Applying Genetic Improvement Techniques for Automated Program Repair of Transpiled Code

Prasham Jadhwani, Carol Hanna,
William B. Langdon, Justyna Petke

University College London

GI '26 • April 12–18, 2026 • Rio de Janeiro, Brazil

Why Code Transpilation Matters

$750M

Commonwealth Bank of Australia spent over 5 years migrating COBOL to Java

The Translation Gap

Even state-of-the-art LLMs produce syntactically incorrect translations

Residual errors persist after applying best translation models

Rule-Based

Rigid, limited language coverage

Machine Translation

High lexical accuracy but parsing errors

Deep Learning / LLMs

>80% accuracy but residual errors remain

Why Genetic Improvement?

GI Advantages

- Automatically adapts and evolves program fixes
- Well-suited for diverse and unpredictable LLM errors
- Uses evolutionary search to detect and correct bugs
- Reduces need for manual post-translation intervention

Research Gap

First time LLMs have been combined with Genetic Improvement for source-to-source transpilation repair.

Key Contribution

Novel LLM-assisted mutation operators integrated into MAGPIE framework for automated transpilation repair

Research Questions

RQ1 What are common transpilation bugs and how do they map to mutation operators?

RQ2 Can secondary LLM calls improve buggy transpiled code for GI tools?

RQ3 How effective are standard & LLM-augmented GI tools on LLM-translated code?

RQ4 How do custom mutation operators compare to traditional ones?

RQ5 How does performance differ on rule-based vs LLM-based transpiled code?

Proposed Solution: Three-Stage Pipeline

1

Rule-Based Test Generation

Parse Python test cases and output JUnit 4 syntax

Python Source

2

LLM Preprocessor

GPT-4o-mini fixes syntax errors

Compilable Java

3

GI Pipelines

Evolutionary repair with MAGPIE

Repaired Java

Stage 1: Rule-Based Test Generation

Process

1. Extract assert statements using regex (assertEqual, assertTrue, assertFalse)
2. Parse assertion arguments, handling edge cases (commas in lists/strings)
3. Convert function names to camelCase and map Python types to Java types
4. Output test cases in JUnit 4 syntax

⚠ Why Custom?

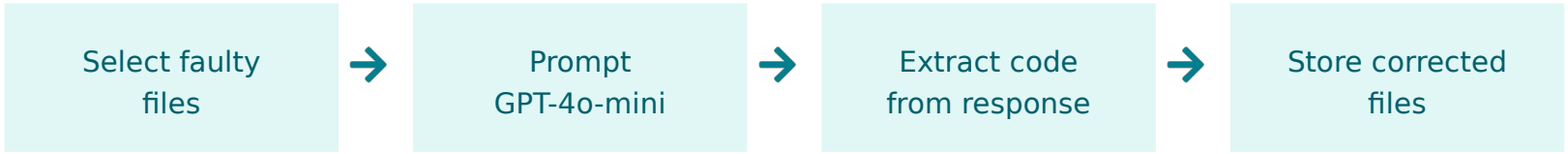
Existing translators (p2j, Py2Java) lacked native support for Python test cases.

Custom script provides flexibility and JUnit 4 compatibility.

Type Mapping

Python types mapped to Java:
List, int, long, float, double,
boolean, Object (fallback for strings)

Stage 2: LLM Preprocessor



Prompting Strategy

Structured Prompting with GPT-4o-mini using XML tag delimiters for code and error sections.

Regular expressions extract code blocks from LLM response text.

Why This Stage?

50% of initial translations had compilation errors (missing imports, invalid tokens).

Secondary LLM call resolves most of these before GI-based repair.

Stage 3: Three GI Pipelines

Pipeline #1: Baseline	Standard MAGPIE with Insert, Delete, Replace + Uniform Concatenation crossover
Pipeline #2: Custom Operators	Novel LLM-assisted Type Change + Boolean Value Change operators with standard crossover
Pipeline #3: LLM Crossover	Standard mutation operators + LLM-assisted crossover (Llama3-8B for parent selection)

Common Settings: Population = 100 | Generations = 20 | srcML for XML conversion | Fitness = repair

Novel Mutation Operators

LLM-Assisted Type Change

Extracts a target statement and queries Llama3-70b to suggest type modifications.

Uses Chain-of-Thought prompting:

1. Identify datatype in expression
2. Determine the programming language
3. Modify the datatype
4. Return only the modified expression

```
int x = 5;  →  long x = 5;
```

Boolean Value Change

Replaces true with false and vice versa at the statement level.

Uses Python's replace() to search for boolean keywords.

MAGPIE Modification

Modified MAGPIE to allow evolutionary process to continue with broken code. Top k=10 variants selected for crossover, including non-compiling variants.

Experimental Setup

Dataset: EvalPlus

Single, self-contained Python functions selected for practical evaluation of repair system.

Split: 100 training + 65 test samples

Experiment Details

5 end-to-end runs (non-determinism)
LLM translation: GPT-4o-mini
Type Change: Llama3-70b
LLM crossover: Llama3-8b
Rule-based: Py2Java

Translation: Python → Java | GI Tool: MAGPIE (open source) | Hardware: MacBook Pro M4
Training set used to identify bugs and develop approach; test set for evaluation

RQ1: Common Transpilation Bugs

Analysis of 100 training files:

50 Compilation Errors

11 Runtime Errors

39 Executed Successfully

Compilation Errors

22	**Invalid Tokens**
18	**Missing Imports**
5	Datatype Mismatches
5	Missing Testcases

Runtime Errors

6	**Logic Errors**
4	**Type Selection Errors**
1	Output Format

Bug Taxonomy → Mutation Operators

Error Group	Error Types	MAGPIE Operators
Translation-induced	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**
Output Handling	Output Formatting Error	Insert, Delete, Replace Statements

Most bugs are compilation errors (Missing Imports & Invalid Tokens). Type Selection and Datatype
Mismatch are key targets for genetic improvement-based repair.

RQ2: LLM Preprocessing Effectiveness

78%

of non-compiling files fixed in training set

57.8%

of non-compiling files fixed in test set

High Consistency

Test set: 57.8% across all 5 runs
Train set: 78% in 4 runs, 76% in 1

Stability likely due to OpenAI prompt caching mechanisms.

Remaining Errors (Test Set)

8 files still had compilation errors:
6 type mismatches (source ↔ test)
2 unrecognised tokens

19 files with runtime errors

RQ3: Standard GI & LLM Crossover

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

Why No Fixes?

Logic errors need non-trivial corrections beyond Insert/Delete/Replace scope. Type overflow issues can't be fixed by standard operators.

LLM Crossover Also Failed

Abstract edit format lacked context for LLM reasoning. Output often deviated from structured format, causing parsing failures.

RQ4: Novel Operators on LLM Translations

LLM-Assisted Type Change

	Before	After	Reduction
Compilation (Train)	11	6	**45%**
Runtime (Train)	32	26	**23%**
Compilation (Test)	8	4	**50%**
Runtime (Test)	19	14	**26%**

Boolean Value Change

No measurable improvement

Fixed some cases but introduced regressions. Too coarse-grained.

Key Insight: Setting Offspring_elitism and Offspring_crossover to 1.0 was crucial — weaker, non-compiling
variants could crossover, eventually producing correct fixes through combined mutations.

RQ5: Rule-Based (Py2Java) Results

0%

Py2Java baseline success rate (no files compiled)

→

50%

Compilation errors fixed after LLM preprocessing

→

+18%

Further improvement with Type Change

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

Key Results Summary

33%

increase in error-free files
for LLM-based translations

18%

increase in error-free files
for rule-based translations

What Worked

- LLM preprocessing (78% fix rate)

LLM-assisted Type Change operator

Allowing non-compiling crossover

What Didn't

- Standard GI operators (0 fixes)

LLM-assisted crossover (abstract edits)

Boolean Value Change (regressions)

Conclusion

- First combination of LLMs with Genetic Improvement for source-to-source transpilation
- Novel LLM-assisted Type Change Operator reduced bugs by 33% (LLM) and 18% (rule-based)
- Secondary LLM preprocessing resolved majority of compilation errors
- Comprehensive bug taxonomy maps transpilation faults to mutation operators
- Allowing non-compiling variants in crossover was essential for complex fixes

Thank You!

Prasham Jadhwani • prasham.jadhwani.21@alumni.ucl.ac.uk