

Code Transformation Rule Synthesis using LLMs: Potential and Limits

Axel Allain[✉]

Univ Rennes, INRIA, CNRS, IRISA
Rennes, France
axel.allain@inria.fr

Djamel Eddine Khelladi

Univ Rennes, INRIA, CNRS, IRISA
Rennes, France
djamel-eddine.khelladi@irisa.fr

Aymeric Blot

Univ Rennes, INRIA, CNRS, IRISA
Rennes, France
aymeric.blot@irisa.fr

Mathieu Acher

Univ Rennes, INRIA, CNRS, IRISA
Rennes, France
mathieu.acher@irisa.fr

Abstract

Due to their black-box nature, LLMs suffer from limited explainability and a lack of determinism. Their usage cost can also rise, particularly with repetitive tasks on large codebases. To mitigate this, we conduct a novel empirical study targeting three domain-specific languages for transformation rules, namely Comby, GritQL, and Ast-Grep. We evaluate three LLMs (GPT-5.4, GPT-oss-120B, and Llama3.1-8B) on six diverse datasets covering four software-evolution tasks: API misuse correction, program repair, API migration, and language version migration. Our results provide evidence that transformation rule synthesis moves beyond proof-of-concept with strong frontier models. GPT-5.4 achieves consistently high rule applicability rates and produces transformations closest to the ground truth across most benchmarks. Smaller and open-weight GPT-oss-120B and Llama3.1-8B models remain effective for simpler, localized changes but struggle with complex migration scenarios. We also observe non-negligible generalizability through the usage of meta-variables and through a high reuse score in the first quartile of many datasets. Finally, when compared to the anti-unification algorithm, LLMs outperform it in correctness, but underperform in rule applicability. Overall, our results show great potential for LLMs to generate sound, correct, generalizable, and reusable rules.

CCS Concepts

- **Software and its engineering** → **Software maintenance tools**;
- **Computing methodologies** → **Artificial intelligence**.

Keywords

Large Language Models, Program Repair, API Misuse, API Migration, Language Version Migration, Transformation Rules

ACM Reference Format:

Axel Allain, Aymeric Blot, Djamel Eddine Khelladi, and Mathieu Acher. 2026. Code Transformation Rule Synthesis using LLMs: Potential and Limits. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26)*, October 12–16, 2026, Munich, Germany. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3832783.3837504>



This work is licensed under a Creative Commons Attribution 4.0 International License. ASE '26, Munich, Germany

© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2882-2/2026/10
<https://doi.org/10.1145/3832783.3837504>

1 Introduction

Software continuously evolves in scale and complexity to accommodate technological advances and emerging use cases [38, 42, 44]. Many challenges arise from the costly phase of software maintenance and evolution [1, 8, 15]. For example, as APIs and programming frameworks evolve, developers must continuously evolve their code and repeatedly cope with various tasks, such as program repair, refactoring, API misuse fixing, code migration, co-evolution, etc. These evolution tasks are tedious, error-prone, time-consuming, and drive up the costs of maintenance. Therefore, automating these laborious tasks is highly beneficial for developers, as it increases efficiency and allows them to focus on higher-value development efforts. Rich literature exists on automating the above tasks [3, 4, 16, 25, 27–32, 34–37, 45], and recent advances show that LLMs are becoming increasingly effective in resolving them [9–11, 20, 24, 43, 49, 50, 52, 53].

However, as the code is generated directly by the LLMs, it still faces major challenges. Indeed, LLMs lack explainability, determinism, and systematic reuse with reduced costs. The black-box behavior of LLMs limits their adoption in critical systems and large-scale environments as they need safe code transformations that are explainable and deterministic. Moreover, in large-scale codebase the use of LLMs for systematic code transformation can lead to increased costs, particularly for organizations with limited resources.

To mitigate these challenges, developers can rely on code transformation DSLs to write transformation rules, which provide a high degree of explainability, determinism, and reuse on large codebase. Yet, transformation rules written in existing domain specific languages (DSLs) such as Comby, GritQL, and Ast-Grep often have a steep learning curve and require expertise to use effectively. Writing and maintaining these rules can be tedious, time-consuming, and error-prone because of their rigid nature, limiting their usability across projects. Hence, an opportunity arises to leverage LLMs to generate the code transformation rules rather than directly updating the code. Yet, to the best of our knowledge, there is a lack of work on LLMs' ability to formalize and generate code transformation rules. We fill this gap.

In this paper, we present a novel empirical study on the ability of LLMs to generate transformation rules by inferring DSL-based rewrite patterns from observed code changes, i.e., pairs of original and evolved code. Due to large-scale pretraining on code transformation corpora, LLMs have developed a strong ability to learn

semantic representations of code changes, including API mappings, program repair strategies, refactoring, and migration/co-evolution patterns. To support rule generation, we provide the LLM with representative examples of documentation and transformation rules from different DSLs, allowing us to assess how additional context influences its ability to infer correct transformation rules. We also experiment with augmenting prompts using examples of documentation and rules, including a limited use of retrieval for one DSL (Comby), to assess its impact on rule generation.

We evaluate the capability of three LLMs (*GPT-5.4*, *GPT-oss-120B*, and *Llama3.1-8B*) to generate transformation rules across four software engineering tasks, program repair, API misuse correction, API migration, and language version migration, on six dedicated benchmarks. Our goal is to identify which DSLs and types of transformation rules are most effectively handled by LLMs and which remain the most challenging. Our results provide evidence that rule synthesis moves beyond proof-of-concept with strong frontier models (*GPT-5.4*), while revealing that effectiveness depends strongly on the software-evolution task and the fit between the edit structure and the rule abstraction. We also found that the generated rules can abstract code transformations through the use of metavariables, resulting in high reuse scores across many datasets. Moreover, compared with the anti-unification baseline, LLMs produce more correct and semantically meaningful rules while avoiding the over-generalization inherent to the anti-unification algorithm. Finally, test-based validation on the program repair benchmarks showed that only a small fraction of the generated transformations that failed AST matching successfully repaired bugs.

In summary, our novel contributions are as follows:

A systematic evaluation of three LLMs on transformation rule synthesis, analyzing their soundness, correctness, and generalization across three DSLs, four code evolution tasks, and six datasets.

A comparative analysis across three transformation rule DSLs, providing evidence on the extent to which LLMs understand these languages, including the effect of retrieval-augmented generation (RAG) on rule quality.

A qualitative analysis of success and failure modes that explains the concrete mechanisms behind model differences, including formatting failures, over- and under-generalization, and missing edits.

Insights on task suitability for rule synthesis: showing that localized edits are substantially more amenable to general, reusable rule synthesis than function-level or multi-statement edits.

A replication package is available at <https://anonymous.4open.science/r/EmpiricalStudyTransformationRules-13B3>

2 Background

To reduce the burden of recurrent changes in large codebase, developers often rely on code search and transformation tools. Search tools such as `grep` and `ripgrep` efficiently locate code fragments, while transformation tools such as `sed` and `sd` automate repetitive code edits. These tools help developers perform large-scale modifications while reducing manual effort. However, using tools such as `sed`, `sd`, or regular expressions requires substantial expertise and often involves a steep learning curve. Moreover, these techniques operate purely at the textual level and cannot capture the

syntactic structure of programs, which may lead to incorrect or unintended transformations. To overcome these limitations, dedicated code transformation DSLs have been developed to make transformation rules easier to write and reuse. These DSLs typically rely on matching patterns enriched with metavariables, enabling developers to express transformations at the syntactic or AST level rather than as plain text. Most DSLs consist of two main components: a matching pattern and a rewriting pattern. The matching pattern captures program elements, such as syntax fragments or AST nodes, using metavariables. The rewriting pattern then reuses these metavariables to generate the transformed code, preserving relevant program elements while simplifying the specification of transformation rules. Several tools implement such rule-based transformation DSLs, including Coccinelle for C programs and Linux kernel co-evolution, Comby for structural code search and rewrite, and AST-based tools such as Ast-Grep and GritQL.

3 Motivating Example

Large-scale codebase often require developers to apply repetitive transformations across many files. Manually performing these modifications is tedious and error-prone, while repeatedly prompting large language models (LLMs) to perform each individual change may increase computational and token costs. At the same time, LLMs remain vulnerable to hallucination issues which may introduce errors during the code modification process. Hence, leading to a lack of explainability and determinism in code evolution.

However, one can address these challenges with code transformation rules. As an illustrative example, consider the diff below.

```
- if (user != null) { sendEmail(user); }
+ Optional.ofNullable(user)
+   .ifPresent(this::sendEmail);
```

This change replaces an explicit null checks with the more concise `Optional` API. While the transformation is simple, it may need to be applied in various contexts hundreds or thousands of times across a large project, making it well suited for automated refactoring. Using a rule-based transformation DSL such as Comby, this refactoring can be expressed as a reusable transformation rule:

```
[optionalIfPresent]
match = "if ([:var] != null) { :[func]([:var]); }"
rewrite = "Optional.ofNullable(:[var])
           .ifPresent(this::[:func]);"
```

Once defined, the rule can be applied automatically across the entire codebase, transforming all matching occurrences in a single pass. The advantage of rule-based code transformations becomes more apparent at scale. If a change occurs n times in a project, an LLM-based workflow may require either repeatedly prompting the model for each occurrence or providing large contexts containing many code fragments. Both approaches increase token consumption and latency, driving up the costs. In contrast, a transformation rule is written once and applied automatically to all matches, providing a deterministic, explainable, and efficient solution.

Recent advances in LLMs like GPT or Claude Code suggest that they may assist developers in generating such transformation rules automatically. Our hypothesis is that given pairs of code snippets representing before and after versions of a change, an LLM could

potentially synthesize a generalized rule that captures the transformation pattern. However, it remains unclear whether LLMs can reliably synthesize correct and reusable transformation rules from example changes. Furthermore, many transformation DSLs exist, each with different syntactic and structural matching capabilities, and to the best of our knowledge, there has been no systematic evaluation of how well LLMs can generate transformation rules for them. In this paper, we address this gap through a novel empirical study that evaluates the ability of LLMs to synthesize transformation rules from example code changes across three popular rule-based transformation tools: Comby, Ast-Grep, and GritQL.

4 Research Questions

With this empirical study we aim to answer the following four RQs:

RQ1: To what extent can LLMs generate sound transformation rules from code diffs while maintaining reasonable computational and financial cost? This assesses the ability of different LLMs to infer valid transformation rules, considering the costs of generating rules in terms of token expense.

RQ2: How complete are the generated transformation rules in terms of coverage, structural richness, and generalization capacity? This aims to cover whether the generated rules capture the code changes in the datasets generalizing beyond individual examples using metavariables.

RQ3: How well do the generated transformation rules reproduce the ground-truth transformations in terms of semantic and syntactic similarity? This evaluates how closely the transformations produced by the generated rules match the expected code modifications, considering the structural similarity of the intended program semantics.

RQ4: To what extent do different LLMs generate complementary transformation rules, and how much overlap exists between them? This evaluates the overlap and uniqueness of rules across models to understand whether they capture similar or distinct transformation patterns.

5 Methodology

To answer these questions, we propose the overall approach shown in Figure 1. It considers different software evolution scenarios. Based on code diffs and examples, LLMs generate transformation rules to replay the code evolutions, which are then evaluated with several quality metrics. Throughout the study, we followed when relevant the proposed guidelines for conducting empirical software engineering studies involving LLMs from Baltes et al. [7].

5.1 Models

We consider three large language models with different architectures and capabilities: *GPT-5.4*, *GPT-oss-120B*, and *Llama3.1-8B*. *GPT-5.4* is a state-of-the-art proprietary model known for its strong reasoning ability and large context window, making it well-suited for complex code transformation and rule inference tasks. *GPT-oss-120B* is a recent open-weight model with 120 billion parameters and a context window of 131,072 tokens, designed to provide strong reasoning performance while remaining accessible for research experimentation. Finally, *Llama3.1-8B* is a smaller open-weight model with 8 billion parameters, included to provide a more lightweight

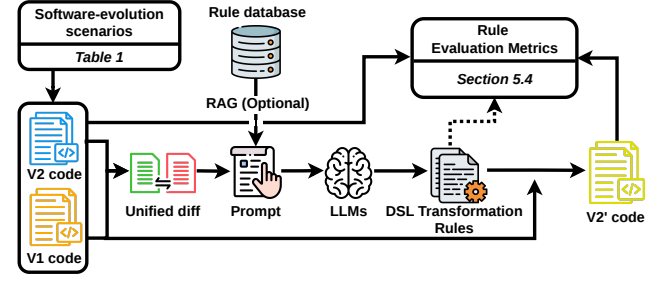


Figure 1: High-level overview of the pipeline for rule generation and application to V1 file.

and widely deployable model alternative. We selected these models to cover diverse ecosystems, parameter scales, and deployment constraints, while reflecting strong coding and instruction-following capabilities. They are widely used in SE/code-generation research.

For all three models, we set the temperature to 0 to reduce output variability and improve experiment reproducibility. Although this does not guarantee fully deterministic outputs across executions, it generally leads to more consistent transformation rules.

5.2 DSL Tools of Code Transformers

We selected three state-of-the-art DSLs for code transformations.

Comby: Comby [46] is a language-agnostic and lightweight DSL for matching or rewriting syntactic structures of a program’s parse tree using transformation rules. The key functionality of Comby rules is the use of template variables, which are holes in the match and rewrite templates that can be filled with code. Comby is also language-aware with an understanding of basic syntax of code, strings, and comment syntax.

For example, the Comby rule below replaces `fit_transform` with `transform` on test data to avoid re-fitting. The metavariables `:obj` and `:arg` capture the receiver object and its argument, enabling the transformation to apply generically.

```
[robust_scaler_test_transform]
match="scaled_x_test=:obj].fit_transform(:[arg])"
rewrite="scaled_x_test=:obj].transform(:[arg])"
```

Ast-Grep: Ast-Grep [6] is a structural search-and-replace tool that operates on the program’s abstract syntax tree (AST). Its match and rewrite patterns rely on metavariables that correspond to AST nodes and sub-expressions. Unlike Comby, which performs syntax-aware matching over textual code blocks, Ast-Grep enforces structurally valid matches at the AST level. As a result, Ast-Grep provides stricter and more precise matching, as each metavariable must correspond to a well-formed AST node.

For example, the Ast-Grep rule below casts the input to `float32` before calling `fit_transform`. The metavariables `$LHS`, `$OBJ`, and `$ARG` capture the assignment target, the object, and the input argument, enabling a generic transformation.

```
id: cast-affinity-float32
language: python
rule:
  pattern: $LHS = $OBJ.fit_transform($ARG)
  fix: $LHS = $OBJ.fit_transform($ARG.astype(np.float32))
```

GritQL: GritQL [17] is also a structural search-and-replace tool operating on AST using a unique syntax close to SQL. This DSL offers a precise and expressive way of writing patterns for safer code transformations. Consequently, it can be harder to learn for a new user as it is a custom language with a stiff learning path and can be overkill for a simple find-replace task.

For example, the GritQL rule below detects incorrect usage of a predict function. The metavariables \$obj, \$X, \$y, and \$expected capture the model, input data, unintended extra argument, and expected output, generalizing the transformation across contexts. The where clause further constrains the rule by requiring a metavariable to match a specific language construct before the rewrite is applied.

```
patterns:
- name: remove_second_predict_arg
  body: |
    engine marzano(0.1)
    language python
    `assert_array_equal($obj.predict($X, $y),
    $expected)` =>
    `assert_array_equal($obj.predict($X), $expected)`
  where $y <: python`None`
```

While these three DSLs use different syntaxes, they share common foundational concepts, their main differences lie in the level of abstraction at which they operate, i.e., either on concrete or abstract syntax. Comby relies on a template system with metavariables symbolized as `[mv_name]` capture strings. This provides strong language-agnosticism and ease of use. However, it comes with limited structural expressiveness. For example, it cannot distinguish between a variable and a function call when they share the same textual form, unless additional delimiters are present. In contrast, Ast-Grep and GritQL build on Tree-Sitter¹ parsers to provide structural awareness, though they differ in how transformations are expressed. Its expressiveness is enhanced by support for hierarchical constraints such as `inside`, `has`, and `follows`, which allow users to match nodes based on their relationships within the AST. GritQL, on the other hand, departs from rigid key-value structures and instead uses a more functional syntax, where the `"=>"` operator enables inline transformations. It also supports variable scoping, allowing information captured deep within nested structures to be reused at higher levels.

5.3 Datasets

This section details the six datasets used in our evaluation, grouped by type of software engineering task, for which we extracted human-written pairs of *before* and *after* code modification. They are summarised in Table 1. We selected them according to six criteria: 1) state-of-the-art and open-access datasets, 2) diverse SE field origin, 3) availability of code pairs (input/output), 4) transformation granularity/complexity (small to large edits), 5) diversity of programming languages, 6) human-written edits (non-generated).

(1) API Misuse *Galappaththi et al.* [14]: A dataset of 43 data-dependent API misuses collected from Stack Overflow posts and GitHub commits related to data science libraries.

(2) Program Repair *ManySStuBs4J* [23]: A corpus of simple statement-level Java bug fixes mined from open-source GitHub projects. It contains two variants: one from the top 100 Java Maven

Table 1: Summary of the datasets used in the evaluations.

Dataset	Language	Pairs	Typical Change Type
API Misuse			
Galappaththi et al.	Python	43	Statement-level fixes
Program Repair			
ManySStuBs4J	Java	528	Single-statement bug fixes
Defects4J	Java	427	Function-level bug fixes
BugsInPy	Python	559	Function-level bug fixes
API Migration			
PyMigBench	Python	605	Multi-statement changes
Language Version Migration			
JMigBench	Java	45	Method-level changes

projects and one from the top 1000 Java projects. For this experiment, we selected a random subset of 528 pairs due to limited time and computational resources. *Defects4J* [22]: A benchmark of 854 reproducible Java bugs from real-world open-source projects, including faulty and fixed versions along with test suites. For this experiment, we selected a random subset of 427 pairs due to limited time and computational resources. *BugsInPy* [48]: A benchmark of 559 real-world Python bugs from 17 projects, inspired by Defects4J, designed for controlled debugging and repair studies.

(3) API Migration *PyMigBench* [19]: A benchmark of Python library migrations with 3,096 migration-related code changes from 335 migrations between 141 analogous library pairs. For our experiment, it represents 605 pairs of code.

(4) Language Version Migration *JMigBench* [5]: A benchmark suite and evaluation pipeline designed to assess large language models (LLMs) in the task of migrating Java functions from Java 8 to Java 11. It contains 45 pairs of Java 8/Java 11 migrations.

5.4 Evaluation Metrics

Soundness of the generated DSL transformation rules is assessed through metrics proposed by Ramos et al. [41] and Ketkar et al. [26].

Failures (F): Number of code edits for which the generated transformation rule could not be successfully applied or evaluated. This includes cases where (i) the generated rule is syntactically invalid, (ii) the DSL engine fails during execution, (iii) the produced migrated code contains syntax errors, or (iv) metric computation (e.g., AST parsing) fails.

Rule Applicability Rate (RA%): The Rule Applicability Rate is the percentage of generated transformation rules that are syntactically valid and executable by the corresponding DSL engine.

Number of Generated Tokens (NT): Measures the number of tokens generated by the LLM when producing a set of transformation rules for a given code edit.

Completeness of the generated DSL transformation rules is assessed using metrics for characterizing efficiency and verbosity.

Number of Rules (#R): Mean number of rules in a given DSL configuration file for one code edit.

Size of Rule (SR): Mean rule size for each dataset, as the length of the character sequence.

Metavariables Token Ratio (MTR): The MTR measures how much of the match pattern is hard-coded based on the number of

¹<https://tree-sitter.github.io/tree-sitter/>

metavariables tokens used in a single pattern.

$$\text{MTR} = \frac{\# \text{ metavariable tokens}}{\# \text{ total tokens in the pattern}}, \quad 0 \leq \text{MTR} \leq 1.$$

An MTR of 0 indicates that no metavariables are used (fully hard-coded pattern), while an MTR of 1 indicates that every token in the pattern is a metavariable.

Finally, the following syntactic metrics are widely used for evaluating code similarity and transformation correctness.

Tree edit Distance (TD): The Tree edit Distance calculates the shortest sequence of edit operations (Insert, Delete, Replace) to transform tree T_1 to tree T_2 . In our case, T_1 is the human-written reference code, while T_2 corresponds to the code produced by applying the generated transformation rule.

Exact Match (EM%): Exact Match is a binary metric for a syntax-level perfect match between the human-written reference code and the transformation rule-generated code.

AST Match (AM%): AST Match is a binary metric for an AST-level perfect match between the human-written reference code and the transformation rule-generated code.

5.5 Evaluation Process

To evaluate each code change of the 6 datasets described above, we provide the LLM with a diff from a pair of code snippets representing the before and after versions of a modification, as shown in Figure 1. A diff allows a developer to look at the two files side by side and see exactly what differentiates them, such as new lines of code that have been added, if variable names have been changed, or if any lines of code have been removed. Using this input-output example, the model is prompted to synthesize one or more transformation rules in a chosen DSL: Comby, Ast-Grep, or GritQL. Since LLMs have limited exposure to these DSLs during training, we augment the prompt with 8 examples written in the target DSL.

For Comby, we further enhance the prompt using a Retrieval-Augmented Generation (RAG) strategy. Leveraging MELT [41], a tool for inferring transformation rules from pull requests, we collected over 1,700 Comby rules. At inference time, we retrieve the 8 most relevant examples to include in the prompt, providing higher-quality and more targeted guidance.

In contrast, we did not identify any equivalent large-scale rule repositories or RAG mechanisms for Ast-Grep and GritQL. Instead, their prompts include a fixed set of 8 examples derived from the DSL official documentation. As a result, the RAG-based evaluation is conducted only for Comby, enabling us to assess the impact of retrieval augmentation separately from standard in-context learning.

As baseline, we consider an anti-unification algorithm [26] that takes as input the pairs of code examples from our six datasets and identifies the most general structure shared by two expressions; in our case, the match pattern and the corresponding rewrite pattern.

We execute the transformation rules on the input/before version to obtain a transformed version of the code representing the output of the generated rule. Once the transformation rule is generated, we apply it to the original before-code snippet to produce the transformed code. Next, we compare the generated code with the ground-truth version in order to compute the evaluation metrics described earlier, including syntactic and semantic equivalence.

The evaluation process first assesses the *applicability* of the generated rules, verifying that they conform to the syntax of the target DSL and can be successfully compiled and executed by the corresponding DSL engine. We also report the number of consumed tokens as a measure of computational cost. We then evaluate the ability of the LLMs to leverage the expressiveness of the DSLs, particularly through the use of metavariable placeholders that enable rule generalization. We also characterize the number and size of the generated rules. Generalizability is also assessed using a reuse score, computed by applying the rules derived from correct transformations to all instances in each dataset and counting the number of additional matches, i.e., how many times a rule matches.

After that, we also measure the correctness of the produced transformations by comparing the generated code with the ground-truth version. This comparison includes both *exact syntax matching* (considering formatting aspects such as spaces and indentation) and *AST-level matching*, which relies on parsing the code into abstract syntax trees to assess structural equivalence. In addition, we compute the tree edit distance to quantify the structural differences between the generated and reference code. We further strengthen semantic evaluation by leveraging the Defects4J and BugsInPy test suites to assess alignment between ground-truth and patched code.

Additionally, to better understand and contextualize our results, we conducted a qualitative analysis for the first three research questions (RQs). Specifically, for each dataset and each DSL, we randomly selected a subset of 72 pairs while applying additional criteria tailored to each RQ. The random selection was performed by choosing one pair for each combination of benchmark $\times$ DSL $\times$ model ($6 \times 4 \times 3 = 72$). However, for RQ1, some DSL-model combinations exhibited 100% RA, reducing the total number of selected pairs to 60. For RQ1, we focused on cases where the generated rules were not applicable, i.e., violated the DSL syntax, or could not be executed by the transformation engine. For RQ2, we selected cases where the generated rules were valid and executable, enabling us to analyze how LLMs leverage DSL expressiveness, particularly through metavariable usage and rule generalization. For RQ3, we considered cases where the generated transformations were applicable but did not exactly match the ground truth, allowing us to analyze syntax-level discrepancies, such as incorrect, missing, or unneeded edits, and to identify notable patterns and outlier results.

Finally, to assess whether observed differences between models are statistically significant, we apply McNemar’s test for paired binary metrics (RA, EM, AM) and Wilcoxon signed-rank tests for the continuous metric (TD), with Benjamini-Hochberg correction for multiple comparisons across the (6×4) 24 dataset-DSL configurations ($\alpha = 0.05$). For aggregate-level metrics (#R, SR, MTR), we use a sign test. Full results and scripts in the replication package.

6 Results

6.1 RQ1: Effectiveness

Table 2 reports the soundness of transformation rules generated by GPT-oss-120B, GPT-5.4, and Llama3.1-8B across all datasets and DSLs. We report on three metrics: the number of Failures (F), Rule Applicability (RA%), and the Number of Tokens (NT). Values in bold correspond to the top 1% of results for each tool.

Table 2: Soundness of generated transformation rules per dataset, tool, and LLM (grouped by scenario)

Tool	GPT-oss-120B			GPT5.4			Llama3.1-8B			Anti-Uni		
	F	RA	NT	F	RA	NT	F	RA	NT	F	RA	NT
API Misuse — Galappaththi et al. (n=43)												
Comby (R)	3	93.0%	192	2	95.3%	208	18	58.1%	166	1	97.6%	283
Comby (NR)	0	100.0%	185	0	100.0%	202	8	81.4%	264	—	—	—
Ast-Grep	17	60.5%	170	3	93.0%	228	25	40.5%	342	19	54.8%	254
GritQL	2	95.3%	253	0	100.0%	215	1	97.7%	293	0	100.0%	279
Program Repair — ManySSuBs4J (n=528)												
Comby (R)	9	98.9%	90	5	99.2%	225	335	36.6%	212	2	100.0%	460
Comby (NR)	12	97.9%	88	88	83.5%	171	128	76.1%	995	—	—	—
Ast-Grep	219	58.7%	108	77	85.6%	108	448	15.5%	557	211	54.1%	451
GritQL	5	99.4%	148	5	99.4%	221	4	99.8%	242	2	100.0%	480
Program Repair — Defects4J (n=427)												
Comby (R)	34	92.3%	160	4	99.1%	211	264	38.0%	124	0	100.0%	82
Comby (NR)	24	94.8%	134	89	79.1%	173	125	71.1%	239	—	—	—
Ast-Grep	203	53.8%	152	92	78.4%	179	30	95.1%	262	299	29.8%	79
GritQL	6	98.6%	201	0	100.0%	240	0	100.0%	174	0	100.0%	100
Program Repair — BugsInPy (n=559)												
Comby (R)	36	93.6%	158	34	93.9%	213	429	23.3%	159	16	97.1%	156
Comby (NR)	18	96.8%	149	178	68.2%	169	134	76.0%	201	—	—	—
Ast-Grep	305	45.4%	140	97	82.6%	187	404	27.7%	336	401	28.0%	149
GritQL	2	99.6%	212	0	100.0%	248	1	99.8%	185	0	100.0%	172
API Migration — PyMigBench (n=605)												
Comby (R)	55	90.9%	293	64	89.4%	394	367	39.3%	183	42	93.0%	421
Comby (NR)	40	93.4%	241	90	85.1%	323	190	68.6%	768	—	—	—
Ast-Grep	311	48.6%	183	114	81.2%	349	462	23.6%	399	565	6.5%	617
GritQL	4	99.3%	365	1	99.8%	471	4	99.3%	255	0	100.0%	666
Language Version Migration — JMigBench (n=45)												
Comby (R)	0	100.0%	113	0	100.0%	141	20	55.6%	112	0	100.0%	141
Comby (NR)	0	100.0%	113	1	97.8%	146	6	86.7%	173	—	—	—
Ast-Grep	19	57.8%	119	12	73.3%	165	32	28.9%	198	34	24.4%	133
GritQL	0	100.0%	139	0	100.0%	161	1	100.0%	161	0	100.0%	156

F: Failures; RA: Rule Applicability rate; NT: Number of Generated Tokens; R: RAG; NR: no RAG.

Overall, GPT-5.4 achieves the highest Rule Applicability (RA), with values ranging from 68.2% to 100%, and consistently records the lowest number of failures when using Ast-Grep, GritQL, and Comby with RAG. Across these settings, RA frequently exceeds 90%, reaching up to 100% in several cases. GritQL stands out as the most reliable DSL, achieving the best RA across all models and datasets, often reaching near-perfect or perfect soundness. In contrast, GPT-oss-120B performs best with Comby without RAG, producing fewer failures than other models while maintaining competitive RA, ranging from 93.6% to 100%. In this setting, GPT-5.4 exhibits a notable drop in performance, reaching as low as 68.2%. Additionally, GPT-oss-120B consistently generates more concise outputs, with token counts typically 20–40% lower than GPT-5.4.

As for anti-unification, we observe a high Rule Applicability (RA) for Comby and GritQL (93%-100%), reflecting the deterministic nature of the algorithm. In contrast, RA is much lower for Ast-Grep (6.5%-54.8%) because the generalized match expressions produced often fail to satisfy its requirement that patterns should correspond to valid AST nodes, making many generated rules inapplicable.

Statistical significance. McNemar tests confirm these trends: GPT-5.4 is significantly better than GPT-oss-120B in 7 of 24 configurations ($p < 0.05$), mostly on Ast-Grep, while GPT-oss-120B wins in 4, all on Comby without RAG; both significantly outperform Llama3.1-8B in at least 16 of 24.

While GPT-5.4 delivers the strongest overall performance at the cost of increased token usage. Llama3.1-8B further amplifies this trend, often producing the largest outputs (e.g., exceeding 300 tokens) while also yielding lower RA and higher failure rates.

A deeper qualitative analysis of the random sample of 60 cases reveals that GPT-5.4 generates more tokens primarily because it handles more complex transformations, particularly in datasets such as Defects4J, BugsInPy, and PyMigBench, which involve larger code changes. Indeed, as shown in Table 1, these benchmarks involve function-level or multi-statement bug fixes, requiring larger and more complex transformation rules that GPT-oss-120B and Llama3.1-8B often fail to generate. In contrast, for simpler transformations, GPT-5.4 produces a number of tokens comparable to GPT-oss-120B. Moreover, most failures stem from formatting issues in the generated configurations, such as the inclusion of generated comments or extra formatting artifacts (e.g., inserting natural language explanations like *// this rule replaces X with Y*, which break the DSL syntax). We also observe frequent errors related to missing rule separators in Ast-Grep, where multiple rules are written consecutively without proper delimiters, leading to parsing failures. Finally, we identify invalid patterns involving multiple matching nodes, such as attempting to match two unrelated AST nodes within a single rule, which is not supported by Ast-Grep and GritQL.

RQ1 insights and findings: GPT-5.4 achieves the best overall soundness, with the highest rule applicability (68.2%–100%) and the fewest failures across most DSLs (GritQL being the most reliable DSL). However, it generates significantly more tokens, largely due to its ability to handle more complex transformations. GPT-oss-120B achieves strong performance specifically on Comby without RAG (93.6%–100%), while Llama3.1-8B exhibits lower RA and higher failure rates. Anti-unification achieves near-perfect RA on Comby and GritQL but low RA on Ast-Grep due to its stricter AST node constraints. Overall, strong LLMs can generate executable transformation rules at a high rate. Remaining failures stem from DSL syntax and conventions rather than incorrect match and rewrite code.

6.2 RQ2: Rule Quality

Table 3 summarizes the structural completeness of transformation rules generated by GPT-oss-120B, GPT-5.4 and Llama3.1-8B across all datasets and DSLs using the metrics number of Rules (#R), Size of Rules (SR), and the Metavariable Token Ratio (MTR%).

Overall, a clear trend emerges across all benchmarks. GPT-5.4 consistently produces larger rules, with the highest SR values (e.g., up to 285.1 tokens on JMigBench), while generating fewer rules (#R typically between 1.1 and 3.7) and using fewer metavariables (MTR mostly between 3% and 13%). In contrast, GPT-oss-120B generates more compact rules, with slightly higher #R (typically 1.6 to 4.7) and significantly higher MTR (ranging from 7% to 20%). This difference is particularly visible in program repair benchmarks such as Defects4J and BugsInPy, where GPT-oss-120B reaches MTR values up to 19–20% with #R around 2.0, while GPT-5.4 produces fewer rules (#R = 1.6–2.3) with lower MTR (as low as 3–11%). Similarly, in PyMigBench, GPT-5.4 generates larger rules (SR = 242.1) with low MTR (3%), whereas GPT-oss-120B maintains higher values (MTR = 7–14%) with slightly higher #R (= 3.3–4.7). This highlights a

Table 3: Soundness of generated transformation rules per Dataset, Tool, and LLM (grouped by scenario)

Tool	GPT-oss-120B			GPT5.4			Llama3.1-8B			Anti-Uni		
	NR	#R	MTR	NR	#R	MTR	NR	#R	MTR	NR	#R	MTR
API Misuse – Galappaththi et al. (n=43)												
Comby (R)	3.4	57.3	7	1.6	162.8	10	3.9	44.5	14	1.9	113.7	54
Comby (NR)	3.4	53.9	7	2.1	115.3	6	7.3	41.4	3	–	–	–
Ast-Grep	3.1	39.6	17	3.6	51.0	9	4.3	48.3	15	1.9	96.7	53
GritQL	2.8	69.6	7	1.9	93.8	13	4.2	46.9	0	1.9	96.7	37
Program Repair – ManyStuBs4J (n=528)												
Comby (R)	1.8	63.4	18	1.6	167.6	10	6.1	48.9	4	2.4	165.5	62
Comby (NR)	1.9	63.5	17	1.9	73.8	6	29.4	43.7	18	–	–	–
Ast-Grep	2.0	55.1	20	1.8	58.6	19	11.1	60.9	6	2.4	156.7	62
GritQL	1.9	65.9	14	1.6	79.5	16	3.5	54.3	1	2.4	156.6	44
Program Repair – Defects4J (n=427)												
Comby (R)	1.8	96.2	13	1.6	183.8	7	3.9	45.2	4	1.2	57.2	56
Comby (NR)	1.9	67.5	16	1.6	160.1	3	9.5	27.1	1	–	–	–
Ast-Grep	2.1	51.6	19	2.0	78.1	14	8.0	56.6	5	1.2	49.3	55
GritQL	1.9	78.5	15	1.7	132.6	11	2.4	48.7	0	1.2	49.3	39
Program Repair – BugsInPy (n=559)												
Comby (R)	2.1	80.8	15	1.7	186.2	4	4.4	42.4	6	1.7	74.1	61
Comby (NR)	2.1	75.9	13	1.7	147.4	3	5.8	43.6	2	–	–	–
Ast-Grep	2.0	54.8	18	2.3	76.7	11	6.7	51.3	8	1.7	63.3	60
GritQL	2.1	77.8	13	1.9	131.3	8	2.8	42.7	1	1.7	59.1	43
API Migration – PyMigBench (n=605)												
Comby (R)	4.7	90.4	7	3.0	242.1	3	5.9	39.6	4	4.8	96.2	49
Comby (NR)	4.6	68.8	9	3.6	160.4	3	24.5	35.4	1	–	–	–
Ast-Grep	3.3	36.3	14	5.4	70.2	8	8.0	52.5	4	5.5	128.5	47
GritQL	4.4	73.0	7	3.7	145.5	7	4.1	39.1	0	5.5	124.1	34
Language Version Migration – JmigBench (n=45)												
Comby (R)	1.8	115.8	17	1.1	278.6	13	3.2	45.8	10	1.4	144.1	31
Comby (NR)	1.6	135.1	14	1.1	285.1	7	5.7	41.9	7	–	–	–
Ast-Grep	2.0	60.0	20	2.9	72.1	22	4.7	47.2	11	1.4	135.0	29
GritQL	1.3	149.8	16	1.2	204.7	13	2.6	53.2	0	1.4	135.0	20

#R: number of Rules; SR: Size of Rules; MTR: Metavariables Token Ratio; R: RAG; NR: no RAG.

trade-off in rule construction strategies. GPT-5.4 tends to generate monolithic and explicit transformation rules, favoring larger patterns with limited abstraction. Conversely, GPT-oss-120B produces more modular and generalized rules, leveraging metavariables more extensively. Llama3.1-8B follows a similar pattern to GPT-oss-120B in terms of rule size but exhibits lower structural completeness overall, with fewer rules and limited use of metavariables.

Anti-unification consistently achieves the highest MTR across all instances because it transforms every common token between the match and rewrite patterns into metavariables. As a result, it reaches a maximum MTR of 62, compared to only 20 for all LLM-generated rules. This over-generalization leads to overly generic and unreadable rules that can match nearly any code fragment, highlighting the limitations of anti-unification when task-specific constraints are not implemented.

Statistical significance. A sign test across the 24 configurations confirms that GPT-5.4 produces larger rules (higher SR in 24/24, $p < 10^{-6}$) with fewer metavariables (lower MTR in 19/24, $p = 0.007$) than GPT-oss-120B, while Llama3.1-8B generates more rules than both in 22/24 ($p < 10^{-4}$).

Moreover, our qualitative analysis on the random sample of 72 cases provides further insight into these trends. For complex transformations, particularly in datasets such as Defects4J, BugsInPy, and PyMigBench, GPT-5.4 demonstrates a stronger ability to leverage

Table 4: Reuse statistics per dataset, DSL, and LLM (grouped by scenario)

DSL	GPT-oss-120B			GPT5.4			Llama3.1-8B		
	Med.	2+	20+	Med.	2+	20+	Med.	2+	20+
API Misuse – Galappaththi et al. (n=43)									
Comby (R)	1.0	10	1	1.0	5	0	1.0	2	0
Comby (NR)	1.0	11	3	1.0	3	1	1.0	3	2
Ast-Grep	1.5	6	2	3.0	15	4	3.0	2	0
GritQL	1.0	6	2	1.0	5	2	1.0	4	2
Program Repair – ManyStuBs4J (n=528)									
Comby (R)	1.0	84	32	1.0	63	11	0.0	35	15
Comby (NR)	1.0	127	35	1.0	91	17	0.0	793	100
Ast-Grep	2.0	151	42	1.0	220	64	1.0	21	6
GritQL	1.0	166	44	1.0	98	48	0.0	59	14
Program Repair – Defects4J (n=427)									
Comby (R)	1.0	33	15	1.0	10	3	1.0	5	5
Comby (NR)	1.0	45	18	1.0	12	7	1.0	16	8
Ast-Grep	2.0	42	15	1.0	63	21	0.0	1	1
GritQL	3.0	51	4	3.0	212	8	3.0	45	6
Program Repair – BugsInPy (n=559)									
Comby (R)	1.0	56	15	1.0	19	3	1.0	9	5
Comby (NR)	1.0	56	14	1.0	34	4	1.0	31	5
Ast-Grep	2.5	40	6	2.0	118	17	5.0	6	1
GritQL	1.0	29	3	1.0	67	6	1.0	15	1
API Migration – PyMigBench (n=605)									
Comby (R)	4.0	281	102	2.0	176	36	3.0	40	5
Comby (NR)	6.0	288	111	3.0	242	41	6.0	273	44
Ast-Grep	13.5	115	51	10.0	245	112	33.5	10	8
GritQL	12.0	379	70	12.0	478	75	10.0	180	31
Language Version Migration – JmigBench (n=45)									
Comby (R)	1.0	6	0	1.0	1	0	1.0	3	0
Comby (NR)	1.0	1	0	1.0	0	0	1.0	2	0
Ast-Grep	1.5	1	0	2.0	7	1	0.0	0	0
GritQL	91.0	12	12	91.0	16	16	91.0	2	2

Med.: Median matches per rule; k+: number of rules with more than k matches; R: RAG; NR: no RAG.

DSL expressiveness. For instance, it frequently uses metavariables such as `$$$BODY` to capture entire sequences of AST nodes, enabling it to rewrite or remove complete function or method bodies in a single rule. Overall, it produces semantically meaningful metavariable names, greatly improving readability for the generated rules.

In contrast, smaller models such as GPT-oss-120B and Llama3.1-8B tend to avoid large or complex transformations, focusing instead on simpler, localized edits. This often results in partial rules that fail to capture substantial code modifications. Additionally, GPT-oss-120B frequently introduces metavariables such as `:[indent]` to explicitly match whitespace, compensating for the lack of native indentation handling in Comby. While this allows for exact matching it significantly increases rule verbosity and reduces generalization.

Generalizability beyond a single example. Table 4 presents the results of the reusability experiments. For GPT-oss-120B, GritQL produced the largest number of reusable rules, with 166 rules matching at least twice and 44 matching more than 20 times, closely followed by Ast-Grep (151 and 42, respectively). GPT5.4 exhibits a similar trend, with Ast-Grep yielding 220 rules reused at least twice and 64 reused more than 20 times, compared with 98 and 48 for GritQL. In contrast, Comby generated fewer highly reusable rules. Llama3.1-8B consistently achieves the lowest reuse across most DSLs, typically with fewer than 60 rules reused at least twice. The main exception is Comby without RAG, which generated 793 rules

reused at least twice and 100 reused more than 20 times. However, these rules have a median reuse of 0, indicating that this behavior is driven by a relatively small number of highly repetitive transformations. A manual inspection revealed many overly generic rules, explaining why this configuration achieves lower AM/EM scores despite producing a large number of reusable matches.

In addition, we provide more detailed illustrations of the reuse score for each dataset by plotting the most frequently matched rules for each DSL and model on a logarithmic scale. These figures better show the high reuse score in the first quartile and the outliers with a very high number of matches, corresponding to recurrent transformations such as `assert(:[var]) → Assert.assert(:[var])`. They are available in the [replication package \[link\]](#).

RQ₂ insights and findings: GPT-5.4 generates larger, more explicit rules (high SR, low MTR, low #R), while GPT-oss-120B produces more compact and generalized rules (higher MTR and #R). In contrast, the anti-unification baseline achieves the highest MTR by over-generalizing transformation rules. Smaller models focus on simpler changes and often miss larger transformations. The qualitative analysis reflects a trade-off between abstraction and coverage. More abstract, metavariable-heavy rules are not automatically better, and explicitness is sometimes the mechanism enabling complex transformations to be captured at all. The reuse score analysis further shows that many LLM-generated rules generalize beyond a single example.

6.3 RQ3: Accuracy

Table 5 summarizes the syntactic and semantic alignment of transformations made through the rules generated by GPT-oss-120B, GPT-5.4, and Llama3.1-8B across six benchmarks, evaluated in terms of semantic and syntactic similarity using the metrics of Exact Match (EM), AST Match (AM), and Tree Distance (TD). Values highlighted in bold correspond to the top 1% of results for each tool. Overall, GPT-5.4 achieves the best performance across all metrics, consistently outperforming other models on most benchmarks. It reaches the highest TD scores (up to 98.1%) and strong EM/AM results, particularly on complex datasets such as Defects4J (up to 56.0% AM) and BugsInPy (up to 54.1% EM/AM). These results indicate that GPT-5.4 generates transformations that are both structurally and semantically closest to the ground truth. GPT-oss-120B performs competitively, especially with Comby, both with and without RAG. For instance, it achieves up to 67.5% EM/AM on API misuse with Comby + RAG and maintains strong performance on JMigBench (up to 68.9% EM). However, its performance drops on more complex transformations, where it struggles to fully capture large edits. In contrast, Llama3.1-8B consistently exhibits low performance across all benchmarks and DSLs, with EM/AM scores often below 30% and dropping as low as 5.2% on Defects4J, highlighting its limited ability to generate accurate transformations.

The anti-unification algorithm performs significantly worse, achieving at most 36.4% EM/AM in the best-case scenario and only 1.2–2.4% EM/AM on benchmarks such as Defects4J. These results primarily come from the algorithm’s simplistic, non-contextual understanding of code, and its inability to handle code additions.

Table 5: Syntactic alignment of generated code per dataset, tool, and LLM (grouped by scenario)

Tool	GPT-oss-120B			GPT-5.4			Llama3.1-8B			Anti-Uni		
	TD	EM	AM	TD	EM	AM	TD	EM	AM	TD	EM	AM
API Misuse — Galappaththi et al. (n=43)												
Comby (R)	82.8	67.5	67.5	88.1	63.4	63.4	80.1	24.0	24.0	71.5	0.0	0.0
Comby (NR)	79.7	53.5	53.5	82.1	48.8	48.8	72.9	25.7	25.7	—	—	—
Ast-Grep	85.6	53.8	53.8	88.6	67.5	67.5	83.1	17.6	17.6	65.7	17.4	17.4
GritQL	86.3	58.5	58.5	88.1	46.5	46.5	77.6	23.8	23.8	73.4	9.5	9.5
Program Repair — ManySStuBs4J (n=528)												
Comby (R)	96.7	45.1	45.1	97.0	57.6	57.6	83.6	11.9	11.9	96.3	3.9	3.9
Comby (NR)	97.3	45.7	45.7	97.7	56.4	56.4	89.5	18.2	18.2	—	—	—
Ast-Grep	90.8	52.1	52.1	94.3	62.5	62.5	68.6	23.8	23.8	90.8	3.2	3.2
GritQL	98.5	46.1	46.1	98.1	56.4	56.4	93.9	5.3	5.3	95.3	1.5	1.5
Program Repair — Defects4J (n=427)												
Comby (R)	92.5	31.6	31.6	94.1	33.6	33.6	84.3	8.0	8.0	91.0	1.2	1.2
Comby (NR)	89.4	28.1	28.1	94.0	35.0	35.0	84.0	9.3	9.3	—	—	—
Ast-Grep	89.5	41.3	41.3	90.0	49.4	49.4	69.7	5.6	5.6	86.9	2.4	2.4
GritQL	92.2	7.4	7.4	93.6	35.4	35.4	90.7	8.5	8.5	89.4	1.4	1.4
Program Repair — BugsInPy (n=493)												
Comby (R)	74.3	22.4	22.4	54.9	11.8	11.8	71.8	6.9	6.9	87.2	3.0	3.0
Comby (NR)	67.3	19.0	19.0	74.5	28.9	28.9	78.6	9.6	9.6	—	—	—
Ast-Grep	84.7	31.5	31.5	93.7	54.1	54.1	68.3	5.8	5.8	78.1	15.4	15.4
GritQL	84.0	18.0	18.0	92.9	33.5	33.5	86.0	9.9	9.9	85.6	5.0	5.0
API Migration — PyMigBench (n=661)												
Comby (R)	69.2	43.3	43.3	69.6	45.7	45.7	67.6	12.2	12.2	61.1	15.8	15.8
Comby (NR)	66.0	41.8	41.8	69.5	49.9	49.9	72.1	25.5	25.5	—	—	—
Ast-Grep	88.3	46.3	46.3	89.3	57.6	57.6	66.9	8.4	8.4	87.2	0.0	0.0
GritQL	78.2	35.3	35.3	80.8	40.1	40.1	76.2	17.3	17.3	75.5	0.0	0.0
Language Version Migration — JMigBench (n=45)												
Comby (R)	91.0	68.9	68.9	87.0	60.0	60.0	67.8	20.0	20.0	76.2	8.9	8.9
Comby (NR)	88.8	62.2	62.2	81.0	45.5	45.5	58.0	23.1	23.1	—	—	—
Ast-Grep	81.6	30.8	30.8	87.7	51.5	51.5	55.7	0.0	0.0	98.8	36.4	36.4
GritQL	76.9	22.2	22.2	76.9	24.4	24.4	74.8	4.5	4.5	74.5	8.9	8.9

TD: Tree Distance; EM/AM: Exact/AST Match rate; R: RAG; NR: no RAG.

Statistical significance. McNemar tests confirm that GPT-5.4 significantly outperforms GPT-oss-120B on AST Match in 12 of 24 configurations vs. only 2 wins for GPT-oss-120B, and outperforms Llama3.1-8B in all 24 (all $p < 0.05$); Wilcoxon tests on tree distance yield consistent results (13/24 significant).

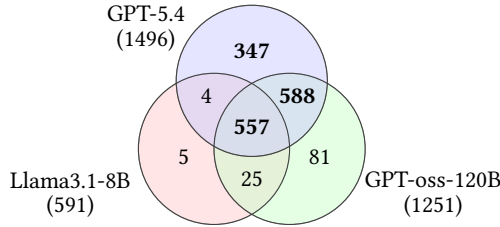
Finally, our qualitative analysis on the random sample of 72 cases provides deeper insights by categorizing errors into edits, deletions, and additions. We observe that Llama3.1-8B frequently produces missing edits and incorrect transformations, and often generates duplicate rules with identical match and rewrite patterns, indicating limited ability to consolidate transformations and generalize across similar cases. This behavior contributes to its low overall correctness. GPT-oss-120B performs better but still exhibits missing edits, particularly in cases where the modification does not correspond to a complete AST node sequence. In such scenarios, the model should generalize the transformation by expanding the matching pattern to capture the full intended change. However, it often fails to do so, producing rules that mirror only a narrowly scoped pattern rather than a generalized one, leading to incomplete edits. In contrast, GPT-5.4 produces a higher proportion of correct edits, especially for complex and non-local transformations.

We also evaluated the semantic equivalence of transformations that did not result in an AST match in our sample of 72 rules. On this subset, 36 transformations failed to produce an AST match, with only one instance preserving semantic equivalence. Overall, none

Table 6: Test validation on Defects4J and BugsInPy test suites

Dataset	DSL	GPT-oss-120B			GPT5.4			Llama3.1-8B		
		A	P	SR	A	P	SR	A	P	SR
Defects4J	Comby (R)	217	6	2.8	243	7	2.9	93	2	2.2
	Comby (NR)	227	5	2.2	24	0	0.0	183	2	1.1
	Ast-Grep	89	6	6.7	40	0	0.0	105	5	4.8
	GritQL	269	8	3.0	204	6	2.9	98	0	0.0
BugsInPy	Comby (R)	405	11	2.7	462	2	0.4	120	6	5.0
	Comby (NR)	437	18	4.1	270	8	3.0	383	23	6.0
	Ast-Grep	174	28	16.1	212	38	17.9	146	4	2.7
	GritQL	456	16	3.5	371	17	4.6	504	12	2.4

A/P: Attempted/Successful Repairs; SR: Success Rate (%); R: RAG; NR: no RAG.

**Figure 2: Venn diagram of applicable transformation rules.**

of the transformations that resulted in differing edits achieved full semantic equivalence. This likely because the goal was to regenerate the same V2 of the code, i.e., exact syntactic equivalence.

The single case of preserved semantic equivalence involved a trivial pattern with a redundant variable declaration (e.g., `int x = 0; int x = 0; return x`). However, such instances do not constitute meaningful or practically useful transformations. Furthermore, some transformations may preserve semantic equivalence at a localized level (e.g., method, import, or class), while still producing code that is invalid or inconsistent at the file level.

Semantic equivalence on Defects4J and BugsInPy. Table 6 shows that only a small fraction of these transformations resulted in successful repairs when there is no AST match. On average, GPT-oss-120B has the largest number of repair attempts on both Defects4J (200.5 attempted repairs per DSL) and BugsInPy (368.0), followed by GPT5.4 (127.8 and 328.8, respectively) and Llama3.1-8B (119.8 and 288.2, respectively). On Defects4J, the best outcome was achieved by GPT-oss-120B with GritQL, with 8 successful repairs (3.0% success rate), while Ast-Grep attained the highest success rate of 6.7% (6/89). On BugsInPy, substantially more successful repairs were observed, with Ast-Grep and GPT5.4 achieving the highest success rate of 17.9% (38/212), and Comby (#R) with Llama3.1-8B producing the most successful repairs (23/383). However, these results could be significantly improved with a feedback loop [33].

RQ3 insights and findings: GPT-5.4 achieves the best overall semantic correctness, particularly on complex transformations. GPT-oss-120B performs well in simpler settings, especially with Comby, but struggles with large edits. Llama3.1-8B shows consistently low performance, with many missing or incorrect transformations. Qualitative analysis shows that GPT-5.4 generates more correct edits in complex scenarios, while smaller

models often miss or incorrectly apply transformations. Test-based validation further confirms that AST matching is a necessary condition for successful test passing, although only a subset of non AST-matching transformations repaired the bugs. Fully reproducing the *intended* transformation remains the main challenge: even GPT-5.4 sometimes produces rules that are too specific to generalize or too partial to capture all required changes, and the gap between models increases on complex and non-local edits (e.g., multi-statement migrations).

6.4 RQ4: Complementarity

Figure 2 presents the overlap of correctly inferred transformation rules across the three models in terms of AST match. Overall, GPT-based models substantially outperform Llama3.1-8B, which contributes very few unique correct transformations (5 cases). In contrast, GPT-5.4 identifies the largest number of unique rules (347), followed by GPT-oss-120B (81), highlighting their stronger capability to generalize beyond shared cases.

A large portion of rules is shared between GPT-5.4 and GPT-oss-120B (557+588), indicating that both models capture similar transformation patterns with different efficiency and verbosity, as discussed previously. Our qualitative analysis shows that the set of rules on which all three models overlap corresponds to typically low-complexity cases, often reducible to one-line transformations.

Importantly, GPT-5.4 demonstrates a clear advantage in handling more complex transformations, as evidenced by its substantial number of unique solutions and its dominance in pairwise overlaps. This suggests that GPT-5.4 is better able to infer nuanced or structurally complex rules that are not captured by smaller models. In contrast, Llama3.1-8B is limited to simple patterns, such as one-line transformations, but remains a viable option for local deployment.

RQ4 insights and findings: GPT-5.4 is in a category of its own, with 347 unique correct rules that no other model produces, compared to 81 for GPT-oss-120B and only 5 for Llama3.1-8B. Our qualitative analysis confirms that only GPT-5.4 consistently solves complex transformations, while Llama3.1-8B is more limited but very effective for simple, one-line changes.

6.5 Threats to Validity

Prompt and LLM sensitivity. Minor variations in prompt phrasing can significantly affect model outputs, influencing generated rule structure and DSL behavior, reducing reproducibility. To mitigate this, we standardize prompt templates across all experiments. Additionally, the temperature parameter can impact output variability and correctness. We mitigate this by fixing it to zero. Finally, the number and selection of pattern examples, particularly when using RAG for Comby, may bias a model toward specific rewriting strategies. To mitigate this, we use a fixed number of examples and consistent retrieval strategies across experiments. These effects are compounded by differences in DSL expressiveness: some DSLs lack features such as guards or fine-grained node constraints, while others, like Ast-Grep, offer more advanced mechanisms for precisely targeting AST nodes, making direct comparisons challenging. To mitigate this, we evaluate three DSLs and constrain the LLM to use only the core match and rewrite functionalities of each DSL.

Evaluation metric limitations. The chosen evaluation metrics (e.g., rule application rate or syntactic correctness) may not fully capture semantic correctness. For instance, a transformation might fail syntactic checks (exact match or AST match) while still preserving semantic behavior when evaluated against a test suite. We investigated this limitation on the two benchmarks with available test suites, but the absence of systematically available executable oracles across all considered benchmarks prevents extending this evaluation more broadly, which is beyond the scope of this work.

Scalability and complexity limits. The evaluation may suffer from bias on small or localized transformations. Performance could degrade on larger codebase, deeply nested structures, or transformations requiring global context. To mitigate this, we include datasets containing a diverse range of transformation complexities which encompass both local and non-local changes.

Programming language bias. The evaluation is limited to Java and Python, two very popular languages representing two distinct syntactic paradigms (bracket-based and indentation-sensitive), but not the full spectrum of programming languages.

Dataset leakage. There is a risk that models have been exposed to similar patterns or rules during training, potentially inflating performance. However, this threat is difficult to fully eliminate, as training data of proprietary LLMs is not publicly accessible.

7 Recommendations

Our results suggest recommendations for (i) practitioners choosing models and workflows for rule generation, (ii) researchers studying the scope and limits of rule synthesis, and (iii) transformation-language designers building more LLM-compatible tooling.

For Practitioners. Transformation rule generation can help automate repetitive code modifications from minimal input. Developers can generate a rule from a single example and apply it recursively across a codebase, significantly reducing manual effort. Such techniques could be embedded into IDEs, where developers provide a representative code diff and receive suggested transformation rules that can be interactively reviewed and applied. An alternative usage scenario consists in generating transformation rules from an input code fragment and a natural-language instruction, enabling developers to describe the intended modification while leveraging LLMs to synthesize the corresponding rule.

However, our findings indicate that transformation rule generation is best suited for integration into *semi-automated workflows* rather than fully autonomous refactoring systems. In particular, complex refactorings are better handled when decomposed into smaller, composable subrules, which improves both reliability and interpretability of the generated transformations. More generally, the intended level of abstraction is not always recoverable from examples alone: LLMs may overfit the observed edits, while more general and reusable rules often depend on explicit intent provided by the user or by additional specifications.

Finally, our results reveal an inherent trade-off between model capability and operational cost. Frontier models handle complex transformations better but require external APIs and higher computational resources. In contrast, smaller models such as Llama-based models are effective at generating simple, one-line transformation rules, making them suitable for local deployment scenarios where

privacy constraints are critical. These results support hybrid workflows, combining different models depending on task complexity.

For Researchers. Beyond benchmarking model performance, our results point to three important research directions: *understanding which software-evolution tasks are inherently amenable to rule synthesis*, improving controllable generalization, and designing effective decomposition strategies for complex transformations.

In addition, rule generation can serve as a research instrument. Automatically inferred rules provide a powerful abstraction for analyzing software evolution at scale, enabling systematic studies of recurring transformation patterns. In particular, researchers can leverage these rules to analyze the distribution and frequency of common bug patterns, API misuses, and migration practices across projects. Furthermore, transformation rules offer a structured and interpretable representation of LLM outputs, which can facilitate their analysis and evaluation. They can also be integrated into fuzzing pipelines, where generated rules are used to systematically explore variations of code transformations. Finally, rule mining can support dataset augmentation and the mining of software repositories, enabling richer and more diverse benchmarks.

Our results also raise questions about benchmark design. Datasets such as Defects4J, BugsInPy, and PyMigBench are significantly more challenging due to the size and complexity of their diffs. These transformations involve large, heterogeneous modifications that do not consistently share common structures between the “before” and “after” code, limiting the ability of models to infer concise and reusable transformation rules. These findings suggest that current benchmarks are not always aligned with the assumptions underlying transformation rule generation, and that future evaluations should explicitly account for task suitability.

For Transformation-Language Designers and Tool Builders. We identify recurring failure modes that are relevant to DSL and tool design. Models may overgeneralize transformation rules, leading to excessive and potentially incorrect matches, or undergeneralize them, resulting in overly specific rules that fail to transfer to new contexts. This highlights the difficulty of balancing precision and generality when inferring transformation rules, and suggests the need for mechanisms to *better control rule scope*. Transformation languages such as GritQL and Ast-Grep rely on AST node matching, which makes large rigid rules difficult to apply effectively; tool designers could consider supporting explicit rule decomposition or hierarchical rule composition. More generally, many remaining failures stem from syntactic and configuration constraints of the target DSL (e.g., malformed YAML, invalid multi-node patterns) rather than from an inability to infer the intended edit. This suggests that more LLM-friendly configuration formats or better error recovery in DSL engines could substantially improve rule applicability.

Our results suggest that adaptation strategies for transformation-rule generation must be carefully aligned with the target DSL and software-evolution task. In our setting, RAG negatively impacted performance, likely because retrieved examples were biased toward Python transformations and unevenly relevant to the target context. More generally, external guidance only helps when it is relevant, diverse, and closely aligned with the requirements of the target transformation. These findings indicate that improving rule generation is not simply a matter of adding more context, but of better matching the conditioning signal to DSL constraints, task structure,

and language diversity. We believe the qualitative analysis to be particularly useful in this regard, because it exposes recurring success patterns and failure anti-patterns that can be operationalized into more systematic support for future rule-generation systems.

8 Related Work

Recent research has explored automated generation of program transformation rules, either by learning from code edit examples or by inferring rules from recurring security and optimization patterns. In particular, many papers focus on generating rules from input and output code pairs resulting from one or more code changes.

Ketkar et al. [26] infer transformation rules from type-change code edits using the Comby DSL and an anti-unification algorithm [39]. This algorithm generalizes multiple code fragments by extracting their common structure and replacing differences with variable placeholders (e.g., Comby placeholders such as `[var]`). This approach was later extended by PyEvolve [13], which improves the handling of unseen data-flow and control-flow variants through graph-based analysis. More recently, PyCraft [12] further expands this work by introducing a Code Change Pattern (CPAT) miner to retrieve similar code changes for rule inference, combined with an LLM-based generator that produces synthetic CPAT variants.

Ramos et al. have developed MELT [41], a rule generation tool that infers rules from pull requests with a relevant code edit, such as the API method we want to infer the rule from. They also proposed SPELL [40], which augments MELT with synthetic input-output-context triplets generated using large language models. These triplets are fed to an anti-unification algorithm to infer an initial rule, which is then iteratively refined by an LLM-based agent.

However, SPELL differs from our setting in both data source and objective. First, SPELL relies on synthetically generated examples to construct transformation rules in a unique DSL, whereas our evaluation is conducted on real-world code changes. Second, SPELL assumes access to multiple examples of the same transformation, which are aggregated to infer a generalized rule that can later be applied to real-world repositories. In contrast we aim to infer transformation rules directly from real-world diffs across multiple DSLs. Because SPELL and MELT methods rely mainly on anti-unification and generation from simple examples to infer transformation rules, they cannot capture transformations that involve some advanced semantic changes. Specifically, existing tools handle only deletions and modifications, as they rely on matching code fragments extracted from diffs. In contrast, our study evaluates LLMs on all types of code edits, including *additions*. This increases the models' task complexity, since they must identify the appropriate anchor location in the code where the new statement should be inserted.

Besides, the automatic generation of rules is widely explored in fields such as the detection of recurrent security vulnerabilities and optimization issues. These approaches aim to identify common patterns in code and derive reusable transformations or fixes that can be systematically applied across a codebase. For instance, prior work like CQLLM [18] has leveraged LLMs and vector knowledge retrieval techniques to infer CodeQL transformation rules to detect common code vulnerabilities. Zhao et al. [51] explored Semgrep rule generation from code diffs to generate optimization strategies which could be reused in a similar context at function-level. Wang

et al. proposed RulePilot [47], a solution using natural-language-based descriptions of a vulnerability to automatically generate the detection rules. They equipped the LLM with an abstract representation of the complexity of config rules into a standardized format, reducing hallucinations, allowing LLMs to focus on rule generation.

Nevertheless, these approaches are often limited for low-resource and domain-specific programming languages, which are under-represented in current LLM training data [21]. Researchers address these limitations with techniques such as domain-specific pre-training, fine-tuning, or retrieval-augmented generation (RAG).

9 Conclusion

We presented an empirical study on the ability of LLMs to generate code transformation rules expressed in DSLs. By evaluating three LLMs across three transformation DSLs and six benchmarks covering multiple software-evolution tasks, we provide a comprehensive assessment of rule synthesis capabilities in realistic settings.

Our results show that LLMs are capable of generating syntactically valid transformation rules, with strong performance though not perfect, from models such as GPT-5.4. However, effectiveness varies significantly depending on the complexity of the transformation, the structure of the underlying code changes, and the expressiveness of the target DSL. In particular, localized edits such as API misuse corrections are substantially more amenable to rule synthesis than function-level repairs or multi-statement migrations. We also observe a trade-off between abstraction and coverage, where models tend to generate overly specific rules instead of leveraging meta-variables and higher-level abstractions. At the same time, many of the generated rules exhibit high reusability, successfully matching multiple code instances across the evaluated datasets. Moreover, our qualitative analysis explains the concrete mechanisms behind model differences and failure modes, including formatting failures, over- and under-generalization, and missing edits. The anti-unification baseline performs overall worse than LLMs in terms of both syntactic and semantic correctness. Nevertheless, combining anti-unification with frontier LLMs could provide a promising hybrid approach with improved determinism.

Overall, our findings show that the main question has shifted from whether LLMs can synthesize transformation rules to under which task and representation conditions the generated rules are sound, correct, generalizable, and reusable. Such approaches are better suited for integration into semi-automated workflows, where generated rules can be reviewed, refined, and applied at scale.

We plan to extend the evaluation to additional benchmarks, models, and DSLs, and investigate possible improvements. A natural next step is to leverage this capability in real-world workflows, e.g., by integrating rule generation into IDEs or CI pipelines. Beyond deployment, future work also includes designing refinement loops with iterative feedback to improve incorrect rules (if any).

Data Availability Statement. We provide both a result artifact on Figshare [2] and a replication package <https://anonymous.4open.science/r/EmpiricalStudyTransformationRules-13B3>.

Acknowledgments

This work is supported by the Inria Défi LLM4Code (DGDS012482).

References

- [1] Ghazi Alkhatib. 1992. The maintenance problem of application software: An empirical analysis. *Journal of Software Maintenance: Research and Practice* 4, 2 (1992), 83–104. doi:10.1002/smr.4360040203
- [2] Axel Allain. 2026. *Result Artifact for “Code Transformation Rule Synthesis Using LLMs: Potential and Limits”*. doi:10.6084/m9.figshare.31861126.v1
- [3] Sven Amann, Sarah Nadi, Hoan A Nguyen, Tien N Nguyen, and Mira Mezini. 2016. MUBench: A benchmark for API-misuse detectors. In *Proceedings of the 13th international conference on mining software repositories*. 464–467. doi:10.1145/2901739.2903506
- [4] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N Nguyen, and Mira Mezini. 2018. A systematic evaluation of static api-misuse detectors. *IEEE Transactions on Software Engineering* 45, 12 (2018), 1170–1188. doi:10.48550/arXiv.1712.00242
- [5] Nishil Amin, Zhiwei Fei, Xiang Li, Justyna Petke, and He Ye. 2026. JMigBench: A Benchmark for Evaluating LLMs on Source Code Migration (Java 8 to Java 11). In *Proceedings of the 1st Workshop on Code Translation, Transformation, and Modernization (ReCode '26)* (Rio de Janeiro, Brazil). ACM, New York, NY, USA, 7. doi:10.1145/3786180.3788316
- [6] ast-grep. 2026. ast-grep. Retrieved July 30, 2026 from <https://ast-grep.github.io/>
- [7] Sebastian Baltes, Florian Angermeier, Chetan Arora, Marvin Muñoz Barón, Chunyang Chen, Lukas Böhme, Fabio Calefato, Neil Ernst, Davide Falesti, Brian Fitzgerald, Davide Fucci, Junda He, Christoph Treude, Marcos Kalinowski, Stefano Lambiase, Daniel Russo, Mircea Lungu, Cristina Martinez Montes, Lutz Prechelt, Paul Ralph, Rijnard van Tonder, and Stefan Wagner. 2026. Guidelines for Empirical Studies in Software Engineering involving Large Language Models. Accepted manuscript, arXiv:2508.15503v7 [cs.SE]. doi:10.48550/arXiv.2508.15503
- [8] Rajiv D Banker. 1992. Software complexity and software maintenance costs. (1992).
- [9] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2025. Repairagent: An autonomous, llm-based agent for program repair. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 2188–2200. doi:10.1109/ICSE55347.2025.00157
- [10] Jonathan Cordeiro, Shayan Noei, and Ying Zou. 2024. An empirical study on the code refactoring capability of large language models. *ACM Transactions on Software Engineering and Methodology* (2024). doi:10.1145/3801158
- [11] Jonathan Cordeiro, Shayan Noei, and Ying Zou. 2025. LLM-Driven Code Refactoring: Opportunities and Limitations. In *2025 IEEE/ACM Second IDE Workshop (IDE)*. IEEE, 32–36.
- [12] Malinda Dilhara, Abhiram Bellur, Timofey Bryksin, and Danny Dig. 2024. Unprecedented Code Change Automation: The Fusion of LLMs and Transformation by Example. *Proceedings of the ACM on Software Engineering* 1, FSE (July 2024), 631–653. doi:10.1145/3643755
- [13] Malinda Dilhara, Danny Dig, and Ameya Ketkar. 2023. PYEVOLVE: Automating Frequent Code Changes in Python ML Systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, Melbourne, Australia, 995–1007. doi:10.1109/ICSE48619.2023.00091
- [14] Akalanka Galappaththi, Sarah Nadi, and Christoph Treude. 2024. An Empirical Study of API Misuses of Data-Centric Libraries. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, Barcelona Spain, 245–256. doi:10.1145/3674805.3686685
- [15] Robert L Glass. 2001. Frequently forgotten fundamental facts about software engineering. *IEEE software* 18, 3 (2001), 112–111. doi:10.1109/MS.2001.922739
- [16] Yaroslav Golubev, Zarina Kurbatova, Eman Abdullah AlOmar, Timofey Bryksin, and Mohamed Wiem Mkaouer. 2021. One thousand and one stories: a large-scale survey of software refactoring. In *Proceedings of the 29th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 1303–1313. doi:10.1145/3468264.3473924
- [17] GritQL. 2026. GritQL. Retrieved July 30, 2026 from <https://docs.grit.io/>
- [18] Weihong Han, Chan Chen, Junyi Zhu, Rufeng Zhan, and Weihong Han. 2025. CQLLM: A Framework for Generating CodeQL Security Vulnerability Detection Code Based on Large Language Model. doi:10.20944/preprints202510.1458.v1
- [19] Mohayeminul Islam, Ajay Kumar Jha, Sarah Nadi, and Ildar Akhmetov. 2023. PyMigBench: A Benchmark for Python Library Migration. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, Melbourne, Australia, 511–515. doi:10.1109/MSR59073.2023.00075
- [20] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Syvatkovskiy. 2023. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM joint european software engineering conference and symposium on the foundations of software engineering*. 1646–1656. doi:10.1145/3611643.3616244
- [21] Sathvik Joel, Jie JW Wu, and Fatemeh H. Fard. 2025. A Survey on LLM-based Code Generation for Low-Resource and Domain-Specific Programming Languages. arXiv:2410.03981 [cs.SE]. doi:10.48550/arXiv.2410.03981
- [22] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. doi:10.1145/2610384.2628055
- [23] Rafael-Michael Karampatsis and Charles Sutton. 2020. How Often Do Single-Statement Bugs Occur? The ManyStubs4J Dataset. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20)*. Association for Computing Machinery, New York, NY, USA, 573–577. doi:10.1145/3379597.3387491
- [24] Zohra Kaouter Kebaili, Djamel Eddine Khelladi, Mathieu Acher, and Olivier Barais. 2024. An empirical study on leveraging llms for metamodels and code co-evolution. *The Journal of Object Technology* 23, 3 (2024), 1–14.
- [25] Zohra Kaouter Kebaili, Djamel Eddine Khelladi, Mathieu Acher, and Olivier Barais. 2025. Automated co-evolution of metamodels and code. *IEEE Transactions on Software Engineering* (2025). doi:10.1109/TSE.2025.3540545
- [26] Ameya Ketkar, Oleg Smirnov, Nikolaos Tsantalis, Danny Dig, and Timofey Bryksin. 2022. Inferring and applying type changes. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1206–1218. doi:10.1145/3510003.3510115
- [27] Djamel Eddine Khelladi, Benoit Combemale, Mathieu Acher, Olivier Barais, and Jean-Marc Jézéquel. 2020. Co-evolving code with evolving metamodels. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 1496–1508. doi:10.1145/3377811.3380324
- [28] Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, and Yann Gaël Guéhéneuc. 2020. Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software* 167 (2020), 110610. doi:10.1016/j.jss.2020.110610
- [29] Quentin Le Dilavrec, Djamel Eddine Khelladi, Arnaud Blouin, and Jean-Marc Jézéquel. 2021. Untangling spaghetti of evolutions in software histories to identify code and test co-evolutions. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 206–216. doi:10.1109/ICSME52107.2021.00025
- [30] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65. doi:10.1145/3318162
- [31] Xia Li, Jiajun Jiang, Samuel Benton, Yingfei Xiong, and Lingming Zhang. 2021. A Large-scale Study on API Misuses in the Wild. In *2021 14th IEEE conference on software testing, verification and validation (ICST)*. IEEE, 241–252. doi:10.1109/ICST49551.2021.00034
- [32] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and Tegawendé F Bissyandé. 2021. A critical review on the evaluation of automated program repair systems. *Journal of Systems and Software* 171 (2021), 110817. doi:10.1016/j.jss.2020.110817
- [33] Zhipeng Liu, Xuefeng Bai, Kehai Chen, Xinyang Chen, Xiucheng Li, Yang Xiang, Jin Liu, Hong-Dong Li, Yaowei Wang, Liqiang Nie, et al. 2025. A survey on the feedback mechanism of LLM-based AI agents. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*. International Joint Conferences on Artificial Intelligence, 10582–10592. doi:10.24963/ijcai.2025/1175
- [34] Tom Mens and Tom Tourwé. 2004. A survey of software refactoring. *IEEE Transactions on software engineering* 30, 2 (2004), 126–139. doi:10.1109/TSE.2004.1265817
- [35] Charles Miranda, Guilherme Avelino, and Pedro Santos Neto. 2025. Test Co-Evolution in Software Projects: A Large-Scale Empirical Study. *Journal of Software: Evolution and Process* 37, 7 (2025), e70035. doi:10.1002/smr.70035
- [36] Martin Monperrus. 2018. *The living review on automated program repair*. Ph. D. Dissertation. HAL Archives Ouvertes.
- [37] Trong Duc Nguyen, Anh Tuan Nguyen, and Tien N Nguyen. 2016. Mapping API elements for code migration with vector representations. In *Proceedings of the 38th international conference on software engineering companion*. 756–758. doi:10.1145/2889160.2892661
- [38] Linda Northrop, Peter Feiler, Richard P Gabriel, John Goodenough, Rick Linger, Tom Longstaff, Rick Kazman, Mark Klein, Kevin Sullivan, Kurt Wallnau, et al. 2006. Ultra-large-scale systems: The software challenge of the future. (2006).
- [39] Gordon D Plotkin. 1970. A Note on Inductive Generalization. ... (1970).
- [40] Daniel Ramos, Catarina Gamboa, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. 2026. SPELL: Synthesis of Programmatic Edits using LLMs. arXiv:2602.01107 [cs]. doi:10.48550/arXiv.2602.01107
- [41] Daniel Ramos, Hailie Mitchell, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. 2023. MELT: Mining Effective Lightweight Transformations from Pull Requests. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1516–1528. doi:10.1109/ASE56229.2023.00117
- [42] Vivek Sarkar, William Harrod, and Allan E Snively. 2009. Software challenges in extreme scale systems. In *Journal of Physics: Conference Series*, Vol. 180. 012045. doi:10.1088/1742-6596/125/1/012045
- [43] Atsushi Shirafuji, Yusuke Oda, Jun Suzuki, Makoto Morishita, and Yutaka Watanobe. 2023. Refactoring programs using large language models with few-shot examples. In *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 151–160. doi:10.1109/APSEC60848.2023.00024
- [44] Harvey P Siy and Dewayne E Perry. 1998. Challenges in evolving a large scale software product. In *Proceedings of Principles of Software Evolution Workshop at the International Software Engineering Conference, ICSE*, Vol. 98. 251–260.

- [45] Amann Sven, Hoan Anh Nguyen, Sarah Nadi, Tien N Nguyen, and Mira Mezini. 2019. Investigating next steps in static API-misuse detection. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 265–275. doi:10.1109/MSR.2019.00053
- [46] Rijnard van Tonder and Claire Le Goues. 2019. Lightweight multi-language syntax transformation with parser parser combinators. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 363–378. doi:10.1145/3314221.3314589
- [47] Hongtai Wang, Ming Xu, Yanpei Guo, Weili Han, Hoon Wei Lim, and Jin Song Dong. 2025. RulePilot: An LLM-Powered Agent for Security Rule Generation. arXiv:2511.12224 [cs]. doi:10.48550/arXiv.2511.12224
- [48] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. 2020. BugsInPy: A Database of Existing Bugs in Python Programs to Enable Controlled Testing and Debugging Studies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1556–1560. arXiv:2401.15481 [cs]. doi:10.1145/3368089.3417943
- [49] Boyang Yang, Zijian Cai, Fengling Liu, Bach Le, Lingming Zhang, Tegawendé F Bissyandé, Yang Liu, and Haoye Tian. 2025. A survey of LLM-based automated program repair: Taxonomies, design paradigms, and applications. *arXiv preprint arXiv:2506.23749* (2025). doi:10.48550/arXiv.2506.23749
- [50] Jiyang Zhang, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2023. Multilingual code co-evolution using large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 695–707. doi:10.1145/3611643.3616291
- [51] Yuwei Zhao, Yuan-An Xiao, Qianyu Xiao, Zhao Zhang, and Yingfei Xiong. 2025. SemOpt: LLM-Driven Code Optimization via Rule-Based Analysis. arXiv:2510.16384 [cs]. doi:10.48550/arXiv.2510.16384
- [52] Celal Ziftci, Stoyan Nikolov, Anna Sjövall, Bo Kim, Daniele Codecasa, and Max Kim. 2025. Migrating code at scale with llms at google. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. 162–173. doi:10.48550/arXiv.2504.09691
- [53] Nada Zine, Clément Quinton, and Romain Rouvoy. 2025. LLM-based Co-Evolution of Configurable Software Systems. In *Proceedings of the 2025 29th ACM International Systems and Software Product Line Conference-Volume A*. 27–38. doi:10.1145/3744915.3748460

Received 2026-03-26; accepted 2026-06-18