LLM-Assisted Crossover in Genetic Improvement of Software

1st Dimitrios Stamatios Bouras
Peking University, Beijing, China
dimitris.bouras@outlook.com

2nd Sergey Mechtaev
Peking University, Beijing, China
mechtaev@pku.edu.cn

3rd Justyna Petke
University College London, London, UK
j.petke@ucl.ac.uk

***Abstract*—This study explores the use of Large Language Models to improve the crossover process in genetic programming, as applied in the genetic improvement domain. Traditional crossover techniques typically combine parent variants by selecting modifications uniformly or even randomly, without consideration of contextual relevance, often resulting in inefficient searches and suboptimal solutions due to incompatible or redundant modifications. In contrast, our LLM-assisted crossover leverages context to select and combine edits from parent solutions that are more likely to work well together, with the goal of producing higher quality variants, accelerating optimization.**

We implemented this approach within MAGPIE, a unified genetic improvement framework. We evaluated against five traditional crossover methods across seven benchmarks, measuring performance on four key metrics: average ranking, best variant execution time, efficiency in reaching performance milestones, and viable variant count. Results show that LLM-assisted crossover achieved an average ranking of 2.27 (on a scale where 1 is best and 6 is worst), making it the top-performing method across benchmarks based on the quality of the optimal variants produced. The LLM-based approach also improved the fitness (execution time) by an average of 8.5% over the best variant produced by the traditional methods. In terms of efficiency, the LLM-assisted crossover required on average 25.6% fewer variants to reach 25%, 50%, 75%, and 100% of the final performance improvement compared to the traditional methods. Additionally, the LLM-assisted crossover produced 4.8% more viable variants across scenarios, including both source code modification and parameter tuning cases.

These findings suggest that LLMs can significantly enhance genetic improvement by guiding the crossover process toward more effective and viable solutions, providing motivation for further research in LLM-assisted search algorithms.

***Index Terms*—Crossover, Genetic Improvement, Genetic Programming, Large Language Models, SBSE, Search-Based Software Engineering, Parameter Tuning**

I. Introduction

The continuous drive for software performance optimization has become increasingly vital as hardware advancement rates have plateaued over the past two decades [1]. This shift has redirected focus toward enhancing software's non-functional properties to meet the increasing demand for improved software performance. Given the escalating complexity of modern software, research in software engineering has emphasized automated strategies to optimize software without manual intervention [2]. These strategies include program transformations, software parameter tuning, and compiler flag optimization, each aiming to navigate a vast search space of potential software variants to improve efficiency, execution speed, and robustness. Tools such as genetic improvement frameworks like `PyGGI` [3] and parameter tuning utilities like `irace` [4] and `ParamILS` [5] exemplify some of these approaches, offering structured methods for automated optimization across a wide array of software applications.

Genetic improvement (GI) is a prominent technique for optimizing software, offering a structured approach to evolving software variants toward optimal solutions. At the heart of GI lies the search strategy, most frequently genetic programming (GP) [6]. The crossover operation plays a crucial role in GP – modifications from parent variants are combined to produce new offspring variants. However, traditional crossover methods, which typically rely on random or uniform selection of modifications from parent variants, often struggle to yield high-quality results efficiently. In the context of source code modification, randomly combining edits can lead to poorly integrated code, resulting in compilation or test failures. Similarly, in parameter tuning, incompatible parameter combinations may be selected, further hindering the effectiveness of crossover. Consequently, traditional crossover approaches can be both ineffective at identifying beneficial edits and inefficient at navigating the solution space, underscoring the need for more intelligent, context-aware crossover mechanisms.

The recent advancements in Large Language Models (LLMs) offer a promising solution to this challenge. LLMs such as GPT-4, trained on vast amounts of text and code, are capable of generating complex code structures, taking into account their context. By leveraging this contextual power, an LLM-assisted crossover approach could theoretically overcome the limitations of conventional crossover methods. Specifically, an LLM could be used to select and combine only those modifications that are likely to integrate well (as such patterns have been observed in large amounts of code), resulting in higher-quality, compilable variants and parameter combinations that have multiplicative effects on the fitness. We hypothesize that LLM-assisted crossover not only has the potential to produce more performant variants but also to increase the number of viable variants by avoiding code combinations that lead to compilation or runtime errors. Furthermore, as AI technology has rapidly advanced in recent years, we now have access to powerful models that can be integrated into genetic improvement frameworks to enhance traditional search-based optimization techniques.

In this work, we implement and evaluate LLM-assisted crossover within the MAGPIE [7] framework. MAGPIE is a flexible and extensible framework for search-based software improvement, designed to support a wide range of modification types, including statement-level changes and parameter tuning. In our experiments, we apply the LLM-assisted crossover to 7 benchmark programs, using both parameter tuning and direct source code modification to assess its effectiveness.

We aim to answer the following questions: Can LLM-assisted crossover improve the quality of generated software variants? Can it accelerate the discovery of optimal solutions compared to traditional crossover techniques and can it lead to more viable variants? By exploring these questions, we aim to demonstrate that LLMs can enhance the genetic improvement process, making it a more efficient and effective tool for software performance optimization.

Our results show that LLM-assisted crossover achieved an average ranking of 2.27 (on a scale where 1 is best and 6 is worst), making it the top-performing method across benchmarks based on the quality of the optimal variants produced. The LLM-based approach also improved the fitness (execution time) by an average of 8.5% over the best variant produced by the traditional methods. In terms of efficiency, the LLM-assisted crossover required 25.6% fewer variants to reach 25%, 50%, 75%, and 100% of the final performance improvement compared to the average of the traditional methods. Additionally, the LLM-assisted crossover produced 4.8% more viable variants across scenarios, including both source code modification and parameter tuning cases. Hence we recommend its use in future work.

II. Background

In this section we introduce genetic improvement, role of crossover, and the MAGPIE framework.

A. Genetic Improvement and Crossover

Genetic Improvement (GI) is a subset of search-based software engineering that uses evolutionary algorithms to enhance software by modifying its source code, parameters, or other configurations. Genetic programming (GP) is a core technique in GI, employing operations like mutation and crossover to evolve software variants that exhibit improved performance or functionality. Crossover, specifically, is a key operation in GP where modifications from two or more parent solutions are combined to create offspring solutions. This process is integral to exploring the solution space, as it enables beneficial traits from multiple parents to be recombined in novel ways.

Numerous crossover methods have been explored in genetic programming [8]. In this study, we focus on five prevalent crossover techniques to evaluate their effectiveness in generating viable, high-performing variants, comparing them to an LLM-assisted approach.

- **UniformConcat:** This method interleaves edits from two parent variants uniformly, aiming for a balanced combination of traits.
- **Concat:** Here, edits from one parent are appended with edits from the other, allowing for a straightforward concatenation of traits.
- **1Point:** A single crossover point is chosen in both parents, with edits swapped at this point to create new variants.
- **2Point:** Two crossover points are selected, allowing for a more flexible crossover where edits between the points are exchanged.
- **UniformInter:** This method selects edits from both parents in a uniform yet interleaved manner, aiming to combine traits without fixed crossover points.

These methods represent a variety of traditional approaches to crossover, offering different ways of blending parental modifications. However, as discussed earlier, they may lack the contextual awareness needed for integrating complex modifications effectively.

B. Large Language Models (LLMs)

Large Language Models (LLMs) are advanced machine learning models trained on massive datasets, allowing them to generate human-like text and complex language structures. LLMs like GPT-4 and its variants have been trained on extensive datasets that include both natural language and source code, enabling them to leverage context to produce human-like code snippets. This capability makes LLMs highly valuable for tasks that require context to be taken into account.

In this study, we utilize **GPT-4o-mini**, a streamlined variant of the GPT-4 model that maintains much of the original model's performance in code-related tasks while operating with reduced computational requirements. GPT-4o-mini serves as the core of our LLM-assisted crossover approach, where it is tasked with selecting and combining beneficial edits from parent solutions. By leveraging the LLM's contextual awareness, we aim to produce not only higher-quality variants but also a greater number of viable ones, as the LLM can potentially avoid incompatible modifications that might lead to compilation errors or test failures.

C. MAGPIE Framework

MAGPIE is a flexible framework for search-based software improvement [7], offering a structured platform for exploring both structural and parametric optimizations. Modifications are captured as **edits**, which represent the changes applied to source code or parameter files. An edit may involve actions such as inserting, replacing, or deleting code statements or adjusting parameter values.

MAGPIE represents programs in a language-agnostic XML format, enabling it to handle benchmarks in various programming languages. This edit-sequence-based approach decouples the search process from specific improvement techniques, making MAGPIE adaptable to a broad range of optimization tasks. Our experiments utilize MAGPIE's GP search strategy, enhanced by an LLM-assisted crossover implementation. This setup enables us to compare traditional crossover techniques

with our LLM-driven approach, assessing its ability to produce high-quality solutions efficiently. To evaluate fitness, we employ the `Linux` time command, aiming to reduce the runtime of the selected benchmarks. By integrating LLMs into MAGPIE, we leverage contextual knowledge to improve the selection and combination of edits, enhancing the overall optimization process.

III. LLM-Assisted Crossover Operator

The LLM-assisted crossover operator introduces an API call at the heart of the crossover process, where the LLM is used to intelligently select and combine edits from parent solutions. The LLM receives a selection of parent solutions, in the form of a list of modifications, along with their fitness scores, and evaluates which modifications from each parent, if any, should be applied to create an optimized child variant.

To enhance the LLM's decision-making, we also provide optional contextual information, which varies based on the type of optimization task. For source code modifications, the LLM receives the file being modified, allowing it to evaluate the parent modifications in the context of the code structure and syntax. For parameter tuning, we include relevant documentation describing each parameter, including its purpose, possible values, and constraints. This contextual information helped ensure that the LLM applied modifications correctly, especially in cases where certain parameter combinations could lead to non-viable configurations.

A critical consideration in designing the LLM-assisted crossover was preventing unwanted mutations. Since the goal of crossover is to combine existing modifications rather than introduce new ones, we introduce a post-processing stage to verify that all edits in the child variant were sourced from at least one of the selected parent solutions. This post-processing step ensures that the LLM does not introduce hallucinations or extraneous modifications that could lead to unintended behavior, particularly in scenarios where the LLM might attempt to apply a novel change outside the scope of parent modifications.

Overall, the LLM-assisted crossover method allows for a more context-aware selection of edits, with the LLM using parent fitness scores and optionally provides documentation to guide the choices. This setup enabled us to evaluate whether LLM-driven crossover could generate more effective and viable variants compared to traditional methods.

A. Motivational Example: LLM-Assisted Crossover in Parameter Tuning for Zlib Compression

To illustrate the potential of LLM-assisted crossover, consider the task of tuning parameters to increase runtime speed in the Zlib compression library. Zlib offers various configuration parameters, such as 'level' and 'memLevel', which control aspects of compression quality and memory usage. While achieving the fastest possible speed would obviously affect the compression quality, our focus on this example is on achieving some level of compression while prioritizing speed as the primary and only objective.

Suppose we have two promising parent configurations:

Parent 1:

- `ParamSetting(('zlib.params', 'param', 'wbits'), 30)`
- `ParamSetting(('zlib.params', 'param', 'level'), 2)`

Parent 2:

- `ParamSetting(('zlib.params', 'param', 'strategy'), 2)`
- `ParamSetting(('zlib.params', 'param', 'memLevel'), 8)`

In these configurations, Parent 1 sets 'level' to a low value (2), favoring speed over compression quality, while Parent 2 sets 'memLevel' to a high value (8), allowing for faster processing at the expense of increased memory usage. Both configurations effectively prioritize speed, each addressing a different aspect: compression level and memory allocation.

With many traditional crossover techniques, a child variant generated from these parents might appear as follows:

- `ParamSetting(('zlib.params', 'param', 'strategy'), 2)`
- `ParamSetting(('zlib.params', 'param', 'wbits'), 30)`

This child variant loses both of the beneficial configurations related to 'level' and 'memLevel', potentially resulting in a less optimal solution for runtime improvement.

In contrast, with LLM-assisted crossover, especially when provided with documentation context, the LLM is more likely to connect 'level' and 'memLevel' with execution speed. Consequently, it has a high probability of including both of these modifications in the child variant. By selecting configurations that are contextually relevant to speed optimization, the LLM-assisted crossover can create a more effective child variant faster, reducing the need to experiment with suboptimal configurations.

LLM's Explanation: To create a child program that optimizes runtime based on the provided parents and their edits, I will select the following edits:

- `ParamSetting(('zlib.params', 'param', 'level'), 2)` is beneficial because it lowers the compression level, resulting in faster compression.
- `ParamSetting(('zlib.params', 'param', 'memLevel'), 8)` is advantageous as it increases the memory level, which can improve performance and thus lower runtime.

IV. Research Questions

This study examines the potential of LLM-assisted crossover in genetic programming within the MAGPIE framework by addressing the following research questions:

A. RQ1: Does LLM-assisted crossover lead to higher-quality variants than traditional crossover operators?

Our first objective is to determine if LLM-assisted crossover produces higher-quality software variants compared to traditional crossover methods. Specifically, we aim to assess

whether LLM-assisted crossover achieves lower fitness values (execution times), indicating better-performing solutions across various benchmarks. We will not only be comparing the optimal variant that the GP process produces but all the variants that are produced along the way. Additionally, we compare the overall ranking of each crossover method for each benchmark, with the goal of identifying whether the LLM-based approach consistently outperforms traditional methods in generating high-quality variants.

B. RQ2: Which crossover method finds improved variants most efficiently?

The second research question focuses on the efficiency of each crossover method in quickly identifying improved variants. Here, we aim to understand whether LLM-assisted crossover can reach significant performance milestones, such as partial or full achievement of the best possible improvement, more rapidly than traditional approaches. Our goal is to establish whether the LLM approach enables faster convergence to optimized solutions.

C. RQ3: Does LLM-assisted crossover generate more viable variants than traditional crossover operators?

The final research question investigates the viability of the generated variants. We seek to determine whether LLM-assisted crossover produces a higher proportion of viable solutions, variants that successfully compile, run and pass tests, across both source code modification and parameter tuning scenarios. By assessing viability across different modification types, we aim to understand if the LLM approach enhances the reliability of generated solutions, particularly in scenarios where traditional crossovers may struggle with functional integration.

V. Methodology

Next, we present our methdology to answer our RQs.

A. Experimental Design

To evaluate the effectiveness of LLM-assisted crossover, we conducted experiments using the MAGPIE framework on a set of carefully selected benchmarks. The experimental setup was designed to assess both the quality and efficiency of the generated variants across various crossover methods.

1) Benchmarks: Our experiments with `MAGPIE` utilized benchmarks from Blot and Petke [7]: `MiniSAT`, `MiniSAT_hack`, `SAT4J`, `Weka`, and `LPG`, selected for their proven suitability for meaningful optimizations within `MAGPIE`. We also included `scipy.optimize.minimize` and `zlib.compress` to introduce diverse computational challenges. While all scenarios yielded mutants with improvements over the initial configuration, our focus lies on comparing the performance of different crossover methods rather than assessing MAGPIE's overall optimization capabilities.

In total, we configured and executed 7 different benchmarks across various functionality and programming languages. For three benchmarks (`scipy`, `zlib`, and `LPG`), only parameter tuning was conducted, as `scipy` and `zlib` lack obvious source code improvement opportunities, and `LPG` was incompatible with our system's compiler. For the remaining four benchmarks, both parameter tuning and direct source code modifications were performed, resulting in a total of 11 distinct scenarios. More details for the benchmarks are presented in Table I. We start with the default parameter values and allow for all configurable parameters to be tuned.

2) LLM Model: The LLM-assisted crossover used in this study relies on **GPT-4o-mini-2024-07-18**, a lightweight version of GPT-4 specifically designed to balance model complexity and computational efficiency. GPT-4o-mini was selected for its good performance in code-related tasks and reduced computational requirements, making it feasible to incorporate in, iterative crossover operations.

3) Genetic Programming Configuration: The genetic programming (GP) configuration used in our experiments was as follows:

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

The experiments were conducted over a total of 11 epochs, so that crossover would be performed 10 times. This configuration, particularly the high crossover rate of 0.6, was chosen to emphasize the crossover operation's influence on the generated variants. By setting the crossover rate relatively high, we aimed to create a wide but shallow search, where the effects of an effective crossover method would be more apparent.

In each epoch, crossover plays a significant role in the generation of new variants, with 60% of the offsprings resulting from crossover rather than mutation or elitism. This setup allows us to directly assess the effectiveness of the LLM-assisted crossover compared to traditional methods, particularly in its ability to generate high-quality and viable variants efficiently. All the new variants are tested to ensure that the changes don't break functionality.

4) LLM-Assisted Crossover Implementation: In addition to running MAGPIE with five traditional crossover methods, we created a sixth version of crossover driven by an LLM, specifically `GPT-4o-mini-2024-07-18`. The implementation follows the approach outlined in Section III. Specifically, our LLM prompts include the following elements:

- **Optimization Objective and Fitness Metric**: Information on what we are optimizing for, with the fitness defined as runtime.
- **Parent Variants**: Each parent variant is provided as a list of modifications, along with its respective fitness score, to guide the LLM in selecting beneficial edits.
- **Target File Context**: The file to which the modifications apply, specified as either a parameter file or a source code file, to give LLM more context.
- **Program Documentation**: For parameter tuning tasks, we include detailed documentation explaining each parameter, its possible values, default settings, and any

TABLE I
SELECTED BENCHMARKS FOR EXPERIMENTS

Benchmark	Description	Modification Type (with LOC)	Language
MiniSAT Hack	SAT solver	Parameter tuning, source code (file: `sources/core/Solver.cc` 742 LOC)	C++
MiniSAT	SAT solver	Parameter tuning, source code (file: `core/Solver.cc`, 732 LOC)	C++
WEKA	Data mining tool	Parameter tuning, source code (file: `classifiers/trees/RandomForest.java` 312 LOC)	Java
zlib	Compression library	Parameter tuning (functions: `compressobj`, `decompressobj`)	Python
SciPy	Scientific computing	Parameter tuning (function: `optimize.minimize`)	Python
Sat4j	Boolean satisfiability	Parameter tuning, source code (file: `sat4j/minisat/core/Solver.java` 1821 LOC)	Java
LPG	AI planner	Parameter tuning	C

constraints or dependencies between parameters. This additional context helps the LLM make informed decisions about parameter combinations.

The LLM is then instructed to generate a child variant by combining modifications from the parent variants to create the fittest possible child, specifically aimed to minimize runtime.

B. Computational Environment and Overhead

All experiments were conducted on a Ubuntu Linux 22.04 desktop machine equipped with an 8-core AMD Ryzen 7 4700 CPU and 16 GB of RAM. This setup provided sufficient computational resources to execute the experiments within a reasonable timeframe.

The LLM API calls to `GPT-4o-mini-2024-07-18` incurred minimal financial and time overhead. On average, each API call involved 1980 input tokens and 103 output tokens, resulting in a cost of $0.0003618 per call. With 120 API calls required per benchmark, the total cost per benchmark was less than half a penny ($0.0043618), making the approach highly affordable for iterative experiments.

Regarding time overhead, the average time per API call was measured at 2.92 seconds, leading to a total time overhead of approximately 350.4 seconds per benchmark. Given that the testing and the evaluation of the fitness of each mutant typically takes over 20 seconds, this additional overhead is relatively low, particularly since only 60% of the generated mutants undergo crossover. This balance between efficiency and cost demonstrates the feasibility of integrating LLM-assisted crossover into genetic programming workflows without significant resource consumption.

VI. RESULTS AND DISCUSSION

Next, we present and analyse the results of our study.

A. RQ1: Does LLM-assisted crossover lead to higher-quality variants than traditional crossover operators?

To address RQ1, we compared the quality of the final variants produced by each crossover method. We used the **average optimal fitness** of the best variants as the primary quality measure and included the **average fitness** across all variants to evaluate overall performance. The **average fitness** was firstly calculated per benchmark and then the average for all benchmarks was produced, to avoid giving higher weights to benchmarks with more viable variants. Additionally, we ranked each method from best to worst on each benchmark based on the optimal variant's execution time, then calculated an **average ranking** across all benchmarks. A crossover method with lower average optimal fitness, lower average fitness, and higher rankings (closer to 1 than 6) indicates a more effective approach.

TABLE II
COMPARISON OF CROSSOVER METHODS BASED ON EXECUTION TIME, AVERAGE FITNESS, AND RANKING ACROSS BENCHMARKS (RQ1)

Crossover Method	AOF(s)	AF(s)	Avg Ranking
UniformConcat	4.598	6.094	3.00
Concat	4.734	5.841	3.55
1Point	4.971	6.333	3.82
2Point	5.169	6.463	4.27
UniformInter	4.991	6.348	4.09
LLM-Assisted	**4.477**	**5.834**	**2.27**

AOF (s): Average Optimal Fitness in seconds.
AF (s): Average Fitness of all variants in seconds.

Table II presents a summary of the average optimal fitness, average fitness scores, and average rankings for each of the six crossover methods across the benchmarks.

The results in Table II show that the LLM-assisted crossover method achieved the lowest average optimal fitness time at 4.477 seconds and the highest average ranking at 2.27 across all benchmarks. Additionally, it produced the lowest average fitness score across all variants (5.834 s), indicating that it consistently generated high-quality variants, not just a single top-performing variant. In comparison, the UniformConcat method achieved the second-best average ranking (3.0) but had a higher average best execution time and average fitness score (4.598 and 6.094 seconds, respectively).

These results suggest that LLM-assisted crossover is not only effective in finding the best-performing variants but also reliable in generating multiple high-quality variants on average, outperforming traditional methods in both quality and consistency.

Answer to RQ1: The LLM-assisted crossover method produced higher-quality variants. It led to the lowest average execution time, being 8.5% faster on average than the 5 traditional crossover methods, while the average variant from the LLM-assisted experiments was 6.1% faster than the average variant in the experiments with the traditional crossover methods. The LLM-assisted approach also accomplished the best average ranking (2.27) demonstrating its effectiveness in generating both high-performing and consistent variant quality across benchmarks.

B. RQ2: Which crossover method finds improved variants most efficiently?

To evaluate the efficiency of each crossover method in reaching improved variants, we analyzed the number of variants required to achieve incremental improvements relative to the optimal variant found in each benchmark. Specifically, we measured how many variants each crossover method needed, on average, to reach 25%, 50%, 75%, and 100% of the best possible improvement in execution speed. Here, 25% improvement signifies that the variant achieved 25% of the total potential improvement in speed reduction relative to the optimal variant. For each crossover method, we calculated the average index of the variant at which these milestones were reached and then averaged the results across all benchmarks.

If a crossover method failed to reach any of these milestones (25%, 50%, 75%, or 100%) in a benchmark, it was assigned a value of 250 variants for that milestone, as the maximum number of variants allowed was 220. This penalty discourages methods that failed to reach specific improvement milestones, emphasizing the importance of finding efficient paths toward high-performing variants.

Table III presents the average variant index required by each crossover method to reach each improvement milestone across all benchmarks.

The results in Table III indicate that the LLM-assisted crossover method reached each performance milestone faster than traditional crossover methods, as evidenced by the lower average variant indices across all improvement levels, with the LLM method needing on average 25.6% less variants. For example, LLM-assisted crossover required an average of 39.82 variants to reach 25% of the best improvement and 209.18 variants to reach 100% of the best improvement, outperforming the next closest method, UniformConcat, which required 56.45 and 227.27 variants, respectively, for these milestones. In cases where a method did not reach certain milestones within the benchmark constraints, the penalty of 250 variants was applied, impacting the overall efficiency score.

Answer to RQ2: The LLM-assisted crossover method found improved variants more quickly than traditional methods, reaching performance milestones with 25.6% fewer generated variants on average.

C. RQ3: Does LLM-assisted crossover generate more viable variants than traditional crossover operators?

To answer RQ3, we compared the number of **viable variants** (which are variants that successfully compiled, run and passed tests), produced by each crossover method across both source code modification and parameter tuning scenarios. While parameter tuning typically results in a higher proportion of viable variants, certain parameter combinations can still lead to non-viable outcomes due to compatibility constraints. In the case of source code modifications, structural incompatibilities often pose a greater challenge, leading to non-viable variants. Therefore, the number of viable variants serves as a key metric for evaluating the effectiveness of the LLM-assisted crossover in generating functional solutions across a variety of modification types.

Given our experimental setup of 11 epochs with a population size of 20, the theoretical maximum number of viable variants for each crossover method is 220. Table IV presents the average number of viable variants generated by each crossover method across all modification scenarios.

The results in Table IV show that the LLM-assisted crossover generated the highest average number of viable variants (194.45), approaching the theoretical maximum of 220. This suggests that LLM-assisted crossover is more effective at maintaining functionality across both source code and parameter modifications, likely due to its context-aware selection process, which minimizes structural and compatibility issues. In comparison, the next highest average number of viable variants was achieved by UniformConcat (187.73), which, while effective, still produced 3.5% fewer viable outcomes than the LLM-assisted approach. On average the LLM-assisted approach produced 4.8% more viable variants than the other methods.

Answer to RQ3: The LLM-assisted crossover generated the highest number of viable variants across all modification scenarios, with an average of 4.8% more viable variants generated compared to the traditional crossover methods, demonstrating its ability to effectively maintain functionality in both source code and parameter tuning cases.

VII. Threats to Validity

Several factors may affect the generalizability and robustness of our findings in this initial exploration of LLM-assisted crossover in GP. First, all experiments were conducted on a single computer with limited computational and financial resources, which restricted us to using only one LLM model, `GPT-4o-mini`. The potential impact of different LLMs on crossover effectiveness remains unexplored, and in future work we plan to evaluate additional models to provide broader insights.

The scope of benchmarks and scenarios is also limited, as our experiments were performed on seven benchmarks across 11 distinct scenarios, covering both parameter tuning and source code modification cases. Although these benchmarks provide a varied sample, they do not encompass the full range of possible optimization contexts or programming languages, which may affect the generalizability of the results. Moreover, this study was conducted exclusively within the MAGPIE framework, without testing on other search-based software improvement tools. As MAGPIE's representations, fitness evaluations, and genetic operators are specific to its design, evaluating LLM-assisted crossover within other frameworks could provide additional perspectives on its effectiveness.

Another factor to consider is the inherent randomness in genetic programming, particularly in selection, mutation, and crossover processes. Due to time and resource constraints, we conducted only a single run of each experiment, without

TABLE III
COMPARISON OF CROSSOVER METHODS BASED ON EFFICIENCY IN REACHING IMPROVEMENT MILESTONES (RQ2)

Crossover Method	25% Improvement	50% Improvement	75% Improvement	100% Improvement
UniformConcat	56.45 variants	68.55 variants	112.27 variants	227.27 variants
Concat	72.91 variants	105.82 variants	140.73 variants	228.73 variants
1Point	70.18 variants	78.55 variants	179.45 variants	242.27 variants
2Point	97.73 variants	129.64 variants	170.18 variants	230.73 variants
UniformInter	77.36 variants	130.27 variants	135.18 variants	244.91 variants
LLM-Assisted Crossover	**39.82 variants**	**48.27 variants**	**119.00 variants**	**209.18 variants**

Note: The values in each column represent the average number of variants required by each crossover method to reach 25%, 50%, 75%, and 100% of the maximum observed speed improvement across benchmarks. If a crossover method failed to reach a specific milestone within the allowed 220 variants, a penalty value of 250 variants was assigned for that milestone.

TABLE IV
COMPARISON OF CROSSOVER METHODS BASED ON NUMBER OF VIABLE VARIANTS ACROSS ALL MODIFICATION SCENARIOS (RQ3)

Crossover Method	Average Number of Viable Variants
UniformConcat	187.73
Concat	186.27
1Point	182.18
2Point	185.73
UniformInter	185.55
LLM-Assisted Crossover	**194.45**

repetitions to account for random variations. As a result, our findings may be influenced by specific random outcomes, and repeating these experiments in future work would help confirm the consistency of the results.

Additionally, while LLM responses are not strictly deterministic, some degree of variance in LLM-assisted crossover can actually be beneficial. The probabilistic nature of LLMs introduces variability in responses, which may occasionally lead to highly beneficial edits or, conversely, to suboptimal modifications. However, this diversity allows genetic programming to explore a broader range of solutions, increasing the chance of discovering effective variants over time. Our approach does not rely on consistently optimal responses from the LLM; rather, it aims to produce variants that, on average, are of higher quality than those generated by traditional crossover methods. Running the same benchmarks multiple times would provide additional data on the robustness of LLM-assisted crossover, but in this specific context, the natural variance can enrich the search process and contribute positively to genetic programming's overall effectiveness.

This study serves as an initial investigation to gauge the potential of LLM-assisted crossover. Our findings suggest promising benefits for variant quality and viability, motivating further research in this direction. Future work will expand this study by including additional benchmarks, experimenting with different LLM models, and conducting multiple repetitions to enhance the robustness and generalizability of the results.

VIII. RELATED WORK

Genetic algorithms (GAs) are optimization methods inspired by the principles of natural evolution, initially developed by Holland [9]. Within GAs, crossover is a crucial operator that facilitates both exploration and exploitation of the solution space by combining genetic information from parent solutions. Numerous crossover methods, such as single-point, two-point, multi-point, and uniform crossover, have been proposed, each offering different benefits depending on problem context and population characteristics [10], [11]. For instance, Hasançebi and Erbatur [12] found that two-point crossover often yields better outcomes for larger populations, while Syswerda [10] suggests that uniform crossover can be advantageous in certain scenarios. These insights underscore the importance of selecting appropriate crossover techniques to suit the specific requirements of a problem.

While traditional crossover techniques rely on structural manipulations, recent research has focused on integrating semantic or context-aware knowledge into the crossover process. Beadle and Johnson's Semantically Driven Crossover (SDC) [13] is a notable example, using reduced ordered binary decision diagrams to ensure that child programs exhibit behavioral differences from their parents. This semantically guided approach reduces redundancy in crossover operations, leading to improved performance and reduced code bloat. However, SDC is specifically tailored for semantic equivalence checking, which makes it computationally expensive for large-scale applications. In contrast, our LLM-assisted crossover operates at a higher level, leveraging domain knowledge and contextual information to produce effective solutions without the overhead of fine-grained semantic checks.

Similarly, Krawiec and Pawlak's Locally Geometric Semantic Crossover (LGX) [14] introduces a method that considers semantic properties when creating offspring, generating behaviorally intermediate solutions by selecting homologous regions in parent programs. While LGX enhances search efficiency by focusing on semantically meaningful regions, it is inherently tied to geometric relationships in the program's semantic space. Our LLM-assisted crossover, by contrast, uses pre-trained language models to interpret high-level contextual information, making it more adaptable across diverse problem domains without requiring explicit semantic mappings.

Shem-Tov and Elyasaf propose a Deep Neural Crossover (DNC) operator that leverages deep reinforcement learning and an LSTM-based encoder-decoder to guide gene selection in crossover operations [15]. DNC applies an attention-driven policy that dynamically selects genes to maximize offspring fitness, achieving higher solution quality and convergence rates. However, DNC requires significant pre-training and fine-

tuning for specific problem domains, which can be resource-intensive and limits its immediate applicability. Our LLM-assisted crossover, in contrast, is designed to work out of the box, requiring minimal setup and no additional training, making it a more practical solution for a wide range of optimization tasks.

The study by Hameed and Kanbar [16] also highlights the importance of adaptive crossover strategies, demonstrating that the performance of different crossover operators varies significantly depending on problem characteristics. Their findings reinforce the need for customized crossover techniques, supporting the motivation for our LLM-assisted approach, which dynamically adapts to diverse optimization scenarios through its ability to leverage contextual insights.

IX. Conclusion

This study provides an initial exploration into using LLMs to assist in the crossover process of genetic programming, aiming to enhance variant quality and accelerate optimization.

By integrating an LLM into the MAGPIE framework, we observed several key improvements with LLM-assisted crossover compared to traditional methods: LLM-assisted crossover achieved an average ranking of 2.27 across benchmarks (where 1 is best and 6 is worst), making it the top-performing method based on the quality of the optimal variants produced. The LLM-based approach improved the execution time of the best variant by an average of 8.5% over the best variant produced by traditional crossover methods. In terms of efficiency, LLM-assisted crossover required 25.6% fewer variants on average to reach 25%, 50%, 75%, and 100% of the final performance improvement, compared to the average of traditional methods. Additionally, the LLM-assisted crossover produced 4.8% more viable variants across all scenarios, encompassing both source code modification and parameter tuning cases.

These results suggest that LLMs can effectively leverage contextual information to guide the search process toward optimal solutions, producing not only higher-quality variants but also a greater number of viable solutions and with greater efficiency in reaching performance milestones.

While our findings are promising, this work serves primarily as motivation for further investigation into LLM-assisted crossover. Expanding this approach with a broader range of benchmarks, additional LLM models, and repeated experiments will help validate and refine the benefits observed here. Future research can also explore integrating LLMs into other stages of genetic programming like mutation, or testing on diverse optimization frameworks. This study lays the groundwork for continued exploration of LLM-assisted techniques, encouraging broader applications and more consistent use of LLMs in genetic improvement and other evolutionary computation methods.

X. Data Availability

All code, documentation, results and complimentary material for this work is available in our repository: https://github.com/SOLAR-group/LLM_Assisted_Crossover.

References

[1] M. M. Waldrop, "More than moore," *Nature*, vol. 530, pp. 145–147, 2016.

[2] S. Eldh, J. Brandt, M. Street, H. Hansson, and S. Punnekkat, "Towards fully automated test management for large complex systems," in *2010 Third International Conference on Software Testing, Verification and Validation*, 2010, pp. 412–420.

[3] G. An, A. Blot, J. Petke, and S. Yoo, "Pyggi 2.0: language independent genetic improvement framework," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 1100–1104. [Online]. Available: https://doi.org/10.1145/3338906.3341184

[4] M. López-Ibáñez, J. Dubois-Lacoste, L. Pérez Cáceres, T. Stützle, and M. Birattari, "The irace package: Iterated racing for automatic algorithm configuration," *Operations Research Perspectives*, vol. 3, pp. 43–58, 2016.

[5] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stuetzle, "Paramils: An automatic algorithm configuration framework," *Journal of Artificial Intelligence Research*, vol. 36, p. 267–306, Oct. 2009. [Online]. Available: http://dx.doi.org/10.1613/jair.2861

[6] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward, "Genetic improvement of software: A comprehensive survey," *IEEE Transactions on Evolutionary Computation*, vol. 22, no. 3, pp. 415–432, 2018.

[7] A. Blot and J. Petke, "Magpie: Machine automated general performance improvement via evolution of software," 2022.

[8] X. Yu and M. Gen, "Simple evolutionary algorithms," pp. 11–38, 2010. [Online]. Available: https://doi.org/10.1007/978-1-84996-129-5_2

[9] J. Holland, *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[10] G. Syswerda, "Uniform crossover in genetic algorithms," in *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann, 1989, pp. 2–9.

[11] D. Jong and W. Spears, "An analysis of the interacting roles of population size and crossover in genetic function optimization," in *Parallel Problem Solving from Nature*. Springer, 1990, pp. 38–47.

[12] O. Hasançebi and F. Erbatur, "A revised comparison of crossover and mutation in genetic programming," *Computers & Structures*, vol. 78, no. 4, pp. 435–448, 2000.

[13] L. Beadle and C. G. Johnson, "Semantically driven crossover in genetic programming," in *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, 2008, pp. 111–116.

[14] K. Krawiec and T. Pawlak, *Locally Geometric Semantic Crossover : A Study on the Roles of Semantics and Homology in Recombination Operators*, 2013.

[15] E. Shem-Tov and A. Elyasaf, "Deep neural crossover," *Genetic and Evolutionary Computation Conference (GECCO 2024)*, pp. 7 pages, 3 figures, 4 tables, 2024, arXiv preprint arXiv:2403.11159. [Online]. Available: https://doi.org/10.48550/arXiv.2403.11159

[16] X.-A. Dou, Q. Yang, X.-D. Gao, Z.-Y. Lu, and J. Zhang, "A comparative study on crossover operators of genetic algorithm for traveling salesman problem," in *2023 15th International Conference on Advanced Computational Intelligence (ICACI)*, 2023, pp. 1–8.