Large Language Model based Code Completion is an Effective Genetic Improvement Mutation

Jingyuan Wang
Department of Computer Science
University College London
London, United Kingdom
andydiantu233@outlook.com

Carol Hanna
Department of Computer Science
University College London
London, United Kingdom
c.hanna@cs.ucl.ac.uk

Justyna Petke
Department of Computer Science
University College London
London, United Kingdom
j.petke@ucl.ac.uk

***Abstract*—In this work, we introduce a novel large language model (LLM)-based masking mutation operator for Genetic Improvement (GI), which leverages code completion capabilities of large language models to replace masked code segments with contextually relevant modifications. Our approach was tested on five open-source Java projects, where we compared its effectiveness against both traditional GI mutations and an existing LLM-based replacement mutation operator using random and local search algorithms. Results show that the masking mutation operator creates a search space with more compiling and test-passing patches, reducing model response time by up to 60.7% compared to the replacement mutation. Additionally, it outperforms the replacement mutation in achieving the highest runtime improvement on four out of five projects and discovers more runtime-improving patches across all projects. However, combining the masking mutation with traditional GI mutations yielded inconsistent results, suggesting further investigation is needed. This study highlights the promise of LLM-based code completion to boost the efficiency and effectiveness of GI for automated software optimisation.**

***Index Terms*—Large language Models, Genetic Improvement**

I. Introduction

The growing size and complexity of modern software systems have increased the manual effort required for maintenance and optimisation, highlighting the need for automation [1]. Genetic Improvement (GI) has emerged as a promising solution, optimising both functional and non-functional properties such as runtime performance [2]. However, traditional mutation operators in GI are limited by a small search space, yielding a low rate of test-passing patches [3].

Recent advances in large language models (LLMs) have shown a strong synergy between software engineering tasks and LLM capabilities, as well as the potential to enhance GI, though the latter remains underexplored [4]. Brownlee et al. proposed an LLM-based replacement mutation operator that utilises LLMs to generate mutations at the function level. Specifically, the operator provides the LLM with the entire function code as part of the prompt, instructing it to produce alternative implementations of the given function. During benchmarking on five real-world projects, the LLM-based mutation produced more compiling and test-passing patches, resulting in significant fitness improvements [5].

Building on this, we propose a new mutation operator based on LLM code completion, termed the *masking mutation operator*. This approach leverages the LLM's code completion capabilities, offering semantic coherence, reduced risk of infeasible code, and lower computational overhead [6], [7]. To evaluate its effectiveness, we applied this approach to five open-source Java projects, comparing its impact on runtime improvement against both traditional GI mutations and the LLM-based replacement mutation.

To understand the validity and efficiency of our masking mutation operator, we applied our approach to five open-source projects using four different LLMs. In particular:

(1) Random sampling experiments showed that the masking mutation operator generated 43.7% more test-passing patches than the replacement mutation operator and outperformed both traditional GI mutations and the replacement mutation in producing unique, valid, compiling, and test-passing patches, indicating that the masking mutation creates a search space dense with test-passing patches.

(2) GI searches demonstrated that the masking mutation achieved an average of 32.6% higher runtime improvement across four of five projects compared to replacement mutation using the same model, highlighting the masking mutation's ability to find patches with significant runtime improvement.

(3) The masking mutation reduced model response time by 56.7% during random search and by 60.7% when using the Mistral:7B model in local search compared to the replacement mutation, showcasing the masking mutation's efficiency.

(4) Allowing GI to alternate between traditional and masking mutations decreased the number of valid patches and did not consistently yield runtime improvements, indicating further research is needed.

(5) A qualitative analysis revealed common reasons for invalid patches, including incomplete code, unexpected formats, or lack of runtime improvement.

Our results demonstrate the viability of the LLM-based masking mutation operator for the GI process and provide valuable insights for future research in this area. All scripts and code used in our experiments are available at https://github.com/SOLAR-group/gin-Masking-llm.

II. Background

In this section, we discuss the technical background and concepts which this paper is based on.

A. Large Language Models

LLMs are neural networks trained on vast text data for tasks like text generation and question answering [8]. Their core architecture, the transformer, uses self-attention to process and contextualise words [9]. LLMs can be fine-tuned after pretraining on diverse datasets for specific tasks such as classification and summarisation [10], [11].

LLMs have shown great promise in software engineering due to their inherent language-based nature, making them well-suited for tasks that require contextual understanding and generation of text [12]. They have been employed in various software engineering applications, including code generation [13], program repair [7], and mutation testing [6], achieving competitive or even state-of-the-art performance on several software engineering tasks.

A key technique driving the success of LLMs in these applications is masking. This technique selectively hides certain input tokens to encourage the model to learn the context and relationships between tokens. Masking is widely used in training [14], fine-tuning [15], and downstream tasks [7] phases of LLM development. It facilitates efficient learning of contextual understanding, enhances generalisation capabilities, and reduces data and computational requirements [15].

B. Genetic Improvement

GI is an automated process that uses search-based techniques (SBSE) to enhance both functional and non-functional aspects of existing software, such as bug fixing and reducing energy consumption [16], [17]. GI has demonstrated capabilities in enhancing large and complex software systems in academia and industry [16], [18]. GI allows for software to automatically self-optimise to the environments it is run in, with the user specifying the optimisation objective.

A notable tool in this field is Gin, developed by Brownlee et al., which serves as a flexible and expandable toolkit for conducting GI experiments [17]. Gin enhances software by introducing a series of minor mutations to the original code, generating a collection of modified patches. These modified patches are then applied to the code, which is then compiled and tested to ensure it improves the software's property of interest. The patch-finding process relies on heuristic approaches like local search, supported by three main functionalities:

Profiling: Gin includes an integrated profiler that identifies 'hot methods' – those that frequently appear at the top of the execution stack during unit test runs, indicating significant resource consumption.

Search: To evaluate the quality of the mutation operator's search space and its ability to find fitness improvements, we utilised two search algorithms provided by Gin: local and random search. Random search samples from the full spectrum of possible edits, while local search accepts new edits only if they enhance runtime performance while passing all unit tests.

Applying Mutations: Gin parses programs into an Abstract Syntax Tree (AST) using JavaParser. Mutations are directly applied to nodes within the AST, allowing precise modifications to the software's code structure. Typically, these mutations are performed at the statement level, involving deletion, replacement, or insertion of statements from other locations in the code. Brownlee et al. [5] recently added an LLM-based replacement mutation operator, which generates replacement code for a given Java method.

III. Approach

An effective mutation operator for GI should provide a search space that is both rich and dense in valid program variants, meaning it can quickly identify patches that offer significant improvements of the target property [17]. To explore the potential of LLMs as mutation operators, we propose the masking mutation operator, based on LLM-based code completion.

A. The Masking Mutation Operator

We propose a new mutation type using LLM-based code completion. Instead of traditional mutations or generating entire methods, we mask a selected statement and prompt the LLM to predict it based on context. This approach has shown promise in mutation testing and program repair [6], [7].

In our experiments, a statement that satisfied our selection mechanism is selected uniformly at random in the hot method identified by Gin's profiling tool. The chosen statement is then masked using the <PLACEHOLDER>sign, and the masked code is passed to the LLM with a prompt (Fig. 1) instructing it to replace the placeholder with a meaningful implementation.

Statement Selection Mechanism: To enhance the masking mutation's efficiency, we implemented a statement selection mechanism targeting statement types more likely to generate compilable, test-passing patches.

Mini-experiments (detailed in Section V) identified specific statement types with a higher probability of producing beneficial patches. This mechanism is designed exclusively for masking mutations, as it operates at the statement level and cannot be applied to function-level mutations. By focusing on specific code segments, it aims to optimise the GI process by focusing on code segments most likely to yield patches that compile, pass tests, and improve non-functional objectives.

B. Combining the Masking Mutation with Traditional GI

Brownlee et al. [5], [19] demonstrated that LLM-based mutations are more likely to generate patches that compile and pass tests, while traditional GI mutations produce more diverse patches. To leverage both strengths, we explore the efficiency and efficacy of combining these approaches by alternating between traditional GI and masking mutations in the GI process. To combine different mutation operators, we extend Gin's `neighbour` selection algorithm.

In Gin's local search algorithm, the `neighbour` selection algorithm generates a patch by selecting and applying a random traditional GI mutation (which either deletes, replaces, or inserts a statement extracted from existing codebase). We extended this by adding the masking mutation as a selectable option. To fully leverage the benefits of the masking mutation, we assigned it a higher selection probability compared to an equal random distribution across all mutation operators.

IV. Research Questions

To evaluate our approach, we pose the following questions:

RQ1: Does the masking mutation create a denser search space with more unique, compiling, and test-passing patches compared to the replacement mutation?

This question assesses the masking mutation's capacity to explore a broader solution space, providing a richer set of unique and test-passing patches.

RQ2: Can the masking mutation identify patches that yield greater runtime improvements compared to the replacement mutation when using the same LLMs on identical datasets?

This question focuses on the effectiveness of the masking mutation, evaluating its ability to discover patches that lead to runtime enhancements. By comparing the runtime performance of patches generated through masking and the replacement mutations, we aim to understand their relative strengths and limitations.

RQ3: What is the runtime efficiency of the masking mutation compared to the replacement mutation?

This question evaluates the runtime efficiency of the masking mutation. Combined with the first two research questions, this assessment provides a comprehensive understanding of the overall performance and practicality of the masking mutation.

If the masking mutation operator proves to be both efficient and effective, we intend to explore its potential in combination with traditional GI mutations. To investigate this, we pose additional research questions:

RQ4: How does the combination of masking and traditional GI mutations compare to using either method alone?

Understanding the efficiency of the GI process in real-world applications is crucial. Given the computational demands of LLMs, we seek to determine if smaller LLMs can sufficiently perform the required tasks. Thus, we ask:

RQ5: How does the capability of the LLM used in the masking mutation affect the effectiveness of the GI process, specifically in finding test-passing patches, runtime improvements, and overall response time?

V. Experimental Setup

We conducted a series of experiments to address our research questions, described as follows.

Experiment 1: To address RQ1, we compared the search spaces generated by the replacement and masking mutations. We ran the random sampling algorithm for each mutation operator and compared the number of valid, compiling, and test-passing patches. Additionally, we analysed a subset of impactful statement types for the masking mutation.

Mini Experiment 1.1: To improve the efficiency of the masking mutation, we conducted a mini-experiment to identify impactful statement types. We applied the random sampling algorithm 40 times for each statement type, generating 40 patches per type. All statement types present in the projects and supported by JavaParser 3.24.0, including Block, Expression, For, If, Return, Throw, While, Break, Switch, and Continue, were analysed. We then compared the number of valid, compiling, and test-passing patches generated for each type. Statement types yielding the highest numbers of test-passing patches were identified as impactful and will guide our statement selection mechanism in the masking mutation process.

Experiment 2: To address RQ2, this experiment assesses the ability of both the replacement and masking mutations to produce runtime-improving patches. We run GI with the local search algorithm for each mutation operator, comparing the number of test-passing, runtime-improving patches and the runtime improvement magnitudes.

Experiment 3: To answer RQ3, we assessed the runtime efficiency of the masking and replacement mutations by collecting and comparing model response times from GI experiments using both local search and random sampling.

Experiment 4: Focusing on RQ4, we examined the effectiveness of combining masking mutation with traditional GI mutations by running local search and random sampling for the combined approach, comparing the results to those from masking and traditional mutations used independently.

Experiment 5: To address RQ5, we repeated Experiments 1, 2, and 3 for the masking mutation using LLMs with sizes ranging from 2B to 9B parameters to evaluate its scalability and performance across different model sizes.

A. Target Codebase & Profiling

Following prior studies, we selected five codebases for GI experiments: Jcodec, JUnit4, Gson, Commons-net, and Karate [5]. These projects were chosen for their popularity, open-source availability, compatibility with the Gin framework, and use of Java 17 with Maven or Gradle.

As described in Section II-B, Gin's profiler samples the execution stack during unit tests, which can yield variable results. To reduce this variability, we profiled each project 20 times and aggregated the results. The 10 most frequently identified hot methods were then chosen as GI targets.

B. Algorithms' Parameters

To ensure comparability with prior work, we used the same parameter settings as Brownlee et al. [5] for search algorithms.

Random Sampling: Each experiment run is configured to generate 1,000 patches, with a timeout of 10,000 milliseconds to prevent infinite loops. If the time limit is exceeded, the patch is considered a test failure.

Local Search: Each experiment run conducts 100 evaluations per hot method identified, with each experiment totalling 1,000 evaluations. The initial evaluation assessed baseline performance, followed by 99 patches for each method. Default parameters were used unless specified otherwise.

Combined Search: For each combined search experiment, we conducted both local search and random sampling, updating the mutation-selection mechanism from equal-probability random selection to the approach outlined in Listing **??**. The probability of selecting the masking mutation over traditional GI mutations was set to 30%, 50%, and 70% in different experimental runs. All other algorithm parameters were kept consistent with those specified in Section V-B.

TABLE I
NAME, ID, QUANTISATION, NUMBER OF PARAMETERS AND SIZE OF LLMS UTILISED IN EXPERIMENTS.

Name	ID	Quantisation	Parameters	Size
Gemma2:2B	430ed3535049	Q4_0	2.61B	1.7 GB
Gemma2:9B	ff02c3702f32	Q4_0	9.24B	5.4 GB
Llama3.1:8B	91ab477bec9d	Q4_0	8.03B	4.7 GB
Mistral:7B	61e88e884507	Q4_0	7.24B	4.1 GB

C. LLM Selection

We chose open-source, locally hosted LLMs for two reasons: first, studies show that smaller local models can perform comparably to commercial models. For example, Brownlee et al. [5] found Mistral:7B outperformed GPT-3.5 in GI on three of five projects. Second, local models improve reproducibility by maintaining consistent versions, unlike commercial models that update frequently without legacy access.

For our experiments, we focused on models spanning a range of 2B to 9B parameters to evaluate the impact of model scale on mutation performance. Specifically, we selected Llama3.1:8B [20] and Gemma2:9B [21] due to their state-of-the-art performance and widespread adoption. Based on the strong results of Gemma2:9B, we included Gemma2:2B [21] to assess the lower end of the parameter range. Additionally, Mistral:7B [22] was chosen as a top performer in previous work [5]. These models were selected for their open-source availability, compatibility with our experimental setup, and strong performance compared to other models with similar parameter counts [20], [21]. Table I summarizes the specifications of the selected models.

To host these LLMs locally, we used the Ollama framework, chosen for its compatibility with many LLMs, ease of use, and straightforward setup. This framework ensures that our experiments can be easily replicated and verified.

D. LLM Prompt Templates

This section outlines the prompt templates used to guide LLMs in the masking mutation. The template dynamically includes function-specific details, such as masked code segments, to enable precise LLM responses. The masking mutation template, shown in Fig. 1, replaces placeholders with specific masked function code and project names, leveraging the LLM's ability to generate relevant code completions or improvements.

We experimented with several prompt variations. Through testing different approaches, we identified the prompt that consistently produced the most useful outputs. During this process, we adhered to prompt engineering best practices, including clear and concise instructions, ensuring relevant context, and structuring the prompt to guide the LLM toward generating effective responses [23], [24]. This iterative process maximised the LLMs' ability to generate valid and performance-enhancing patches during the GI experiments.

E. Implementation Details

Our experiments were conducted using the Gin framework [17], with extensions based on the LLM branch commit **`f2f6e10`** from the Gin GitHub repository. To contribute to Gin's development, we submitted a pull request with these extensions. All experiments were performed on an Ubuntu 22.04 LTS system with an NVIDIA RTX 4060 Laptop GPU (8GB), Intel Core i7-13650HX, and 16GB RAM, running Java 17, Maven 3.6.3, and Gradle 8.0.2.

Please replace <<PLACEHOLDER>> sign in the method below with meaningfull implementation.
<<Masked Code>>
*"This code belongs to project **<<Project Name>>**. Wrap all code in curly braces, if it is not already. Do not include any method or class declarations. Label all code as java.*

Fig. 1. Prompt template used for the masking mutation operator.

For LLM integration, we used Ollama 0.1.48 to host local LLMs and Ollama4j 1.0.44 to develop extensions for Gin, facilitating communication with the local Ollama server.

All LLM hyperparameters, including temperature, top-p, and others, were set to the default values provided by the Ollama framework to maintain consistency across experiments.

VI. EVALUATION

In this section, we conduct both quantitative and qualitative analyses of the results produced by the GI process with random and local searches using the masking mutation operator, to evaluate the operator's efficacy and efficiency.

1) Random Sampling

We analysed the random sampling results for the GI process using the masking mutation across different LLMs, focusing on the total and unique counts of valid, compiled, and test-passing patches. These results were compared to those generated by traditional GI mutations and replacement mutations, with all experiments conducted in the same environment to ensure consistency. The results are shown in Fig. 2.

The Gemma2:9B model achieved the highest counts of valid, compiled, and test-passing patches across all projects, averaging 908 valid, 542 compiled, and 516 test-passing patches per 1,000 generated. This marks a significant improvement over prior studies, where local LLMs underperformed in valid patch generation, with the masking mutation surpassing traditional GI across all five projects.

However, the increase in unique patches was less pronounced. While most models produced similar results to previous studies, Gemma2:2B generated the most unique patches in four projects, and Llama3.1:8B led in unique valid patches, despite generating fewer overall valid and test-passing patches.

To isolate the effects of masking mutation, we compared masking and replacement mutations using Mistral:7B. On average, masking generated 8.6% more valid patches, 11.87% more test-passing patches, but 6.45% fewer unique valid patches. However, it produced 8.9% more unique test-passing patches than replacement mutation.

Compared to recreated replacement mutation results, the masking mutation consistently outperformed in generating valid, compiled, and test-passing patches across all projects, even with Mistral:7B.

In summary, the GI process with the masking mutation, particularly with Gemma2:9B, yielded significant gains in valid, compiling, and test-passing patches, outperforming both traditional GI mutations and the replacement mutation. Al-

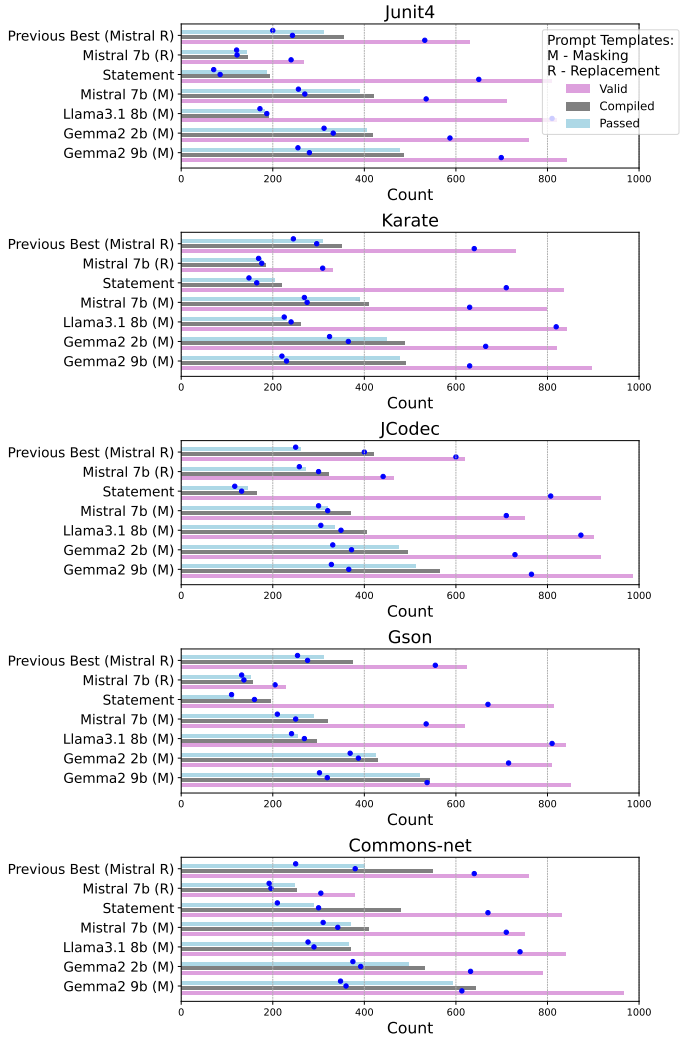

Fig. 2. Random sampling results for each LLM and project combination, illustrating the counts of valid, compiled, and test-passing patches. Blue dots indicate unique patches. (M) denotes the masking mutation, (R) represents the replacement mutation, and Statement refers to traditional GI mutations. "Previous Best (Mistral R)" reflects the results of Mistral:7B from Brownlee et al.'s work [5], while "Mistral:7B (R)" represents the replacement mutation results for Mistral:7B, as recreated in our experimental environment.

though unique patch counts were somewhat lower, models like Gemma2:2B and Llama3.1:8B excelled in this area. Thus:

Answer to RQ1: The GI process using the LLM-based masking mutation operator provides a denser search space with compiling and test-passing patches compared to the LLM-based replacement mutation operator, although it produces slightly fewer unique valid patches.

2) Local Search

We analysed the local search results for the GI process using the masking mutation across each LLM, focusing on maximum and median runtime improvements achieved by fitness-improving patches. These results were compared to those produced by GI processes using traditional GI mutations and the replacement mutation. The results for each project are shown in Fig. 3.

For both maximum and median fitness improvements, the GI process using the Gemma2:9B model with the masking mutation yielded the best results across all projects. Notably, despite its smaller size, the Gemma2:2B model achieved the second-best maximum improvements in 3 out of 5 projects.

To isolate the impact of the masking mutation, we compared results between masking and replacement mutations using Mistral:7B. The masking mutation consistently outperformed the replacement mutation in maximum and median improvements across all projects except JUnit. Additionally, when compared to traditional GI mutations, the masking mutation consistently delivered higher maximum and median fitness improvements across all models and projects.

We also analysed the number of runtime-improving patches generated during the local search experiments (Table II). The Gemma2:9B model with the masking mutation consistently produced the most runtime-improving patches, averaging 122 per run —- significantly higher than traditional GI mutations (38.2). Comparing the Mistral model's performance, the masking mutation consistently outperformed the replacement mutation across all projects, highlighting the masking mutation's advantage in identifying performance-improving patches.

Answer to RQ2: The LLM-based masking mutation outperformed the LLM-based replacement mutation in 4 out of 5 projects. Additionally, the masking mutation generated more performance-improving patches and consistently outperformed traditional GI mutations across all experiments.

3) Model Response Time

To evaluate GI process efficiency across mutation operators, we analysed the average model response time for the GI process of each mutation and model combination. The results are shown in Fig. 4. Comparing local search to random sampling, we found that local search generally had a higher average response time along with greater standard deviation, largely due to shorter execution times in the Gson and Commons-net projects, which identified fewer than 10 hot methods.

When comparing the masking and replacement mutations, the masking mutation consistently showed lower average response times across all models, in both local and random sampling experiments. With the Mistral:7B model, the masking mutation achieved a 56.7% reduction in response time in random sampling and a 60.7% reduction in local search compared to the replacement mutation. Gemma2:2B exhibited the lowest average response times overall, with response times 62.8% lower than replacement mutations in random sampling and 64.8% lower in local search.

These findings indicate that masking mutation, especially with the Gemma2:2B model, significantly reduces model response time, suggesting potential benefits for power efficiency.

Answer to RQ3: The GI process using the LLM-based masking mutation operator offers competitive performance while significantly reducing model response time compared to the replacement mutation.

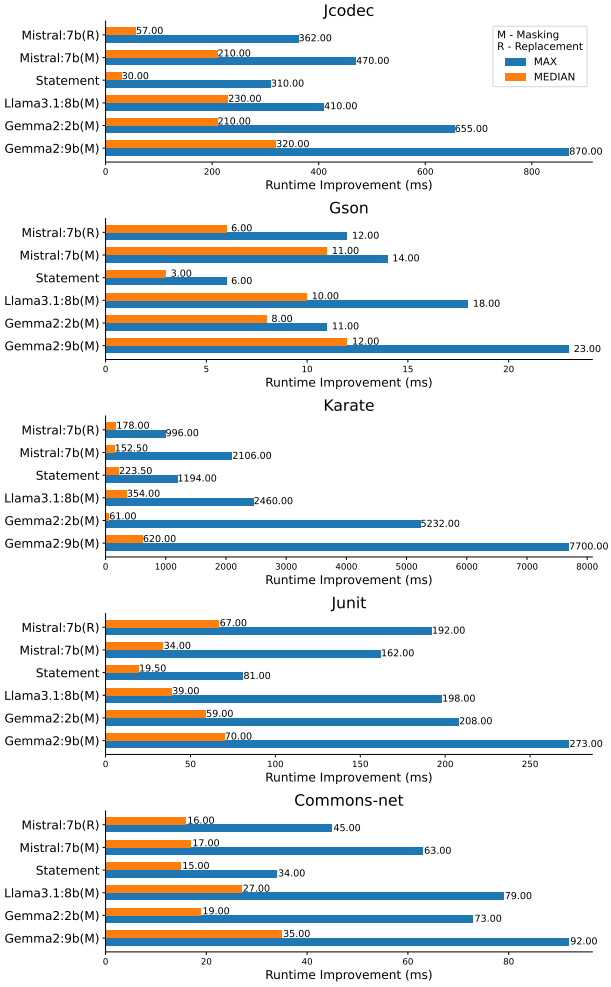

Fig. 3. Local search results for each LLM and project combination, showing the maximum and median runtime improvements. (M) denotes masking mutation results, and (R) denotes replacement mutation results. Statement represents the traditional GI mutations.

TABLE II
NO. OF RUNTIME IMPROVING PATCHES VIA GI USING LOCAL SEARCH.

	JCodec	Gson	Commons-net	Karate	Junit
Previous Best Mistral:7B (R)	28	47	24	83	74
Statement	18	17	34	65	57
Mistral:7B(R)	74	3	25	29	25
Mistral:7B(M)	59	59	27	124	98
Llama3.1:8B(M)	45	35	19	98	37
Gemma2:2B(M)	98	27	52	132	78
Gemma2:9B(M)	**132**	**66**	**115**	**152**	**145**

4) Combined Search

We evaluated the efficiency of a combined search approach that alternates between traditional GI mutations and the masking mutation, using the Mistral:7B model in both local search and random sampling. Probabilities of selecting the masking mutation (30%, 50%, 70%) were tested, with results compared to using masking mutation alone.

Random Sampling: As shown in Fig. 5, in the random sampling scenario, decreasing the masking mutation probability led to an increase in both total and unique valid patches. For example, a 30% masking probability produced 19.8% more valid patches and 21.5% more unique valid patches than using the masking mutation alone, although it generated 7.2% fewer compiling and 6.9% fewer test-passing patches.

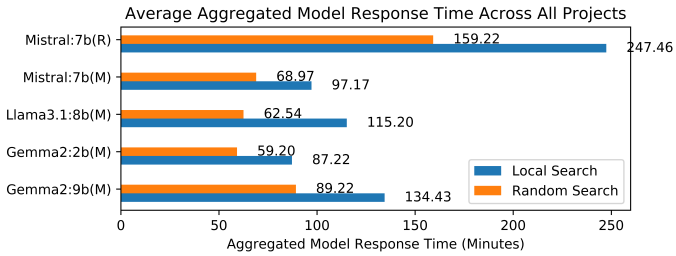

Fig. 4. Avg. model response time across all projects for each model and mutation combination. (M) denotes masking, and (R) replacement mutation.

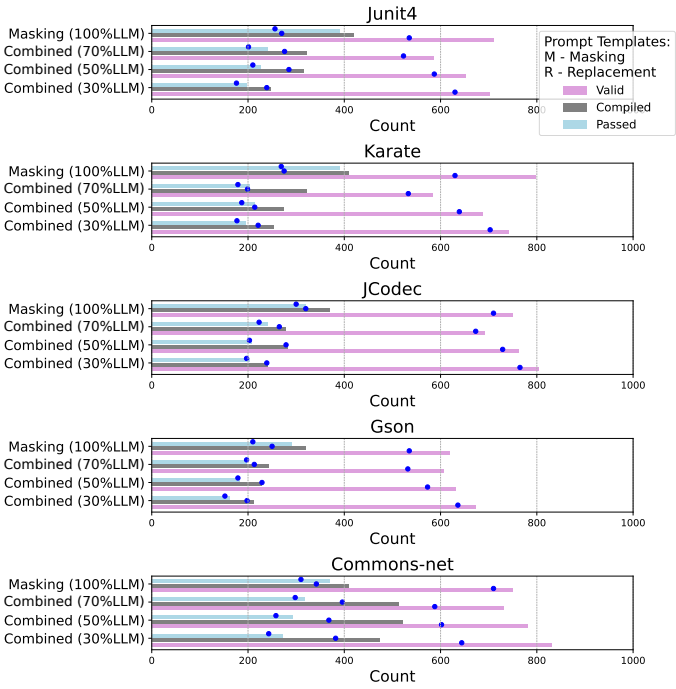

Fig. 5. Random sampling results for the combined search using the masking mutation and Mistral:7B model, showing the number of valid, compiled, and test-passing patches. The blue dot indicates the number of unique patches.

Local Search: Local search results, shown in Fig. 6, indicate that the combined approach outperformed the masking mutation alone only in the Gson project, achieving a 5% higher maximum runtime improvement. For the other projects, no combined configuration surpassed the runtime improvements achieved by the masking mutation alone.

Among combined configurations (30%, 50%, 70%), no consistent trend emerged in identifying the best patches. The 30% masking probability performed best in two projects, 70% in two others, and 50% in the remainder, with only marginal differences in runtime improvements across configurations.

Overall, the combined search approach delivered mixed results. While it enhanced the number of valid and unique patches in random sampling, it was inconsistent in local search and did not consistently outperform the masking mutation alone or traditional GI mutations. Further research is needed.

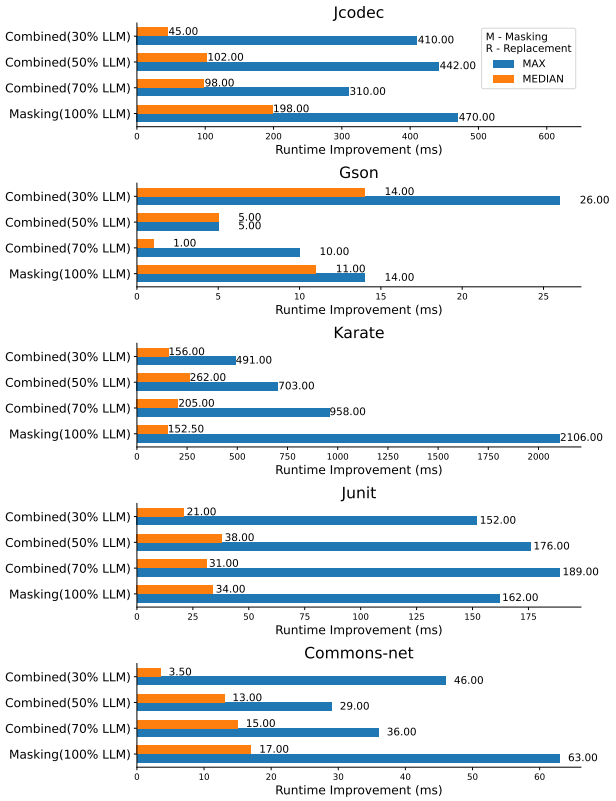

Fig. 6. Local search results for the combined search using the masking mutation and Mistral:7B model, displaying max and median runtime improvements.

Answer to RQ4: Alternating between traditional GI mutations and LLM-based masking mutations generates more valid patches than masking alone, but inconsistent runtime improvements suggest further investigation is needed to maximise this approach's effectiveness.

A. *Qualitative Analysis*

While the masking mutation provides a denser search space than traditional GI and replacement mutations, many patches remain invalid, fail to compile, or do not pass tests. To address these issues and guide future improvements, we conducted a qualitative analysis of LLM responses and Gin's parsing outputs, categorising issues into three main types:

Category 1: Incomplete Code Returned: LLMs sometimes returned partial or incomplete code instead of a complete solution, an issue observed across all LLM and project combinations. Responses occasionally included only instructions or examples instead of executable code. For instance, models from the Gemma family (e.g., Gemma2:2B, Gemma2:9B) sometimes returned only function signatures with explanatory text and no implementation.

Category 2: Code Not in Expected Format: We implemented a code extraction mechanism to detect and extract code blocks, marked by ```<CODE>```. Although this worked in 80% of cases, some responses deviated from the expected format. Models like Llama3.1:8B and Gemma2:9B sometimes included unrelated helper functions or returned entire classes rather than just methods, making extraction challenging. Due to the statistical nature of LLM responses, defining extraction rules that accommodate all variations remains difficult. This highlights an opportunity for future research to develop more sophisticated extraction methods.

Category 3: Meaningful Code Without Improvement: The most frequent failure type involved LLMs generating meaningful code that, while executable, failed to pass tests or improve performance. Specific subcases included (1) returning the input code unchanged, (2) replacing the placeholder with comments, and (3) producing repetitive outputs with no new variation across runs.

VII. Threats to Validity

Our experiments used four locally hosted LLMs to evaluate the masking mutation's validity and efficiency. A primary threat to validity is the inherent black-box nature of LLMs, as updates to models can lead to variability in results. To address this, we specified LLM versions and hosting tools.

A common challenge in search-based software engineering, including this study, is accurately measuring runtime improvements. Additionally, we relied on the validity of the provided test suites. To address these concerns, we followed established GI guidelines for all measurements and used benchmarks from prior studies to mitigate inadequate testing risks [5].

Prompts used in this study, while informed by best practices, were not developed through a strictly scientific method, which may limit performance if prompts do not fully leverage the LLMs' capabilities. This challenge is inherent in LLM research, and we adhered to literature-backed prompt engineering guidelines to mitigate potential issues.

The non-deterministic behaviour of LLMs also poses a risk to validity. Our experiments required approximately 225 computational hours. Despite these limitations, our findings align with prior conclusions regarding replacement and traditional GI mutation operators established by Brownlee et al.

VIII. Related Work

In this section, we discuss related work in the areas of LLM-enhanced GI mutation and LLM-based code completion, focusing on the limitations and breakthroughs in related work.

GI enables software self-optimisation based on user-defined objectives, effectively optimising functional and non-functional properties in large software systems [16]. However, traditional GI is limited by a narrow set of mutations [25].

Recent work has used LLMs to expand mutation scope in GI. Kang and Yoo [4] employed code-davinci-002 to improve time and memory efficiency in functions, though some LLM outputs were incorrect. In this work, we consider more fine-grained level changes. Brownlee et al. [19] integrated LLM-based mutations into the GI toolkit Gin and introduced the replacement mutation operator. This operator employs LLMs to generate mutations at the function level by providing the entire function code as input and prompting the model to produce alternative implementations. Benchmarking this approach on real-world projects revealed that, while traditional mutations generated more diverse patches, LLM-based mutations consistently produced more test-passing patches. Extending this

work, Brownlee et al. [5] tested the replacement mutation operator on three LLMs across five projects, with results showing LLM-based mutations outperformed traditional methods, underscoring the potential of LLMs in enhancing GI. Brownlee et al. do not employ masking. This could prompt LLM to generate similar (inefficient) solutions. Moreover, we restrict statement level types to boost probability of test-passing variants. Our results show that masking strategy outperforms the previous approach.

Unlike code generation, which creates code from scratch, LLM-based code completion fills missing segments, providing flexible, fine-grained integration with traditional GI mutations. This approach leverages LLMs' pattern-matching capabilities and has been widely adopted in code completion tools [26].

In related fields, LLM-based code completion has shown promising results in mutation testing and program repair. Ribeiro et al. [7] framed automated program repair as a completion task, using CodeGPT to repair buggy lines with a 27% repair rate on the ManySStuBs4J dataset [27].

IX. Conclusions

We introduced a masking mutation operator, leveraging LLM-powered code completion, to improve the GI process. Evaluated on five open-source projects with four locally hosted LLMs, the operator was benchmarked against traditional GI mutations and Brownlee et al.'s replacement mutation [5]. Additionally, a mini-experiment identified key statement types to enhance patch pass rates, and we tested the combined effect of masking and traditional GI mutations.

The masking mutation demonstrated strong performance, generating up to 38.2% more test-passing patches than the replacement mutation and producing a diverse set of valid and compiled patches. It consistently outperformed the replacement mutation in runtime improvements across four of five projects, with an average gain of 32.6%. The Gemma2:9B model showed the highest performance, while the smaller Gemma2:2B model also ranked second in three projects.

Efficiency gains were evident, with the masking mutation reducing response times by 56.7% during random sampling and 60.7% in local search. Combining masking with traditional GI mutations increased valid patch counts but yielded fewer compiling and test-passing patches, with inconsistent runtime improvements, suggesting further refinement is needed.

In summary, the masking mutation operator provides a richer search space of test-passing patches, greater runtime improvements, and significant efficiency gains over traditional GI and LLM-based replacement operators, making it a promising tool for future GI research.

References

[1] M. Benaroch and K. Lyytinen, "How much does software complexity matter for maintenance productivity? the link between team instability and diversity," *IEEE TSE*, vol. 49, no. 4, pp. 2459–2475, 2023.

[2] F. Sarro, "Search-based software engineering in the era of modern software systems," in *Proc. of RE*, vol. 2023. IEEE, 2023, pp. 3–5.

[3] J. Petke, B. Alexander, E. T. Barr, A. E. Brownlee, M. Wagner, and D. R. White, "Program transformation landscapes for automated program modification using gin," *EMSE*, vol. 28, no. 4, p. 104, 2023.

[4] S. Kang and S. Yoo, "Towards objective-tailored genetic improvement through large language models," in *GI@ICSE*. IEEE, 2023, pp. 19–20.

[5] A. E. I. Brownlee, J. Callan, K. Even-Mendoza, A. Geiger, C. Hanna, J. Petke, F. Sarro, and D. Sobania, "Large language model based mutations in genetic improvement," *ASE*, 2024.

[6] A. Khanfir, R. Degiovanni, M. Papadakis, and Y. L. Traon, "Efficient mutation testing via pre-trained language models," *arXiv:2301.03543*, 2023.

[7] F. Ribeiro, R. Abreu, and J. Saraiva, "Framing program repair as code completion," in *APR@ICSE*, 2022, pp. 38–45.

[8] E. Kasneci, K. Seßler, S. Küchemann, M. Bannert, D. Dementieva, F. Fischer, U. Gasser, G. Groh, S. Günnemann, E. Hüllermeier *et al.*, "ChatGPT for good? On opportunities and challenges of llms for education," *Learning & individual differences*, vol. 103, 2023.

[9] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[10] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong *et al.*, "A survey of large language models," *arXiv:2303.18223*, 2023.

[11] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.

[12] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," *arXiv:2310.03533*, 2023.

[13] G. Antal, R. Vozár, and R. Ferenc, "Assessing gpt-4-vision's capabilities in uml-based code generation," *arXiv:2404.14370*, 2024.

[14] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *MLR*, vol. 21.

[15] C. Chen, X. Wang, T.-E. Lin, A. Lv, Y. Wu, X. Gao, J.-R. Wen, R. Yan, and Y. Li, "Masked thought: Simply masking partial reasoning steps can improve mathematical reasoning learning of language models," in *Proc. of the 62nd Meet. of the Assoc. for Comp. Linguistics*, L.-W. Ku, A. Martins, and V. Srikumar, Eds. Assoc. for Comp. Linguistics, 2024.

[16] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward, "Genetic improvement of software: a comprehensive survey," *TEVC*, vol. 22, no. 3, pp. 415–432, 2017.

[17] A. E. I. Brownlee, J. Petke, B. Alexander, E. T. Barr, M. Wagner, and D. R. White, "Gin: genetic improvement research made easy," in *GECCO*. ACM, 2019, p. 985–993.

[18] S. Kirbas, E. Windels, O. McBello, K. Kells, M. Pagano, R. Szalanski, V. Nowack, E. R. Winter, S. Counsell, D. Bowes, T. Hall, S. Haraldsson, and J. Woodward, "On the introduction of automatic program repair in bloomberg," *IEEE Software*, vol. 38, no. 4, pp. 43–51, 2021.

[19] A. E. Brownlee, J. Callan, K. Even-Mendoza, A. Geiger, C. Hanna, J. Petke, F. Sarro, and D. Sobania, "Enhancing genetic improvement mutations using large language models," in *SBSE*.

[20] A. D. et al., "The llama 3 herd of models," 2024. [Online]. Available: https://arxiv.org/abs/2407.21783

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

[22] M. AI, "Announcing mistral 7b," https://mistral.ai/news/announcing-mistral-7b/, 2024, accessed: 28-Aug-2024.

[23] S. Schulhoff, M. Ilie, N. Balepur, K. Kahadze, A. Liu, C. Si, Y. Li, A. Gupta, H. Han, S. Schulhoff *et al.*, "The prompt report: A systematic survey of prompting techniques," *arXiv:2406.06608*, 2024.

[24] OpenAI, "Openai api documentation: Prompt engineering guide," https://platform.openai.com/docs/guides/prompt-engineering, 2024, accessed: 28-Aug-2024.

[25] J. Petke, B. Alexander, E. T. Barr, A. E. Brownlee, M. Wagner, and D. R. White, "A survey of genetic improvement search spaces," in *GECCO*, 2019, pp. 1715–1721.

[26] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," *SIGPLAN Not.*, vol. 49, no. 6, p. 419–428, 2014.

[27] R.-M. Karampatsis and C. Sutton, "How often do single-statement bugs occur? the manysstubs4j dataset," in *MSR*, 2020, pp. 573–577.