Applying Genetic Improvement Techniques for Automated Program Repair of Transpiled Code

Prasham Jadhwani, Carol Hanna, William B. Langdon, Justyna Petke
University College London
Gower Street, London WC1E 6BT, UK
{prasham.jadhwani.21,carol.hanna.21,w.langdon,j.petke}@ucl.ac.uk

Abstract

We use Genetic Improvement (GI)-based Automated Program Repair (APR) techniques for syntax correction on transpiled code produced by both large language models (LLMs) and rule-based translators. A three-stage pipeline was developed, combining rule-based test generation, an optional LLM-driven preprocessing stage for syntax correction, and GI-based repair strategies. LLM-assisted Type Change Operator and Boolean Value Change Operator were added to the MAGPIE GI framework, which reduced transpilation bugs from Python to Java by **33%** (LLM) and by **18%** on rule-based translations. A comprehensive taxonomy of common transpilation bugs was developed, mapping faults to mutation operators, alongside an evaluation of the effectiveness of secondary LLM interventions.

CCS Concepts

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

Keywords

Software Maintenance, Code Transpilation

ACM Reference Format:
Prasham Jadhwani, Carol Hanna, William B. Langdon, Justyna Petke. 2026. Applying Genetic Improvement Techniques for Automated Program Repair of Transpiled Code. 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/10.1145/3786162.3793234

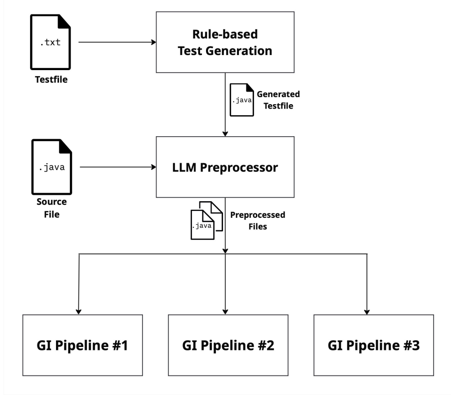

Figure 1: Three Stage Approach for Translation Repair

1 Introduction

Source-to-source transpilers, also known as source-to-source transcompilers/translators, transform source code from one programming language to another while preserving the program's logic and functionality. These tools are used for performance optimisation, legacy system migration, and cost enhancement [5] [6]. However manual source-to-source translation can be highly resource-intensive. For instance, the Commonwealth Bank of Australia spent ≈$750 million over five years to switch from COBOL to Java [11].

Initial techniques for source-to-source translation involved manually specifying translation rules for the translation of Ada to Pascal and Pascal to Ada [3]. Nyugen et al. [14] used lexical phrase based statistical machine translation to perform method-to-method migration from Java to C#. Despite achieving high lexical translation accuracy, many translations were syntactically incorrect and had parsing errors [1, 14, 21]. More recently Transcoder [20] used sequence-to-sequence (seq2seq) Large Language Models (LLMs) and achieved over 80% functional translation accuracy across Java, C++ and Python. As of today, three primary techniques are used for source-to-source code translation: rule-based, machine translation-based and deep learning-based methods. Despite significant advancements, source-to-source translation has yet to achieve fully automated and functionally reliable results, particularly due to residual errors that persist even after applying state-of-the-art translation models. Pan et al. [17] have highlighted this gap in translation quality and the need for alternative approaches to complement and enhance the capabilities of LLMs in source-to-source translation.

Unlike LLMs, deep learning and rule-based techniques, genetic improvement (GI) can automatically adapt and evolve program fixes [12], making it particularly well-suited for correcting diverse and unpredictable errors generated by LLMs. GI-based APR applies evolutionary search techniques to automatically detect and correct errors in programs by evolving program variants toward improved correctness and functionality. We integrate GI-based APR into the source-to-source translation pipeline to enable automated post-translation correction and so reducing the need for manual intervention.

2 GI-based Repair of Transpiled Code

Figure 1 shows our design for producing fixes for translation bugs. We start with *Testfile* and apply a rule-based solution, which gives a possibly buggy Java translation. Optionally this is processed by an LLM, and then one of three the GI Pipelines. All details are presented in Section 3.

To evaluate our approach, we pose these research questions:

RQ1: What are the common categories of bugs in transpiled code, and how can they be effectively mapped to specific mutation operators within GI tools?

RQ2: To what extent can secondary LLM interventions enhance the quality of buggy transpiled code to enable successful application of GI tools?

RQ3: How effective are existing standard GI tools and LLM-augmented GI tools in repairing LLM-translated code?

RQ4: How do traditional mutation operators compare to custom mutation operators in their ability to produce correct patches for transpilation-induced bugs?

RQ5: How does the performance of our approach differ when applied to rule-based transpiled code compared to LLM-based transpiled code?

3 Methodology

The following section outlines the methodology used to investigate the effectiveness of GI tools in addressing translation bugs. We describe the implementation of three key stages of our approach: Rule-based Test Generation, the LLM preprocessor, and the GI pipelines.

3.1 Rule-based Test Generation

For the Rule-based Test Generation functionality we considered existing off-the-shelf rule-based translators like p2j and Py2Java. However, these translators lacked native support for Python test cases, making them unsuitable for our needs. Since the translation of Python test cases was a critical component of this project, it became evident that implementing a custom rule-based test generation script would provide more flexibility and ensure compatibility with the specific format required for `JUnit 4` syntax. The Rule-based Test Generation script was written in Python 3.11, to parse the plaintext test case file and output the test cases in `JUnit 4` syntax.

We briefly outline the steps followed in the implementation of this script. The first step in parsing each individual test case began by extracting the `<assert_statement>` using one of three different regular expressions: `r‘self.assertEqual\((.*)\)’`, `r‘self.assertTrue\((.*)\)’`, `r‘self.assertFalse\((.*)\)’`. These three regular expressions were chosen based on a preliminary analysis of the test case corpus. The second step is parsing the assertion arguments. Once the arguments within the `<assert_statement>` were captured, the next step was to extract the `expected_output` and the function call in the form `<test_function>(**function_args)`. This was accomplished by locating the first comma, which typically served as the delimiter between the two arguments. However, an edge case arose when the `expected_output` itself contained a comma, such as when it was a `list` or `String`. To correctly handle such cases, two flag variables, `inside_quotes` and `bracket_level`, were maintained while parsing the argument string from left to right. The first comma encountered when both flags were set to `false` was reliably identified as the true delimiter separating the `expected_output` from the function call. The third step is processing and typing the arguments. The extracted `<test_function>` was first converted to camelCase to adhere to Java naming conventions. Then, the `expected_output`, initially extracted as a string, was converted to its corresponding datatype using Python's `eval()` function, which dynamically interprets Python expressions from strings. Based on the parsed value, the expected output was mapped to one of the following Java-compatible types: `List`, `int`, `long`, `float`, `double`, `boolean`, or `Object`. The `Object` type served as a fallback, particularly for string values, since all Java datatypes—primitive or reference—inherit from `Object`, ensuring compatibility and preventing type mismatch errors during compilation.

3.2 Optional LLM Preprocessor

The LLM preprocessor selects the files that need to be preprocessed for syntax errors, prompting the GPT-4o-mini LLM with the faulty code and a custom prompt, extracting the code from the LLM's response, and storing it back to the file. It was written in Python 3.11 with OpenAI, dotenv, re, os and sys libraries. A regular expression was used identify and extract the code block from the text returned by the LLM.

We used Structured Prompting [9] with GPT-4o-mini. Prompts were carefully designed using clear delimiters (e.g., XML tags for code and error sections) to guide the LLM to producing output that could be parsed using regular expressions.

3.3 Three GI Pipelines: Faulty Code, Type Change, Logic Flip

The final stage of our solution design was the GI Pipelines' Stage, incorporated to evaluate the effectiveness of automated evolutionary algorithms in fixing faults left over from previous stages of our solution. We divided our experimental setup into three pipelines with different GI configurations – each incorporating modifications in key components including mutation operators, fitness function, search algorithm, and crossover operator.

In MAGPIE [7], the fitness evaluation of a program variant involves compiling the modified program and its associated test suite, executing the test suite, and calculating the fitness score as the ratio of failing test cases to the total number of test cases. However, this process is sequential, meaning each step is only performed if the preceding step is successful. Consequently, if a program variant fails to compile, the evaluation halts, and that variant is excluded from further consideration in the search process. More critically, if the original input program contains a compile-time error, no mutations are applied at all, and the genetic improvement process terminates prematurely. As many of the transpiled programs in our dataset contained compile-time errors, it was essential to enable the search process to operate even on faulty code. Therefore, we modified MAGPIE to allow the evolutionary process (mutation, selection, and crossover) to continue with broken code. We modified MAGPIE to select the top `k = 10` program variants for crossover, thereby allowing some non-compiling variants to participate in crossover. This introduced greater diversity and improved its ability to repair more complex translation faults.

We introduce two novel mutation operators. The first is `LLM-assisted Type Change Operator` which is designed to modify the data type of a variable by leveraging an external LLM (Section 5.1). (Bouras et al. [8] have also reported successful use of MAGPIE with LLMs.) The LLM-assisted Type Change Operator extracts a target program statement and queries an external LLM to suggest appropriate type modifications. We used Llama3-70b LLM's APIwith a Chain-of-Thought prompting strategy. The prompt used is shown in Listing 1

Listing 1: Prompt for Type Change

```
Here is some expression from a programming language:
    {code}
You are tasked with modifying the datatype in the
    expression if any. If no datatype is present in
    the expression, return the same expression.
Here's an example of how you might approach the task
    when given an expression of the form "int x =
    5;":
1. Identify if any datatype is in the expression,
    here "int" is the datatype.
2. Think of which programming language it might
    belong to, here it is Java.
3. Modify the datatype in the expression. For
    example "int x = 5;" can be modified to "long x
    = 5;".
4. Return only the modified expression like "long x
    = 5;".
5. If no datatype is present in the expression,
    return the same expression "int x = 5;".
Don't return any additional text, only the modified
    or unmodified expression. Here is the expression
    again: {code} Your output response must adhere
    to the text format: Expression: ***the
    expression***. Your output response should not
    contain any text on thinking.
```

The second novel operator is `Boolean Value Change Mutation` which replaces `True` with `False` and vice versa. Like the LLM-assisted Type Change Operator, it operates at the statement level by using Python's `replace()` and searching for the keywords `true`, `false`.

4 Experimental Setup

To address *RQ1* we conducted a manual analysis of the training set of EvalPlus dataset (next section). For *RQ2*, we also passed these files through the LLM preprocessor (previous section) collected the outputs and performed a detailed manual inspection of the remaining bugs to assess its effectiveness. To evaluate *RQ3* and *RQ4* we implemented and compared three GI pipelines described in Sections 4.2 and 4.3. Finally, to investigate *RQ5*, we used Py2Java rule-based translation tool, to compare our system's end-to-end effectiveness with rule-based Python-to-Java translation.

Due to non-determinism of the LLMs and the GI process, we ran our experimental pipeline 5 times end-to-end starting from rule-based test generation to running GI. All experiments in our empirical evaluation were conducted on an Apple MacBook Pro with a M4 chip, running macOS Sequoia 15.3.2.

4.1 Training and Test Datasets

Due to their usage in previous LLM-translation bug studies, we considered the following datasets: CodeNet [19], AVATAR [2], EvalPlus [13], CoST [22], Click [10], and Commons CLI [4]. However, Pan et al. [17] have previously shown that LLMs perform poorly when translating long programs from real-world projects, with translation accuracy falling to nearly zero. In addition to size, and the availability of test cases, the EvalPlus dataset offered the most practical and rigorous environment for evaluating the performance of our repair system. For these reasons, we ultimately selected the EvalPlus dataset which includes single, self-contained Python functions. To conduct our experimental analysis, we randomly split the EvalPlus dataset into a training set of 100 samples and a test set of 65 samples. The training set was used to identify common transpilation errors and inform the development of our approach for addressing them. We then evaluated the effectiveness of it on the test set.

4.2 Common MAGPIE Configurations

Each of the three GI Pipelines shared several core settings. Each began by using `srcML` to convert the source program file into an equivalent XML representation. Similarly since our objective is to fix translation-induced bugs, rather than optimising performance or reducing code size (i.e. removing bloat), all pipelines set MAGPIE's fitness attribute to `repair`.

To ensure a sufficient search capacity to explore a diverse set of program variants through mutation and crossover, we followed typical GI practice [18] of a population size of 100 and 20 generations for each run.

4.3 Three Genetic Improvement Pipelines

GI Pipeline #1 employed MAGPIE's default genetic programming configuration, which included standard XML tree statement level mutation operators `Insert`, `Delete`, and `Replace`, alongside its default crossover strategy, Uniform Concatenation. This crossover uniformly combining program edits from two parents, either by mixing and matching edits between them or by discarding one parent entirely and preserving the more promising variant. The purpose of this pipeline was to evaluate the baseline effectiveness of standard genetic programming techniques in addressing transpilation bugs.

GI Pipeline #2: investigated the effectiveness of manually curated combinations of novel and standard mutation operators. Specifically, we defined two operator subsets: the first combined our LLM-assisted Type Change operator with the standard Replace and Delete operators, while the second combined our Boolean Value Change operator with the same standard set. These combinations were suggested by our analysis of the buggy translation dataset, and so we hoped to design a set of operators to deal with the fault patterns observed. As with Pipeline #1, we used MAGPIE's default Uniform Concatenation crossover.

To ensure that high-fitness candidate patches were not prematurely excluded, we also configured MAGPIE so that both `Offspring_elitism` and `Offspring_crossover` were 1.0. This encouraged exploration by enabling crossover between all candidate patches, regardless of their fitness scores.

To evaluate our custom operators, we applied each subset independently across all faulty translations and assessed their impact on the success of the GI repair process.

GI Pipeline #3: combined standard mutation operators with our LLM-assisted crossover mutation operator. This operator uses an external LLM to perform crossover by providing it with a set of parent candidates, each represented as an edit sequence. The LLM is then tasked with selecting suitable parents and generating offspring through a learned crossover strategy. The LLM returns a list of offspring programs derived from these crossover operations. MAGPIE employed the Llama3-8B model.

5 Results and Discussion

Next, in Sections 5.1 to 5.5, we present the findings of our empirical evaluation and show how they answer our research questions, RQ1: to RQ5: (Section 2 above).

5.1 RQ1: "What are the common transpilation bugs?"

We analysed 100 program files in the EvalPlus training set: 50 gave compilation errors, 11 runtime errors and 39 executed successfully. Later we refer to this base line as *File State(s)*

As 50% of the files failed to compile, we further classified them. There are four primary compilation errors:

22 Invalid Tokens: Files containing unintended LLM response text (e.g., `Sure, here's your code`).

18 Missing Imports: Vital import statements for essential data structures were omitted.

5 Datatype Mismatches: Inconsistencies between datatypes of arguments in the source file and expectations in the test file (e.g., passing `int` instead of `long` as parameters to a function).

5 Missing Testcases: Cases where no tests were generated, were often due to context window limit being exceeded by our LLM prompt.

The high incidence of **Invalid Tokens** and **Missing Imports** is likely to be due to inherent limitations in the LLM. GPT-4o-mini has a smaller context window [15]. This meant that it often struggled with handling long-range dependencies within source code, i.e. keeping track of important information spread across longer program files. This often led to inconsistencies in API usage, data structure handling, and control flow when processing larger programs.

For the classification of the causes of runtime errors, we identified three primary categories:

6 Logic Errors: Files containing logic mistakes, such as performing operations without appropriate checks (e.g., initialising an array variable without verifying that its value is non-negative).

4 Type Selection Error: Files in which numeric computations exceeded the range of variable for the data type, resulting in arithmetic overflow (e.g., attempting to store the 41st Fibonacci number in a 32-bit signed integer).

1 Output Format: consisting of ill-formatted unicode characters in a test case.

By analysing the distribution of compilation and runtime errors, we were able to develop a targeted approach. Specifically, the high incidence of compilation errors stemming from missing imports

Table 1: Mapping Error Groups to Magpie Mutation Operators

Error Group	Error Types Included	Magpie Mutation Operators
Translation-induced Errors	Invalid Tokens, Missing Imports, Missing Test Cases	Insert, Delete, Replace Statements
Type & Logic	Datatype Mismatch, Type Selection Error, Logic Error	LLM-assisted Type Change, Boolean Value Change, Delete, Replace Statements
Output Handling Errors	Output Formatting Error	Insert, Delete, Replace Statements

and invalid tokens motivated the design of our LLM preprocessor. Additionally, errors related to missing test cases and type errors led us to incorporate a rule-based test generation module. Compilation errors caused by datatype mismatches informed the development of novel, custom mutation operators. After constructing our complete solution, we conducted testing on the Evalplus test set, Section 5.1. Table 1 summarises our mapping of transpilation errors to groups of Magpie mutations.

Key Takeaway: Most transpilation bugs are related to compilation errors, which outweigh runtime errors that lead to failing tests. Among these compilation issues, the most prevalent are **Missing Imports** and **Unrecognised Tokens**. Beyond these, two additional categories: **Type Selection** and **Datatype Mismatch**, emerge as particularly relevant for further investigation from a genetic improvement perspective.

5.2 RQ2: Effectiveness of Secondary LLM calls

The first step in our approach to addressing transpilation bugs involved processing each source file through the LLM preprocessor. This step effectively simulates a secondary LLM invocation aimed at improving translation accuracy. As illustrated in Figure 2, the LLM-based preprocessing step improved compilation success rates by **78%** in the training set and **57.8%** in the test set.

However, there were still files which gave compilation errors after preprocessing. We conducted a detailed analysis of the compilation errors that persisted in the eight files of the test set following LLM-based preprocessing (dark green RHS lower part of Figure 2). Among these, six files exhibited type mismatches between the source code and corresponding test files, where the argument type of the function under test differed from the arguments specified in the test assertions. These type mismatches stemmed from limitations in the rule-based test generation process. For example, passing arguments of type `List` where the method signature explicitly required an array. This caused the test case to fail. Also, two files still had unrecognised tokens such as `'End of Code'` at the end of the file, which highlighted the inconsistent formatting of the output produced by the LLM. In Section 5.3.1 we will describe

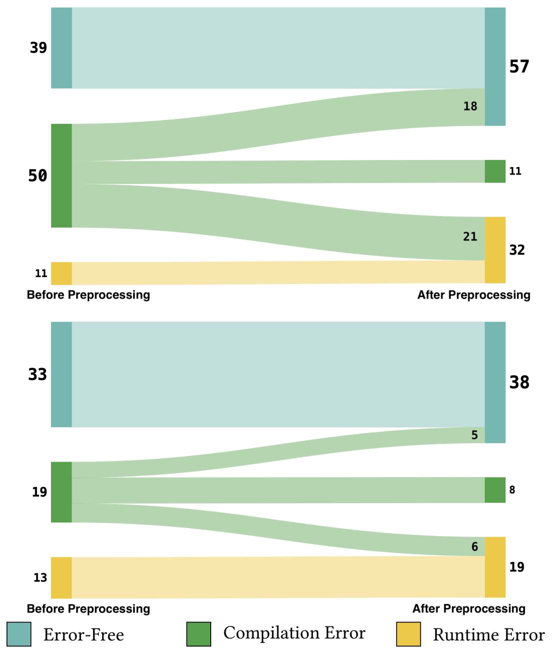

Figure 2: Error transitions: Top train (100), bottom test (65) files

in detail the analysis, which we made for RQ3, of the files in the test set with runtime errors after preprocessing.

To ensure the consistency of secondary calls, we repeated the preprocessing stage across 5 runs and measured the preprocessing success rates across the 5 runs. This was done to evaluate the stability and determinism of the LLM preprocessor when invoked multiple times under identical conditions. Since large language models can exhibit variability in their outputs due to sampling-based generation, even when given the same prompt, it was important to assess whether the preprocessing outcomes remained consistent. In four run on training the success rate was 78% and in one 76%.

The success rate of the LLM preprocessor for the test set remained stable across all five runs, consistently achieving **57.8%**. This stability suggests that the LLM's output was highly consistent and free from large variation or hallucinations. We suggest this may be partially attributed to prompt caching mechanisms employed by OpenAI [16], which can reduce output variability when identical prompts are reused. In contrast, the train set exhibited a slight dip in success rate at Turn 4 before returning to the previous level. Upon investigating the underlying cause, we identified that the model (GPT-4o-mini) had omitted a closing parenthesis in one of the Java files during that turn, leading to a compilation failure and a temporary decline in preprocessing success.

In addition to evaluating error correction, we also collected supplementary statistics during the preprocessing stage. Specifically, we measured the average preprocessing time as well as the compilation times before and after preprocessing. This analysis aimed to identify any implicit performance optimisations introduced by the LLM, such as faster compilation resulting from improved or simplified code, even in the absence of explicit directives. The recorded statistics for both the training and test sets are presented below.

Table 2: Mean time (seconds) for the Preprocessing Stage

Dataset	Preprocessing	Compilation (Before Pre-)	Compilation (After Pre-)
`Train`	22.17 ± 17.50	0.40 ± 0.04	0.43 ± 0.04
`Test`	17.27 ± 6.70	0.36 ± 0.03	0.40 ± 0.04

As shown in Table 2, average compilation times increased slightly after preprocessing compared to before. This is expected, as preprocessed files compiled successfully and fully, whereas failed files likely terminated early due to syntax or semantic errors.

In contrast, the LLM preprocessor introduced the largest time overhead, averaging 22.17 seconds for the training set and 17.27 seconds for the test set. This disparity is expected due to the latency involved in querying a large language model, particularly for larger code files, which includes network communication, model inference, and response processing. Most of the overhead stemmed from the LLM's compute-intensive effort to parse and correct large inputs. Even the smaller variant such as GPT-4o-mini we used still requires substantial computational resources to perform inference, especially under high-load conditions. Furthermore, as the preprocessor relies on remote LLM services, overall performance is dependent upon network stability and bandwidth; intermittent connectivity or server-side queuing can introduce additional, non-deterministic delays.

Key Takeaway: Secondary LLM calls proved effective in resolving **78%** of previously non-compiling files in the train set and **57.8%** in the test set. Although resource-intensive, these invocations of the remote LLM successfully addressed the majority of **Missing Imports** and **Unrecognised Tokens**, enabling the programs to reach a compilable state suitable for APR genetic improvement-based repair.

5.3 RQ3: GI Tool's Performance on Transpilation Bugs

We evaluate the performance of MAGPIE in addressing transpilation bugs generated by LLMs, under two distinct experimental settings. The first is MAGPIE's default configuration, which employs standard mutation operators (`Insert`, `Delete`, and `Replace`) alongside the Uniform Concatenation crossover operator. The second configuration adds LLM-assisted crossover. The following 2 subsections (5.3.1 and 5.3.2) present and analyse the results obtained from each configuration, highlighting their effectiveness in generating correct program fixes. As in the previous stage of our approach, MAGPIE was executed five times, each time using the corresponding output generated by one of the five runs of the LLM preprocessor.

5.3.1 RQ3.1: GI Performance with Standard Strategies. We evaluated MAGPIE's performance under its default configuration on train and test files exhibiting both compilation and runtime errors.

Table 3: MAGPIE performance using standard mutation and crossover operators on files with transpilation bugs

Dataset	Error Type	Files Processed	Fixes
Train	Compilation Error	11	0
	Runtime Error	32	0
Test	Compilation Error	8	0
	Runtime Error	19	0

Table 4: Runtime error types remaining after preprocessing

Error Type	Number of Files
Runtime Exceeded	2
Type Selection Error	5
Logic Error	12

Our results from the 5 experimental runs show that MAGPIE's standard mutation and crossover strategies had no measurable effect in repairing transpilation faults across both compilation and runtime error categories. In all cases, the evolved variants either failed to compile or did not lead to improved program fitness.

As summarised in Table 3, it was found that running MAGPIE with its standard configuration did not yield correct or plausible patches to improve the fitness of the program. To understand the limitation of the standard MAGPIE configuration, we performed a post experimental analysis of the kind of runtime errors in the test set. Table 4 contains our results summarised from our post experimental analysis.

As shown in Table 4, the majority of runtime errors encountered in the test set were logic-related. These errors stemmed from issues such as missing method calls (e.g., `.split()`), missing input validation, or improper loop initialisation values. Consequently, the outputs generated by the buggy functions deviated from the expected outputs defined in the test suite, leading to test failures. Many of these logical errors required non-trivial corrections that were beyond the capabilities of standard mutation operators (i.e., `Insert`, `Delete`, and `Replace`) employed by MAGPIE.

However, upon manual inspection of two files exhibiting logic errors, we observed that MAGPIE was capable of generating plausible fixes. That is, patches that improved runtime behaviour without fully correcting the bug. To further explore this, we reran MAGPIE for five additional iterations on these files and discovered one run in which it successfully composed a sequence of edits that yielded a **50%** improvement in the fitness score. Despite this improvement, the resulting patches did not fully resolve the underlying logical bugs and would likely fail under a more comprehensive test suite.

Among the remaining error types, issues due to incorrect type selection were prevalent. In several cases, functions intended to return numeric values produced incorrect results due to arithmetic overflow, stemming from the use of an incorrectly chosen data type for storing the computed result. A notable example involved a program calculating Fibonacci numbers: although the function returned an integer, the 41st Fibonacci number exceeds the bounds of Java's 32-bit signed integer type, resulting in overflow. Such issues were also beyond the scope of correction using standard mutation operators.

A particularly insightful observation was that, while standard mutation operators occasionally produced patches that enabled previously failing test cases to pass, they often did so at the cost of causing previously passing cases to fail. This behaviour led to patches that increased the overall fitness score but diverged from the goal of achieving zero failing tests.

As for the compilation errors, our analysis in Section 5.2 we found that more than half the errors were due to a type mismatch in the arguments of the function being tested whereas the rest were either due to invalid tokens or missing dependencies. For the programs with type mismatches, fixes could not be found by a combination of `Insert`, `Delete`, and `Replace` operators.

We had expected errors with invalid tokens to have been resolved by MAGPIE's deletion operator. However, upon inspecting the XML representation generated for this code snippet, we found that the '`End of Code`' text was not recognised as a valid program statement. As a result, the deletion operator could not be applied to the XML node corresponding to the unrecognised tokens.

5.3.2 RQ3.2: GI Performance with LLM-assisted Crossover. Similar to our findings with the Uniform Concatenation crossover operator in MAGPIE, the use of LLM-assisted crossover did not lead to measurable improvements in any of the target files. Despite leveraging a large language model to guide the combination of program mutations, the resulting variants failed to produce any fixes which improved the fitness of the program.

A key factor contributing to this was the form in which mutation information was provided to the model. Specifically, edits were expressed in an abstract format such as `SrcmlStmtDeletion((<file name>, <node_tag>, <numeric_location>))`. While this representation is suitable for tracking changes internally, it likely lacked sufficient context for the LLM to meaningfully reason about the underlying program logic or the intended effect of the mutation.

As a result, many LLM-assisted crossover attempts were effectively random, producing variants that neither improved the fitness of the program nor provided meaningful diversity. The lack of grounding in concrete code context limited the model's ability to generate semantically relevant edits.

Furthermore, the LLM's output frequently deviated from the structured format specified in the prompting instructions. This misalignment often caused the crossover phase to be skipped entirely due to parsing failures, thereby reducing the number of valid candidates explored.

Overall, while LLMs offer powerful language modelling capabilities, their effectiveness in GI crossover tasks appears limited without additional context.

Key Takeaway: Although we initially thought that standard mutation operators and LLM-based crossover strategies could repair transpilation errors, our experimental results demonstrated that these approaches were largely ineffective at producing plausible patches.

5.4 RQ4: Effectiveness of Novel Mutation Operators

Next we discuss our findings on the effectiveness at addressing transpilation bugs that were either present in the original dataset or remained after preprocessing of two novel mutation operators: LLM-assisted Type Change (Section 5.4.1 and Boolean Value Change (Section 5.4.2). We assess their impact on both compilation and runtime errors and provide a critical analysis of the reasons behind their effectiveness.

5.4.1 Performance of LLM-assisted Type Change Operator. The LLM-Assisted Type Change Operator was highly effective in reducing compilation errors, achieving a **45%** (11 $\rightarrow$ 6) reduction in the train set and a **50%** (8 $\rightarrow$ 4) reduction in the test set. Additionally, the operator reduced runtime errors by **23%** (32 $\rightarrow$ 26) in the train set and **26%** (19 $\rightarrow$ 14) in the test set. These results support our initial hypothesis that transpilation bugs necessitate the introduction of a novel Type Change Operator. Upon further analysis, we identified that the primary issues addressed by our operator were **Type Selection Errors** and **Datatype Mismatches**, as discussed in earlier sections. Notably, a closer examination of the logs for the fixed files revealed that the fixes often involved a combination of two or more mutations applied sequentially. Interestingly, these individual mutations frequently had a fitness value of `inf`, as they did not compile on their own. This suggests that allowing weaker program variants to crossover, despite their inability to compile individually, can lead to the generation of a correct fix in the context of transpilation bugs. The success of the LLM-Assisted Type Change Operator can also be attributed to the special modifications and configurations applied to MAGPIE. Specifically, we observed that setting the `Offspring_elitism` and `Offspring_crossover` variables to 1.0 was crucial in enabling weaker variants to crossover and be added to the search space. Here, `Offspring_elitism` refers to the proportion of the previous population that is carried over unchanged into the next generation, while `Offspring_crossover` denotes the proportion of the previous population selected to undergo crossover with randomly chosen valid parents. Without these configurations, MAGPIE exhibited inconsistency in identifying the correct fix, frequently exhausting its step budget before finding a solution.

5.4.2 Performance of Boolean Value Change Operator. The Boolean Value Change Operator did not lead to any measurable improvement in the overall success rate of program repair. In several cases, while it was able to fix some failing test cases, it introduced regressions. This suggests that the operator may be too coarse-grained, making broad changes that disrupt the program's existing logic rather than targeting the root cause.

The trade-off between fixing and breaking behaviour suggests that Boolean mutations may result in overfitting to specific test cases rather than producing generalisable fixes. As such, its application in the context of transpilation bugs appears limited, and further refinement would be necessary for it to be reliably useful in this setting.

Table 5: Py2Java compilation error types remaining after preprocessing

Error Type	Number of Files
Datatype Mismatch	20
Incorrect API Usage	5

Table 6: Py2Java runtime error types after preprocessing

Error Type	Number of Files
Logic Error	9
Output Formatting	2
Type Selection Error	2

Key Takeaway: The **LLM-assisted Type Change Operator** was effective in addressing **Datatype Mismatch** and **Type Selection** errors, reducing compilation errors by **50%** and runtime errors by **26%** in the test set. In contrast, the **Boolean Value Change Operator** was largely ineffective at fixing transpilation bugs.

5.5 RQ5: Performance Rule-Based Transpiled Code Py2Java

In the final stage of our research, we conducted an empirical evaluation of our proposed approach on code generated by a rule-based transpiler. Specifically, we tested the robustness of our method on outputs produced by the Py2Java translator. Our findings showed that Py2Java achieved a 0% success rate, where a translation is deemed successful if the resulting Java file at least compiles without errors.

This outcome is primarily due to the complexity of the programs in the Evalplus dataset, which make use of advanced Python features and diverse data structures. The Py2Java tool lacks the sophisticated rules required to accurately translate such constructs into Java, and thus was unable to perform successful translations on its own.

To further understand the nature of the failures, we manually inspected a sample of the translations and observed common patterns such as incorrect type assignments, missing class or method definitions, and invalid syntax constructs that have no direct Java equivalent. These observations highlight the limitations of relying solely on handcrafted rules.

Consequently, our approach relied heavily on the secondary LLM-based preprocessing step to resolve syntactic issues, infer types for complex data structures, and address missing dependencies. We observe that LLM-Preprocessing resulted in half of the compilation error files being fixed, thereby proving the effectiveness of secondary LLM calls. Since the distributions of the train and test sets were similar, we decided to perform a manual analysis on the test set to discover the different kind of compilation and runtime errors leftover after the preprocessing stage. A breakdown of the error types are shown in Tables 5 and 6.

Except for the Incorrect API Usage errors (Table 5), several of the errors observed after preprocessing were consistent with those seen in LLM-based translations. These specific errors arose from the use of functions that exist in Python but have no direct equivalents in Java, leading to compilation failures. The rule-based translator, constrained by its limited set of transformation rules, lacked the ability to map such APIs across languages. Additionally, the LLM-based preprocessor likely failed to correct these issues due to the presence

Table 7: Type-Change Operator on rule-based transpiled code

Dataset	Error Type	Num Files	Fixes	Improvement
Train	Compilation Error	32	4	12.5%
	Runtime Error	23	3	13.0%
Test	Compilation Error	25	5	20.0%
	Runtime Error	13	2	15.3%

of multiple other errors in the same file, compounding the difficulty. It appears that GPT-4o-mini, which we used for preprocessing, is not capable of resolving all the complex syntactic issues within those files.

The error distributions presented in Tables 5 and 6 differ from those observed in LLM-based translations. We attribute this to the fundamental differences in how type inference is handled by rule-based versus LLM-based translation approaches. Specifically, we observed a higher incidence of data type mismatch errors in the Py2Java rule-based translations, largely because it tends to rely on rigid heuristics, which often default to a single data type, such as arrays.

When these error-prone files were passed through the GI pipelines, no measurable improvement was observed with Pipelines #1 and #2, which employed standard mutation operators and standard or LLM-assisted crossover techniques. Additionally, even Pipeline #3 with the Boolean Value Change Operator saw no measurable improvement. This outcome mirrored the results previously seen with LLM-based translations. In contrast, our pipeline incorporating the LLM-Assisted Type Change operator demonstrated a modest but meaningful improvement, Table 7. Specifically, it reduced the number of runtime error files from **23** to **20** in the train set and from **13** to **11** in the test set, primarily resolving errors related to type issues that caused arithmetic overflows. Regarding compilation errors, the Type Change operator reduced the number of erroneous files by **4** on the train set and **5** on the test set. These corrected files predominantly involved type mismatches between the source code and its corresponding test cases. A summary of these results is presented in Table 7.

Further analysis revealed that out of the 20 files in the test set affected by type mismatch errors, only 5 were successfully repaired. This limited success was due to dependencies within the code that relied on the original argument types. For example, while changing a test case argument to the correct type (e.g., from `List` to an array) resolved the immediate mismatch, however subsequent statements, such as the use of the `get()` method, became invalid, introducing new errors.

Key Takeaway: Our approach yielded similar results on rule-based translated code as it did on LLM-translated code, both showing a reduction in compilation errors after preprocessing, and further reductions in both compilation and runtime errors following GI-based repair with the LLM-assisted Type Change Operator. However, the Type Change Operator achieved only **20%** reduction in compilation errors and **15.3%** in runtime errors.

6 Conclusion

The cost of refactoring can be horrendous in both time and money. Even a single project moving a code base between well known industry standard languages can run into hundreds of millions of dollars and take multiple years. This is well known but modern artificial intelligence techniques, such as Large Language Models (LLMs), unaided, cannot be trusted with computer program refactoring. Yet this is the first time LLM have been combined with Genetic Improvement for program source-to-source transpilation. Our results are based on state-of-the-art LLMs (OpenAI's commercial GPT-4o-mini) and genetic improvement tools (open source Magpie). With widely used datasets (EvalPlus), we found our novel genetic operations gave a **33%** increase in error free files for LLM-based translations and a **18%** increase for rule-based translations.

References

[1] Karan Aggarwal, Mohammad Salameh, and Abram Hindle. 2015. Using machine translation for converting Python 2 to Python 3 code. PeerJ PrePrints 3:e1459v1.
[2] Wasi Uddin Ahmad et al. 2023. AVATAR: A Parallel Corpus for Java-Python Program Translation. In *Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9-14, 2023*. 2268–2281.
[3] Paul F. Albrecht et al. 1980. Source-to-source translation: Ada to Pascal and Pascal to Ada. *SIGPLAN Not.* 15, 11 (1980), 183–193.
[4] Apache Commons CLI. 2023. Apache Commons CLI. https://commons.apache.org/proper/commons-cli/. Accessed: 2025-04-01.
[5] F. Andrés Bastidas and María Pérez. 2018. A systematic review on Transpiler usage for Transaction-Oriented Applications. In *2018 IEEE Third Ecuador Technical Chapters Meeting (ETCM)*.
[6] Andrés Bastidas Fuertes, María Pérez, and Jaime Meza Hormaza. 2023. Transpilers: A Systematic Mapping Review of Their Usage in Research and Industry. *Applied Sciences* 13, 6 (2023). doi:10.3390/app13063667
[7] Aymeric Blot and Justyna Petke. 2022. MAGPIE: Machine Automated General Performance Improvement via Evolution of Software. arXiv 2208.02811. doi:10.48550/ARXIV.2208.02811
[8] Dimitrios Stamatios Bouras, Sergey Mechtaev, and Justyna Petke. 2025. LLM-Assisted Crossover in Genetic Improvement of Software. In *GI 2025*. Ottawa, 19–26. doi:10.1109/GI66624.2025.00012
[9] Banghao Chen et al. 2023. Unleashing the potential of prompt engineering in Large Language Models: a comprehensive review. arXiv 2310.14735.
[10] Click. 2023. Click. https://click.palletsprojects.com/en/8.1.x/. Accessed: 2025-04-01.
[11] IT News Staff. 2024. CommBank declares core bank overhaul complete. https://www.itnews.com.au/news/commbank-declares-core-bank-overhaul-complete-321165
[12] Claire Le Goues et al. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE TSE* 38, 1 (2012), 54–72. doi:10.1109/TSE.2011.104
[13] Jiawei Liu et al. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *NeurIPS 2023*.
[14] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2013. Lexical statistical machine translation for language migration. In *ESEC/FSE 2013*. 4 pages.
[15] OpenAI. 2023. GPT-4 Technical Report. arXiv 2303.08774.
[16] OpenAI. 2024. Prompt Caching Guide. https://platform.openai.com/docs/guides/prompt-caching. Accessed: 2025-04-13.
[17] Rangeet Pan et al. 2024. Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code. In *ICSE 2024*.
[18] 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*. doi:10.1007/978-3-662-44303-3_12
[19] Ruchir Puri et al. 2021. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. In *NeurIPS Datasets and Benchmarks 2021*.
[20] Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised Translation of Programming Languages. In *NeurIPS 2020*. https://proceedings.neurips.cc/paper/2020/hash/ed23fbf18c2cd35f8c7f8de44f85c08d-Abstract.html
[21] Ashish Vaswani et al. 2017. Attention is all you need. In *NIPS 2017*. 6000–6010.
[22] Ming Zhu, Karthik Suresh, and Chandan K Reddy. 2022. Multilingual Code Snippets Training for Program Translation. In *AAAI Conf. on AI*. 11783–11790.