University College London, Gower Street, WC1E 6BT, London
United Kingdom

Abstract

GI-Agent integrates Large Language Models (LLMs) into Genetic Improvement (GI) to autonomously optimise computer program source code. GI-Agent uses an LLM to allow multi-generational evolutionary learning. Guided by a memory of past actions, known in AI as reflections, it exploits them to give insight and rationale behind software edits, giving better context-aware, less stochastic, mutations and crossovers. Reflections enable GI-Agent to reason about both earlier compile time and runtime successes and failures, and so refine its strategy over time. Integrated into the Magpie GI framework and evaluated on the SAT4J (Java) and MiniSAT (C++) benchmarks, GI-Agent consistently generates more viable and better variants. By combining few-shot prompting with structured search, GI-Agent demonstrates how LLMs can enhance automated program optimisation.

CCS Concepts

• **Software and its engineering → Search-based software engineering**.

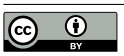
Keywords

Artificial Intelligence (AI), Chain-of-Thought, LangChain, OpenAI GPT-o3 Mini, SBSE, Evolutionary Computing, Genetic Programming, Lamarck multigenerational learning, Linux perf.

ACM Reference Format:
Donghyun Lee, William B. Langdon and Justyna Petke. 2026. GI-Agent Search-Based LLM Agent for Code Optimization with Genetic Improvement. In *15th International Workshop on Genetic Improvement (GI '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3786162.3793232

1 Introduction

We investigate a compelling proposition: the potential for Large Language Models (LLMs) to serve as *agents for search-based code optimization* that learn from successive genetic improvement (GI) generations. We propose a novel agentic framework, *GI-Agent* for iteratively evolving program source code for optimization. GI-agent generates multiple code variants, evaluates their performance, reflects on the factors contributing to success or failure, and applies these insights to refine subsequent GI mutations and crossovers.



GI '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2394-0/2026/04
https://doi.org/10.1145/3786162.3793232

2 Literature Review

2.1 Genetic Improvement (GI) Applications in Software Engineering

Bruce et al. [5] showed that genetic improvement (GI) can be successfully applied to reduce energy consumption in software systems, they used GI techniques to optimize runtime efficiency. Petke et al. [22] employed GI to enhance the performance of the MiniSAT solver (Section 5), demonstrating its utility in refining existing software tools. Beyond CPU-based systems, Langdon et al. [13, 14] extended GI to GPU software, improving CUDA-based DNA analysis and medical imaging tools and showing GI's scalability to parallel computing environments. Recent work by Wang et al. [9] and Bouras et al. [3] has demonstrated the potential of using Large Language Models (LLMs) to guide genetic operations, specifically, mutation and crossover, showcasing their promise in finding effective speedup patches.

2.2 Large Language Models (LLMs) and Search-Based Software Engineering

The fusion of LLMs with SBSE has emerged as a promising avenue for enhancing automation and optimization in software development. For example, Gao et al. [25] propose a hybrid approach that integrates genetic algorithms (GAs) with LLMs for code optimization. Their GA finds optimization patterns, which their SBLLM retrieves (using a Retrieval-Augmented Generation (RAG) LLM [17]) and applies them by using an LLM to mutation and crossover operations. However, their method lacks a fitness feedback loop for the LLM-generated transformations, relying instead on the retrieval process to navigate the search space. Kang and Yoo [11] presented a proof-of-concept where an LLM serves as a mutation operator. While Haraldsson [7] extend this idea to automatic program repair (APR), using LLMs for both mutation and crossover operations. Brownlee et al. [4] incorporate few-shot LLMs as a subroutine for mutation in GI. Also Lemieux et al. [16] propose CODAMOSA, a hybrid SBSE approach that employs LLMs to overcome coverage plateaus in test generation.

2.3 LLM-Based Agents: Chain of Thought (CoT), Agents and Software Engineering

Agent-based methodologies, facilitated through advanced prompt engineering and architectural innovations, enable LLMs to emulate human-like reasoning processes, such as step-by-step analysis, exploration of alternatives and self-improvement [8] and so are a natural fit with Software Engineering (SE).

Xi et al. [32] explain that LLMs increasingly serve as the "brain" for computational agents.

The Chain-of-Thought (CoT) prompting technique, introduced by Wei et al. [28], prompts LLMs to generate intermediate reasoning steps before arriving at a final answer. Zhang et al.'s AI-Assisted Software Development Framework (AISD) [26] leverages Reflexion [24] to iteratively refine code implementations based on feedback, achieving a 75.2% pass rate on benchmark tasks. By incorporating Reflexion's self-reflection mechanism, AISD enables agents to learn from failed attempts and improve over time. Joshi et al. [10] use LLMs to create bug fixes. Similarly, Ma et al's SpecGen [18] employs mutation operators and heuristic selection to generate and optimize program specifications.

3 Genetic Improvement Agent: GI-Agent

We propose *GI-Agent* to address the limitations of traditional GI: namely its lack of memory across generations, absence of context-aware edit strategies, and limited interpretability. GI-Agent enhances standard GI by enabling learning from prior outcomes, generating effective program source code edits and explaining the reasoning behind them. It adds the following innovations to Magpie [2]:

Context-Aware Genetic Operations: Traditional GI applies mutation and crossover operators heuristically, often disregarding the structure or semantics of the target program. GI-Agent instead leverages LLMs to perform edits that are informed by both the code context and past outcomes, producing more targeted and meaningful modifications.

Iterative Learning Across Generations: While traditional GI is guided by fitness scores, it lacks the ability to remember or generalize from prior edits. GI-Agent introduces a reflection mechanism that records feedback from each edit. These reflections are used to inform future mutation and crossover decisions, seeking to enable adaptive optimization over time.

Interpretability and Justification: In traditional GI, patches are generated through random or heuristic means, offering no explanation for why a particular change was made. In contrast, GI-Agent prompts the LLM to reason about the strategy before generating an edit. This built-in reasoning (articulated before the code update) is visible and reviewable, offering insights into the intent behind each modification. As a result, GI-Agent enhances interpretability by revealing the LLM's decision-making process.

3.1 Implementing GI-Agent

We extended Magpie[1] to seamlessly incorporate GI-Agent, starting by adding `algorithm = GeneticProgrammingLLM` to Magpie's scenario file format. The GeneticProgrammingLLM algorithm is implemented (Figure 1) by 1) initialising reflection memory to support learning across generations. 2) sample LLM reflections. 3) create offspring via LLM-guided mutation using sampled reflections. 4) reflections generated by LLM comparing parent with its offspring. 5) create offspring via LLM-guided crossover using reflections. 6) add more reflections by LLM comparing both parent with their offspring. Figure 1 shows the position of the genetic LLM generating offspring and reflection LLM within GI-Agent.

[1]GI-Agent extends Magpie (commit 83e0276) downloaded from https://github.com/bloa/magpie January 2025.

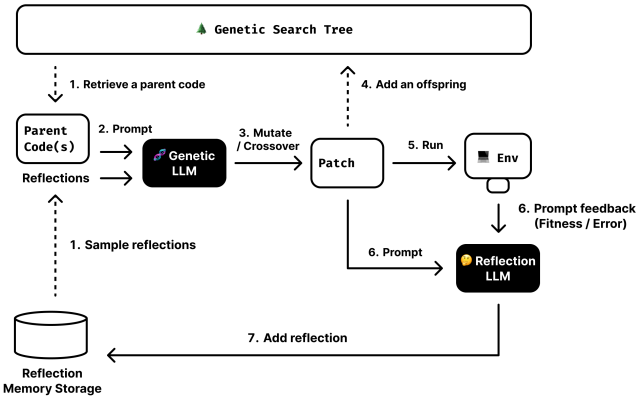


Figure 1: Reflection process within GI-Agent. Two LLMs (1. 6. dark) and a Reflection store (7.) are integrated into genetic improvement's mutation/crossover patch creation (2. 3.) and fitness testing (5.) (white boxes).

This iterative process enables the GI-Agent to learn from past generations, refining its mutation and crossover strategies over time. The use of reflections addresses the limitation of traditional GI's stochastic edit selection by introducing adaptability and context-awareness. Future work, such as incorporating Retrieval-Augmented Generation (RAG) [17] for more contextually relevant reflection retrieval, could further enhance this process by ensuring the most helpful reflections are used.

3.2 Mutation

3.2.1 Diversification. We ask the LLM to generate multiple mutations in one output generation. This approach ensures the LLM is aware of all mutations it produces within a single prompt. To encourage both more intentional and varied edits, we also require the LLM to specify a high-level "strategy" for each mutation (e.g., is it optimizing a specific method, blocks, or import packages),

3.2.2 Reflections. We add reflections from prior generations into the prompt given to the "genetic" LLM. These reflections, summarizing past successes and failures, enable the LLM to learn from previous outcomes and refine its strategies.

3.2.3 Diff Format LLM prompts. As was found with GenProg [29], when dealing with mutations etc., it is better to work with code differences rather than complete programs. Therefore we use a diff-based format, common in tools like Cursor [6], where the LLM outputs lines to remove (prefixed with a "-") and replace (prefixed with a "+").

Each mutation in GI-Agent is represented as a diff node containing only the LLM generated edit. To execute a variant, the corresponding diff is applied dynamically to the original source code, reconstructing the modified program on the fly.

3.3 Crossover

Crossover is a fundamental genetic operation used to combine parts of multiple parent solutions to produce new variants that potentially inherit beneficial traits from each. The intuition is that combining complementary "genes" may yield offspring with higher performance than any single parent.

In traditional GI crossover is random, operating at the line or AST level usually without deeper semantic reasoning. In contrast, *GI-Agent* uses the reasoning capabilities of a LLM to guide crossover. It aims to generate semantically coherent and performance-improving variants by selecting and merging traits based on contextual understanding. Notice the LLM's output should contain text giving its `strategy:`.

3.3.1 Diversification. As with mutation, we ask the LLM to generate multiple crossover outputs in each generation. To enable informed crossover decisions, we provide it with the fitness score of each parent.

3.3.2 Reflections. Similar to mutations, we incorporate reflections for crossover operations. Unlike mutations, however, these reflections focus solely on prior crossover attempts (excluding mutation operations), enabling the LLM to concentrate on successful crossover strategies.

3.3.3 Complete Original Code for Crossover. In contrast to mutations, we include the original target code as an input for crossovers. This serves as an anchor, providing a consistent reference for the LLM to generate a crossover diff. Without this, the LLM might produce diffs relative to either parent code, complicating the subsequent logic and processing.

3.4 Reflection

Here, we elaborate on the mechanism of reflection, a process where the LLM analyzes modifications between parent and child code to assess their impact on fitness. The LLM is prompted with contextual inputs, such as parent code, child code, fitness scores, and run results (see prompt below), to generate concise reflections (limited to 100 words) explaining what strategies succeeded or failed.

```
You are an expert developer. Previously, you were given a code and you wrote a new code to make the code faster by improving the fitness score.

This was the parent code you were given: {parent_codes}

This was the fitness score of the parent code: {parent_fitnesses}

This was the new code you wrote: {child_code}

Here is the run result: {run_result}

Here is the run stderr (if the code failed): {run_stderr}

New code fitness score: {run_fitness} (-1 means the code failed to run. The positive, smaller fitness score means the new code is faster.)

Now, you need to reflect on this.
The purpose is to learn what strategy worked, what didn't, so that you can later use this compressed knowledge to write better codes.
For example, if the new code is faster, you should reflect on what strategies made it faster.
If the new code is slower, you should reflect on what strategies made it slower.
If the test or compile failed, you should reflect on what went wrong by looking at the stderr.

First, reflect on the changes.
And then, rewrite the reflection by starting with <reflection> tag and end with </reflection> tag so that another LLM can learn from the reflection inside those tags without seeing the code.
Now, please reflect on the new code and the run result:
```

We defined three reflection types to guide the Mutation/Crossover LLM in generating improved code variants: ALL, TOP K [1, 23] and NONE. ALL means all four reflection types (Speedup, Increased Latency, Compilation and Test Error) are provided to the Mutation/Crossover LLM, enabling it to learn from both successes and failures. In contrast, TOP K feeds only the speedup reflection. For example, Speedup gave the reflection "Removing the call to sort literals in the clause addition routine significantly reduced overhead during clause insertion, yielding a roughly 20% speed improvement. Other minor refactoring and formatting had minimal impact. This suggests that minimizing unnecessary array operations in hot paths, like eliminating `sort(ps)` in `addClause`, can substantially enhance SAT solver runtime."

4 Research Questions

RQ1: How Does GI-Agent Compare to Traditional Genetic Improvement in Terms of Solution Quality?

RQ2: How Efficiently Does GI-Agent Discover High-Quality Program Variants?

RQ3: How Does GI-Agent Affect the Viability of Generated Program Variants?

RQ4: How Does the Use of Reflection Influence the Diversity of Generated Patches?

5 Experiments

5.1 Benchmarks: MiniSAT and SAT4J

MiniSAT [27] is a compact SAT solver implemented in `C++`. With a relatively small codebase (approximately 1000 lines of code, LOC), it is often used in optimization [21].

SAT4J, in contrast, is a `Java`-based SAT solver that supports a broader range of SAT-related problems, including pseudo-Boolean and MaxSAT. **SAT4J**'s more than 2 500 lines of Java code (2.5k+ LOC) have been carefully optimized for practical use cases and are actively maintained [15].

5.2 LLM: LangChain GPT-o3 Mini Configuration

All mutation, crossover, and reflection operations are performed using the GPT-o3 Mini language model (`o3-mini-2025-01-31`), accessed via the OpenAI API within the LangChain framework [12]. Following Yang et al. [31], we use a temperature of 1.0 to encourage the generation of diverse and creative solutions [19].

5.3 GI-Agent and Magpie Configuration

Following Bouras et al. [3] we chose:

- `pop_size` = 20
- `max_steps` = 220
- `offspring_elitism` = 0.2
- `offspring_mutation` = 0.6
- `offspring_crossover` = 0.2

This enables both GI-Agent and Magpie to explore 220 mutations and crossovers across 10 generations (`max_steps` / `pop_size`). (20 evaluations are used for the initial random populations.) We adopt this setup across all GI-Agent variants: `NoReflect`, `ReflectAll` and `ReflectSuccess`. For `ReflectAll`, we feed the LLM a curated set of 15 operations per generation: the top 5 operations with highest speedup improvement, the bottom 5 with lowest improvement, and 5 randomly sampled operations that failed tests or compilation. For `ReflectSuccess`, we reflect only on the top 15 highest-performing successful operations. This selective strategy prevents overwhelming the LLM with excessive context while preserving informative diversity in reflective prompts.

Experiments were conducted on a dedicated Intel Xeon CPU E5-1620 3.60GHz, 8 cores, running Rocky Linux.

6 Results and Critical Analysis

6.1 RQ1: How Does GI-Agent Compare to Traditional Genetic Improvement in Terms of Solution Quality?

We compare the speed up achieved by GI-Agent variants (`NoReflect`, `ReflectAll` and `ReflectSuccess`) against traditional genetic improvement (Magpie). Each is evaluated over 20 runs using different pseudo-random seeds. The results are summarized in Table 1, which reports the minimum, median, and maximum speedups. Magpie and GI-Agent use independent pseudo random number seeds and so our twenty runs should be independent however there is some evidence that due to caching [20] LLM runs using identical prompts are not independent. Nevertheless we feel that this is not likely to affect the validity of our results due to the (pseudo) randomness of the prompts given to the LLM. Therefore we use the Mann-Whitney U test as it is non-parametric (i.e., unlike the common student's t-test, it does not assume a particular distribution of values) to establish when our results cannot be reasonably attributed to simple luck.

From Table 1, we observe that GI-Agent variants consistently achieve superior solution quality compared to traditional GI. On the SAT4J benchmark, the median speedup for traditional GI is only 1.8%, while all GI-Agent variants exceed this. `ReflectSuccess` achieves the highest median speedup of 3.5% and ties with `ReflectAll` for the highest minimum improvement (1.2%), indicating both better central performance and improved reliability.

The Mann-Whitney U test confirms that for SAT4J (Java 2 500 LOC) all GI-Agent variants significantly outperform traditional GI ($p < 0.05$). However it does not distinguish results on MiniSAT (C++, 1000 LOC). Which suggests that reflective guidance, especially when conditioned on successful edits, can reliably lead to higher-quality solutions on more structured codebases.

To further understand the source of these improvements, we qualitatively examined the diffs produced by each method. Traditional GI primarily removes control-path conditions such as redundant reinitialisation logic (e.g., `if(!alreadylaunched || !this.keepHot)`), which reduces overhead but only yields modest gains. In contrast, GI-Agent introduces more impactful edits. For example, in MiniSAT, it removes calls to `simplify(outLearnt)`, a potentially expensive clause simplification routine, as well as debug-related listeners like `toDimacs(p)` during propagation. These edits streamline the solver's inner loop, optimizing hot paths that dominate runtime. This suggests that LLM-guided edits identify and remove deeper computational bottlenecks compared to shallow control flow changes made by traditional GI, especially when performance is governed by tight inner-loop logic.

To better understand the source of these gains, we also conducted a comparative diff analysis between traditional GI and GI-Agent on MiniSAT. Without LLM assistance, Magpie made a small number of superficial edits, mostly unwrapping one-line conditions and removing `/*auto*/` wrappers, but introduced a semantically redundant line that duplicated `learnts.shrink(i - j)`, indicating low edit precision and no substantive structural change. By contrast, GI-Agent made a key semantic modification: it disabled MiniSAT's restart mechanism by replacing the entire restart loop in `solve_()` with a single call to `search(-1)`. This prevents MiniSAT using periodic restarts, eliminating overhead from restart bookkeeping, garbage collection triggers, and heuristic resets. In a lean and efficient solver like MiniSAT, where many instances are already tractable, this "no-restart" strategy accelerates execution significantly. Additionally, GI-Agent applied a broader set of micro-optimizations across high-frequency code paths such as `propagate()` and `analyze()`, reducing branching and simplifying logic in inner loops. These edits compound at runtime, leading to a solution that is up to **3× faster** than traditional GI.

Also, we note that no statistically significant differences are observed among the three GI-Agent variants on either SAT4J or MiniSAT. Possibly because the main driver of improved solution quality is the integration of LLMs into the mutation and crossover processes, with the specific reflection strategy contributing marginally, if at all, to final performance.

We suggest the effectiveness of reflective mechanisms in GI-Agent may be task-dependent. I.e., they offer greater benefit on complex, modular codebases like SAT4J, where past edits provide meaningful structure to learn from, but yield limited additional gains on simpler systems like MiniSAT. Alternatively the LLM may be better able to cope with a verbose language like Java than with C++.

In summary (for **RQ1**), GI-Agent significantly outperforms traditional GI in terms of solution quality, particularly on structured and modular codebases. The LLM component contributes substantially to this improvement, while the role of reflection appears more nuanced and context-sensitive.

Table 1: Speedup improvements above baseline for 20 independent runs

	SAT4J			MiniSAT		
Method	Min	Median	Max	Min	Median	Max
GI	0.0%	1.8%	2.9%	0.0%	15.2%	81.0%
NoReflect	0.0%	2.9%	8.3%	**10.3%**	**34.9%**	**82.9%**
ReflectAll	**1.2%**	2.6%	**9.9%**	4.8%	30.8%	80.9%
ReflectSuccess	**1.2%**	**3.5%**	5.7%	9.8%	31.5%	80.8%

6.2 RQ2: How Efficiently Does GI-Agent Discover High-Quality Program Variants?

To answer RQ2 we analyze the rate at which each method finds improved program variants during the optimization process. Table 2 shows the median speedup at 20%, 40%, 60%, 80%, and 100% of the total 220 steps.

On the SAT4J benchmark, traditional GI is consistently the least efficient approach. It gets to its peak performance early, but reaches only 1.8% by the 60% step mark and remaining flat for the remainder of the search. In contrast, all GI-Agent variants demonstrate substantially improved efficiency. Among them, `ReflectSuccess` achieves the highest overall performance (3.5%) and converges early—reaching this value as early as the 40% step mark. This suggests that selectively reflecting on top-performing edits reinforces effective transformations and accelerates convergence. `NoReflect` and `ReflectAll` also improve upon traditional GI, though to a lesser extent. `ReflectAll` performs best at the very beginning (2.6% at 20%) but fails to improve further. `NoReflect`, on the other hand, improves more gradually and ultimately achieves a final speedup of 2.9%. Perhaps `ReflectAll`'s stagnation is due to the inclusion of low-quality and failed edits in its reflective context. While ReflectAll's diverse sampling strategy aims to increase robustness, it may also introduce noise and conflicting optimization signals, confusing the LLM and impeding learning. The LLM may struggle to reconcile mixed quality signals, particularly on structured systems like SAT4J, leading to early saturation. As expected, the Mann-Whitney U test confirms that for SAT4J all GI-Agent variants outperform traditional GI with high statistical confidence.

On MiniSAT, the efficiency dynamics are more nuanced. `NoReflect` exhibits the strongest early performance, achieving a 16.2% speedup by 20% of steps, suggesting that LLM-driven edits alone can quickly explore promising regions of the search space. However, `Reflect Success` quickly catches up, by 40% and 60% it surpasses `NoReflect`, peaking at 30.7%. Interestingly, after this point, `ReflectSuccess` appears to stall, while `NoReflect` continues to improve, ultimately reaching the highest final speedup (34.9%).

Despite this consistent performance advantage over traditional GI, no statistically significant differences are observed among the GI-Agent variants themselves at any step percentile, on either benchmark. The similarity of the our results for our three LLM approaches suggests that the primary driver of efficiency gains is the integration of the LLM into the mutation and crossover pipeline. While reflection mechanisms offer task-specific advantages, their relative impact is modest when compared to the advantage introduced by the LLM itself.

Perhaps the efficiency of GI-Agent arises from the LLM's ability to propose high-quality edits early in the search, reducing the need for extensive trial-and-error. On structured systems like SAT4J, reflective strategies such as ReflectSuccess may help reinforce useful transformation patterns and accelerate convergence. On simpler systems like MiniSAT, however, the added memory of prior edits may reduce exploration and increase the risk of local stagnation. In such cases, NoReflect maintains higher exploratory pressure, possibly enabling it to ultimately discover better solutions.

In summary (for **RQ2**), all GI-Agent variants discover high-quality variants more efficiently than traditional GI. ReflectSuccess provides the fastest improvement rate on SAT4J, while NoReflect achieves the strongest final outcomes on MiniSAT by sustaining exploration throughout the search.

6.3 RQ3: How Does GI-Agent Affect the Viability of Generated Program Variants?

To answer RQ3, we measure the the proportion of patches that both compile and pass all tests. In addition to successful outcomes, Table 3 also reports test errors (patches that compile but fail tests) and compilation errors (patches that fail to compile).

From Table 3, we observe that Magpie without LLMs suffers from relatively high failure rates. On SAT4J, only 54.2% of variants are viable, with nearly half of the remaining cases failing to compile (45.3%). On MiniSAT, it performs slightly better, reaching a 64.2% success rate, but on average still suffers from a high compile error rate of 32.1%.

In contrast, all GI-Agent variants dramatically improve the viability of generated patches. On SAT4J, they achieve success rates exceeding 91%, while on MiniSAT, success rates are consistently near or above 80%. These improvements are primarily due to substantial reductions in compile errors. For example, NoReflect reduces compile errors from 45.3% to just 3.0% on SAT4J, and from 32.1% to 7.0% on MiniSAT. This indicates that LLM-guided transformations are more likely to produce syntactically valid and type-safe edits, thereby reducing low-level failures in the compilation pipeline.

At the same time, we note a modest increase in runtime test errors across all GI-Agent variants. This suggests that while the patches generated by GI-Agent are more likely to compile, they may still introduce subtle semantic issues that lead to test failures. One possible explanation is that the LLM is biased toward generating syntactically plausible code snippets, which do not always preserve program behavior. Unfortunately, this trade-off (fewer compile-time issues but more test failures) is consistent with the observation that LLMs favour fluent generation over precise semantics unless

Table 2: Median speedup during SAT4J and MiniSAT GI runs. The values are the increase at 20%, 40%, 60%, 80%, and 100% of the whole run across 20 runs.

	SAT4J					MiniSAT				
Method	20%	40%	60%	80%	100%	20%	40%	60%	80%	100%
Traditional GI	0.6%	1.5%	1.8%	1.8%	1.8%	0.3%	7.8%	8.1%	14.9%	15.2%
NoReflect	1.8%	2.3%	2.6%	2.9%	2.9%	**16.2%**	21.3%	27.4%	28.0%	**34.9%**
ReflectAll	**2.6%**	2.6%	2.6%	2.6%	2.6%	6.0%	14.0%	29.2%	**30.7%**	30.8%
ReflectSuccess	2.3%	**3.5%**	**3.5%**	**3.5%**	**3.5%**	7.5%	**28.6%**	**30.7%**	**30.7%**	31.5%

Table 3: Viability of SAT4J and MiniSAT patches. Percentages are the median proportion of successful variants, test errors and compilation errors across 20 runs.

	SAT4J			MiniSAT		
Method	Success	Test Error	Compile Error	Success	Test Error	Compile Error
Traditional GI	54.2%	**0.5%**	45.3%	64.2%	**2.5%**	32.1%
NoReflect	**92.5%**	4.0%	**3.0%**	78.9%	12.4%	**7.0%**
ReflectAll	91.5%	3.5%	4.0%	**80.1%**	11.4%	7.2%
ReflectSuccess	91.5%	4.0%	4.0%	79.6%	10.0%	**7.0%**

explicitly constrained. Which is of course expensive and potentially labour intensive.

Mann-Whitney tests confirm for both SAT4J and MiniSAT all GI-Agent variants significantly outperform traditional GI across all three viability metrics ($p < 0.0001$ in every case). However, no statistically significant differences are found among the GI-Agent variants themselves. Suggesting that the choice of reflection strategy (whether to reflect on all past operations, only successful ones, or not at all) does not substantially affect viability. We hypothesize that the primary driver of improved viability is the LLM's learned understanding of programming syntax and structure, which allows it to avoid typical compilation pitfalls seen in traditional random-mutation-based GI. While reflection strategies may influence how the search is guided, their impact on syntactic correctness and basic functional behavior is comparatively minimal. Instead, the quality and safety of generated patches appear to depend more on the model's prior knowledge than on its memory of previous edits.

In summary (to answer **RQ3:**), GI-Agent significantly improves the viability of generated variants compared to Magpie without LLMs, achieving higher compilation and test-passing rates with both Java SAT4J and C++ MiniSAT. These improvements are largely attributable to the LLM's ability to produce syntactically correct code. Differences between reflection strategies are negligible in this regard, suggesting that all three variants are similarly good at producing viable patches.

6.4 RQ4: How Does the Use of Reflection Influence the Diversity of Generated Patches?

To answer RQ4, we analyse the structural diversity of the generated patches by measuring the percentage of repeated variants across different experimental runs. For each of GI-Agent's methods, we compute the minimum, median, and maximum repetition rates across the 20 runs. A higher repetition rate indicates lower diversity, (meaning the LLM) generates structurally identical patches more frequently, Table 4,

Somewhat surprisingly as shown in Table 4, Magpie without LLMs consistently generates the most diverse set of patches, with the lowest repetition rates across both benchmarks. Even so, on average it repeats more than a third of its mutants (median repetition rate 37.3% SAT4J and 36.6% MiniSAT). In contrast, all GI-Agent variants exhibit significantly higher repetition, with median values generally above 50%. This suggests that the addition of LLM-driven mutation and crossover, while improving viability and solution quality, leads to more convergent and less exploratory behaviour during search.

Among the GI-Agent variants, ReflectAll shows an intriguing edge case: while it achieves 0% repetition in the best case on SAT4J, its median and maximum repetition rates remain high (50.7% and 61.2%, respectively), indicating that the 0% scenario is rare and does not reflect its typical behaviour. This outcome could be the result of occasional high-variance runs where reflection injects sufficient diversity, but overall, the method does not appear to offer consistent improvements in structural variation. This may be due to ReflectAll's inclusion of both successful and unsuccessful edits; potentially introducing noise or conflicting optimization signals that obscure the LLM's ability to generalise diverse edits effectively. All differences between traditional GI and GI-Agent variants are statistically significant on both SAT4J and MiniSAT, with $p < 0.001$ for every pairwise comparison.

However, the Mann-Whitney test found no statistically significant differences between the GI-Agent variants themselves. This suggests that the specific reflection strategy (whether reflecting on all edits, only successful ones, or not using reflection at all) has limited impact on diversity outcomes. Instead, the dominant factor

Table 4: Minimum, median and maximum percentage of repeated patches for SAT4J and MiniSAT. Values are percentages of repeated variants across 20 runs.

	SAT4J			MiniSAT		
Method	Min	Median	Max	Min	Median	Max
Traditional GI	28.9%	**37.3%**	**44.3%**	**23.9%**	**36.6%**	**48.8%**
NoReflect	44.8%	52.5%	60.2%	32.8%	45.3%	53.2%
ReflectAll	**0.0%**	50.7%	61.2%	39.8%	44.8%	55.2%
ReflectSuccess	46.8%	51.2%	58.2%	35.5%	44.3%	50.7%

appears to be the shift from Magpie's stochastic mutation-based exploration to the more deterministic and convergent editing patterns produced by the LLM.

It is also important to note that the repetition values in Table 4 were calculated after applying a normalization pass to eliminate non-semantic differences. In the original computation, patches were compared as raw strings, including differences in whitespace and comments. In our (presented) analysis, we strip whitespace and remove comments before computing repetition. This refinement results in higher and more meaningful repetition values. On average, the minimum, median, and maximum repetition rates increased by 29.2%, 36.3%, and 163.3%, respectively. This emphasises that many patches differed only superficially, and that LLM-generated edits are often semantically similar or identical.

In conclusion (for **RQ4**), while GI-Agent variants outperform traditional GI in efficiency and quality, they do so at the cost of reduced diversity. The use of LLMs introduces more directed and convergent behaviour, leading to higher repetition rates. Reflection strategy has little influence on this trend, and future work may consider hybrid approaches that reintroduce diversity-promoting mechanisms without compromising solution quality.

7 Conclusions

We have presented GI-Agent, a novel framework that uses the power of Large Language Models (LLMs) to enhance code optimisation in Search-Based Software Engineering (SBSE). By integrating LLMs into the Genetic Improvement (GI) process (in particular into the widely used GI framework, Magpie), GI-Agent enables context-aware mutations and crossovers, reducing dependence on external code corpora and introducing iterative learning through reflection. Experimental results on the MiniSAT and SAT4J benchmarks demonstrate its capability to generate viable, performance-enhancing code variants, outperforming traditional GI. While GI-Agent employs few-shot learning for lightweight deployment and rapid adaptation, our investigations reveal that the reflection mechanism, despite its conceptual appeal, offered limited additional performance benefit in practice.

Overall, this work advances the synergy between LLMs and SBSE, laying the groundwork for adaptive, interpretable software optimization tools capable of autonomously evolving code with improved efficiency and effectiveness.

Acknowledgments

We would like to thank our anonymous referees.

References

[1] Kumail Alhamoud, Shaden Alshammari, Yonglong Tian, Guohao Li, Philip H.S. Torr, Yoon Kim, and Marzyeh Ghassemi. 2025. Vision-Language Models Do Not Understand Negation. In *Computer Vision and Pattern Recognition (CVPR 2025)*. IEEE, Nashville, TN, USA, 29612–29622. http://dx.doi.org/10.1109/CVPR52734.2025.02757

[2] Aymeric Blot and Justyna Petke. 2022. MAGPIE: Machine Automated General Performance Improvement via Evolution of Software. doi:10.48550/arXiv.2208.02811 arXiv:2208.02811 [cs].

[3] Dimitrios Stamatios Bouras, Sergey Mechtaev, and Justyna Petke. 2025. LLM-Assisted Crossover in Genetic Improvement of Software. In *14th International Workshop on Genetic Improvement @ICSE 2025*, Aymeric Blot, Vesna Nowack, Penn Faulkner Rainford, and Oliver Krauss (Eds.). Ottawa, 19–26. http://dx.doi.org/10.1109/GI66624.2025.00012 Best presentation.

[4] Alexander E. I. Brownlee, James Callan, Karine Even-Mendoza, Alina Geiger, Carol Hanna, Justyna Petke, Federica Sarro, and Dominik Sobania. 2023. Enhancing Genetic Improvement Mutations Using Large Language Models. In *SSBSE 2023: Challenge Track (LNCS, Vol. 14415)*, Paolo Arcaini, Tao Yue, and Erik Fredericks (Eds.). Springer, San Francisco, USA, 153–159. http://dx.doi.org/10.1007/978-3-031-48796-5_13

[5] Bobby R Bruce, Justyna Petke, and Mark Harman. 2015. Reducing energy consumption using genetic improvement. In *Genetic and Evolutionary Computation Conference, GECCO 2015*. Madrid, Spain, 1327–1334. http://dx.doi.org/10.1145/2739480.2754752

[6] Cursor Team. 2025. Cursor: The AI Code Editor. https://www.cursor.com/. Accessed: 2025-04-24.

[7] Gudny B. Saemundsdottir and Saemundur Oskar Haraldsson. 2024. Large Language Models as All-in-one Operators for Genetic Improvement. In *Genetic and Evolutionary Computation Conference Companion (GECCO 2024)*, Dominik Sobania and Aymeric Blot (Eds.). ACM, Melbourne, Australia, 727–730. http://dx.doi.org/10.1145/3638530.3654408

[8] Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. 2024. From LLMs to LLM-based Agents for Software Engineering: A Survey of Current, Challenges and Future. arXiv:2408.02479. https://arxiv.org/abs/2408.02479

[9] Jingyuan Wang, Carol Hanna, and Justyna Petke. 2025. Large Language Model based Code Completion is an Effective Genetic Improvement Mutation. In *14th International Workshop on Genetic Improvement @ICSE 2025*, Aymeric Blot, Vesna Nowack, Penn Faulkner Rainford, and Oliver Krauss (Eds.). Ottawa, 11–18. http://dx.doi.org/10.1109/GI66624.2025.00011 Best paper.

[10] Harshit Joshi, Jose Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radicek. 2023. Repair Is Nearly Generation: Multilingual Program Repair with LLMs. In *AAAI-2023*, Vol. 37. Washington, DC, USA, 5131–5140. https://doi.org/10.1609/aaai.v37i4.25642

[11] Sungmin Kang and Shin Yoo. 2023. Towards Objective-Tailored Genetic Improvement Through Large Language Models. In *12th International Workshop on Genetic Improvement @ICSE 2023*, Vesna Nowack, Markus Wagner, Gabin An, Aymeric Blot, and Justyna Petke (Eds.). IEEE, Melbourne, Australia, 19–20. http://dx.doi.org/10.1109/GI59320.2023.00013 Best position paper.

[12] LangChain. 2025. LangChain: Empowering Language Model Applications. https://www.langchain.com/. Accessed: March 25, 2025.

[13] William B. Langdon, Brian Yee Hong Lam, Marc Modat, Justyna Petke, and Mark Harman. 2017. Genetic Improvement of GPU Software. *Genetic Programming and Evolvable Machines* 18, 1 (March 2017), 5–44. http://dx.doi.org/10.1007/s10710-016-9273-9

[14] William B. Langdon, Brian Yee Hong Lam, Justyna Petke, and Mark Harman. 2015. Improving CUDA DNA Analysis Software with Genetic Programming. In *GECCO '15: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, Sara Silva et al. (Eds.). ACM, Madrid, 1063–1070. http://dx.doi.org/10.1145/2739480.2754652

[15] Daniel Le Berre and Anne Parrain. 2011. The Sat4j library, release 2.2: System description. *Journal on Satisfiability, Boolean Modelling and Computation* 7, 2-3

(2011), 59–64. https://doi.org/10.3233/SAT190075

[16] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 919–931. http://dx.doi.org/10.1109/ICSE48619.2023.00085

[17] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in neural information processing systems, NeurIPS 2020*, Vol. 33. virtual, 9459–9474. https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf

[18] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2025. SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. In *ICSE-2025*. IEEE Computer Society, Ottawa, Canada, 16–28. doi:10.1109/ICSE55347.2025.00129

[19] OpenAI. 2024. OpenAI API Documentation. https://platform.openai.com/docs/api-reference/introduction

[20] OpenAI. 2024. Prompt Caching Guide. https://platform.openai.com/docs/guides/prompt-caching.

[21] Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. 2014. Using Genetic Improvement and Code Transplants to Specialise a C++ Program to a Problem Class. In *EuroGP 2014 (LNCS, Vol. 8599)*, Miguel Nicolau et al. (Eds.). Springer, Granada, Spain, 137–149. http://dx.doi.org/10.1007/978-3-662-44303-3_12

[22] Justyna Petke, William B. Langdon, and Mark Harman. 2013. Applying Genetic Improvement to MiniSAT. In *Symposium on Search-Based Software Engineering (Lecture Notes in Computer Science, Vol. 8084)*, Guenther Ruhe and Yuanyuan Zhang (Eds.). Springer, Leningrad, 257–262. http://dx.doi.org/10.1007/978-3-642-39742-4_21 Short Papers.

[23] O. Rosenbaum. 2023. LLMs Don't Understand Negation. *HackerNoon* (2023). https://hackernoon.com/llms-dont-understand-negation Accessed: 25 March 2025.

[24] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *NeurIPS 2023* 36 (2023), 8634–8652. https://openreview.net/forum?id=vAElhFcKW6

[25] Shuzheng Gao, Cuiyun Gao, Wenchao Gu, and Michael Lyu. 2025. Search-Based LLMs for Code Optimization. In *ICSE 2025*. Ottawa, Canada. doi:10.1109/ICSE55347.2025.00021

[26] Simiao Zhang et al. 2024. Experimenting a New Programming Practice with LLMs. arXiv:2401.01062. https://arxiv.org/abs/2401.01062

[27] Niklas Sorensson and Niklas Een. 2005. MiniSat v1.13 – A SAT Solver with Conflict-Clause Minimization. *SAT* 2005, 53 (2005), 1–2.

[28] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Thirty-Sixth Conference on Neural Information Processing Systems, NIPS 2022* (New Orleans, LA, USA). Curran Associates Inc., New Orleans, USA, Article 1800, 14 pages. https://openreview.net/forum?id=_VjQlMeSB_J

[29] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *International Conference on Software Engineering (ICSE) 2009*, Stephen Fickas (Ed.). Vancouver, 364–374. http://dx.doi.org/10.1109/ICSE.2009.5070536 Winner ACM SIGSOFT Distinguished Paper Award. Gold medal at 2009 HUMIES. Ten-Year Most Influential Paper [30].

[30] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2019. It Does What You Say, Not What You Mean: Lessons From A Decade of Program Repair. ICSE 2019 Plenary Most Inflential Paper. https://conf.researchr.org/profile/icpc-2019/westleyweimer

[31] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. 2024. Large Language Models as Optimizers. In *International Conference on Learning Representations, ICLR 2024*. Vienna, Austria. https://iclr.cc/virtual/2024/poster/19209

[32] Zhiheng Xi et al. 2025. The rise and potential of large language model based agents: A survey. *Science China Information Sciences* 68, 2 (2025), 121101. https://doi.org/10.1007/s11432-024-4222-0