MatchFixAgent: Language-Agnostic Autonomous Repository-Level Code Translation Validation and Repair

ALI REZA IBRAHIMZADA*, University of Illinois Urbana-Champaign, USA
BRANDON PAULSEN, Amazon, USA
REYHANEH JABBARVAND, University of Illinois Urbana-Champaign, USA
JOEY DODDS, Amazon, USA
DANIEL KROENING, Amazon, USA

Code translation transforms source code from one programming language (PL) to another. Validating the functional equivalence of translation and repairing, if necessary, are critical steps in code translation. Existing automated validation and repair approaches struggle to generalize to many PLs due to high engineering overhead, and they rely on existing and often inadequate test suites, which results in false claims of equivalence and ineffective translation repair. To bridge this gap, we develop MatchFixAgent, a large language model (LLM)-based, PL-agnostic framework for equivalence validation and repair of translations. MatchFixAgent features a multi-agent architecture that divides equivalence validation into several sub-tasks to ensure thorough and consistent semantic analysis of the translation. Then it feeds this analysis to *test agent* to write and execute tests. Upon observing a test failure, the *repair agent* attempts to fix the translation bug. The final (in)equivalence decision is made by the *verdict agent*, considering semantic analyses and test execution results.

We compare MatchFixAgent's validation and repair results with four repository-level code translation techniques. We use 2,219 translation pairs (each consisting of a source function and its translation) from their artifacts, which cover 6 PL pairs, and are collected from 24 GitHub projects totaling over 900K lines of code. Our results demonstrate that MatchFixAgent produces (in)equivalence verdicts for 99.2% of translation pairs, with the same equivalence validation result as prior work on 72.8% of them. When MatchFixAgent 's result disagrees with prior work, we find that 60.7% of the time MatchFixAgent 's result is actually correct. In addition, we show that MatchFixAgent can repair 50.6% of inequivalent translation, compared to prior work's 18.5%. This demonstrates that MatchFixAgent is far more adaptable to many PL pairs (with a small overhead of 1,650 lines of code) than prior work, while producing highly accurate validation results.

Additional Key Words and Phrases: Program Analysis, Neuro-Symbolic Code Translation Validation and Repair, LLM Agent

1 Introduction

Code translation, the process of converting source code from one programming language (PL) to another, is a cornerstone of software modernization efforts that enhance performance, maintainability, and reliability [30, 31, 34, 35]. Translation validation and repair are integral steps in code translation for determining functional equivalence and patch generation for incorrect translations. However, performing validation and repair manually—particularly in large codebases—can be tedious, time-consuming, and error-prone, especially when complex code structures and dependencies are involved [28, 37, 75]. Prior work on repository-level code translation defines value and type equivalences and translations to compare source and target implementations over pairs of concrete inputs. The inputs either come from existing tests from the source project [29, 53, 73, 86] or differential fuzzing [17, 82]. Despite notable advancements in validation and repair of repository-level code translation, existing techniques are hampered by the following limitations:

*Author was an intern at AWS at the time of this work.

Authors' Contact Information: Ali Reza Ibrahimzada, alirezai@illinois.edu, University of Illinois Urbana-Champaign, Urbana, Illinois, USA; Brandon Paulsen, bpaulse@amazon.com, Amazon, Arlington, Virginia, USA; Reyhaneh Jabbarvand, reyhaneh@illinois.edu, University of Illinois Urbana-Champaign, Urbana, Illinois, USA; Joey Dodds, jldodds@amazon.com, Amazon, Portland, Oregon, USA; Daniel Kroening, dkr@amazon.com, Amazon, Seattle, Washington, USA.

(1) *Difficulty Generalizing to Many PL Pairs.* While the fundamental ideas of current validation approaches extend to many PL pairs, their actual implementations typically support just one language pair. This is because supporting language interoperability between a pair of PLs requires a large engineering effort, as evidenced by the size of these tools[1]. Given the quadratic number of PL pairs, language interoperability techniques are extremely challenging to scale to many PL pairs.
(2) *Unknown Test Requirements.* Current translation validation approaches require a set of valid inputs to validate the input-output equivalence between source functions and their translation. They generate these inputs by either executing available source tests [29, 53, 73, 86] or fuzzing the source project [17, 82] Unit tests are often incomplete, missing important inputs, and resulting in false claims of equivalence. Fuzzing techniques suffer from generating invalid inputs, also resulting in false claims of inequivalence, and in general, failing to reach deep into the code or create complex objects in the context of real-world projects [27, 36].
(3) *Ineffective Translation Repair.* Recent studies have shown that a more rigorous validation can reveal more translation bugs [49]. Hence, advancement in translation validation should be accompanied by effective repair strategies. Existing techniques, however, either require the user to fix incorrect translations manually [73] or use feedback-driven re-prompting strategies [29, 86] that are barely effective in repository-level code translation due to long call-chain dependencies in real-world projects [29].

Prior work consists of high-effort implementations that can deliver inconsistent results and only apply to one of a quadratic number of language pairs. Large Language Models (LLMs) have recently been successful at same-language equivalence validation [41, 76], so replacing cross-language equivalence implementations with LLM decisions is a logical next step. While there is a risk of incorrect results, the baseline mechanical approaches already display low accuracy.

Towards this end, we propose MatchFixAgent, a *language-agnostic* approach to automate *validation* and *repair* of repository-level code translation (§3.1). MatchFixAgent combines the generative power of LLMs with several approximate code semantic analyses, i.e., analysis of control- and data-flow paths, library APIs, exception handling, and (formal or informal) specification (§3.2). These semantic analyses are then fed to a *test generator & repair agent* to generate tests for assessing functional equivalence, and repair the translation in the case of failing tests (§3.3). The final equivalence decision will be made by the *verdict* agent, considering the approximate semantic analyses and test execution results (§3.4). MatchFixAgent is very lightweight (1650 lines of code), modular, and interoperable with existing repository-level translation systems.

We evaluate the effectiveness of MatchFixAgent for repository-level translation validation and repair against four existing techniques [29, 47, 73, 86]. Our benchmark comprises 2,219 source–translation function pairs, which cover 6 PL pairs, drawn from 24 real-world projects totaling over 900K lines of code (§4.1). For each translation pair, we obtain an equivalence verdict (validation outcome) from both MatchFixAgent and other techniques. Overall, MatchFixAgent returns a verdict for 99.2% of pairs, while alternative approaches do so for only 71.6% (§4.2.1). On the 1,571 pairs where both produce verdicts, MatchFixAgent agrees with other approaches in 72.8% of cases. For the remaining disagreements, a systematic manual investigation finds MatchFixAgent to be correct in 60.7% of cases and incorrect in the rest (§4.2.2). In translation repair, MatchFixAgent can fix 50.6% of translation bugs, 32.1% more than existing approaches (§4.3). We show that MatchFixAgent is compatible with different LLMs and agent frameworks, producing comparable results (§4.4). Lastly, our ablation study shows that removing code analyses and in-the-loop test

[1]The implementation of notable recent translation and validation techniques are AlphaTrans (10859 LoC), Oxidizer (19052 LoC), and Skel (3843 LoC).

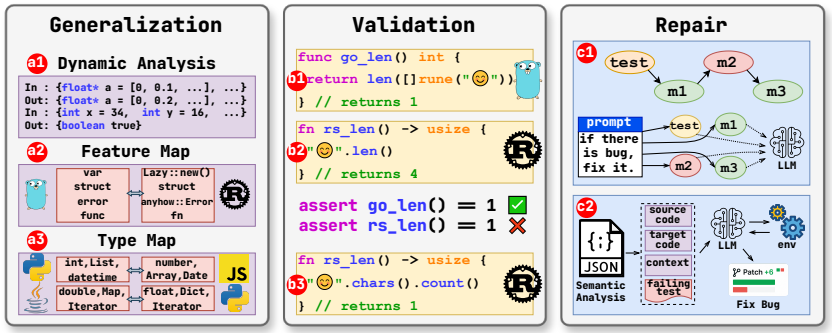

Fig. 1. Illustration of key limitations of existing techniques in validation and repair of repository-level code translation and MatchFixAgent addressing them.

generation reduces verdict accuracy by 42.3%, while increasing token usage by 5.2% (§4.5). These results confirm that MatchFixAgent is a viable alternative to prior work's validation and repair approaches, while being vastly easier to adapt to new PL pairs.

Our notable contributions are:

(1) We present MatchFixAgent, a PL-agnostic, agentic approach for validation and repair of repository-level code translation.
(2) We demonstrate that MatchFixAgent is a viable alternative to prior work's validation approach, while being vastly easier to adapt to new PL pairs.
(3) We demonstrate the benefit of MatchFixAgent's multi-agent architecture compared to simpler standalone-agent design.

2 Limitations of Prior Work

To demonstrate the limitations of existing techniques [29, 53, 73, 86] for validation and repair, we use the examples in Figure 1.

Limitation 1: Generalization. Existing automated translation tools require substantial engineering effort to build validation systems for individual language pairs. For instance, Oxidizer [86] and Syzygy [53] require dynamic analysis and I/O extraction from source code (a1). Oxidizer further requires a predefined feature map (a2), which together with their dynamic analyzer is 19,000 lines of code. Other tools like AlphaTrans [29] and Skel [73] validate programs using high quality type map (a3), which requires manual maintenance over time as PLs evolve. In comparison, MatchFixAgent uses only language-agnostic LLM prompts, an off-the-shelf LLM coding agent tool, and lightweight static analysis. While the static analysis must be implemented for each PL, each PL requires approximately 280 additional lines of code to support, hence it is extremely easy to generalize to many PLs.

Limitation 2: Validation. Most prior work depends on existing source project tests for translation validation, but these tests may have insufficient coverage. The test suites in AlphaTrans [29] have an average of 56.57% method coverage. Even when developer-written tests provide adequate coverage, they may miss the edge cases that expose subtle semantic differences between source and target languages. Consider the real-world case from Oxidizer [86], where a `Go` program (b1) that counts characters in a string is translated to `Rust` (b2) using the `.len()` method, which counts bytes rather than characters. Oxidizer validates this pair as *functionally equivalent* because it only exercises this program with ASCII inputs. MatchFixAgent, in contrast, marks the translation as not equivalent, and synthesizes a test with Unicode inputs (e.g., `U+1F60A`, a four-byte emoji representing a single character) to confirm the inequivalence. Upon detecting the translation bug,

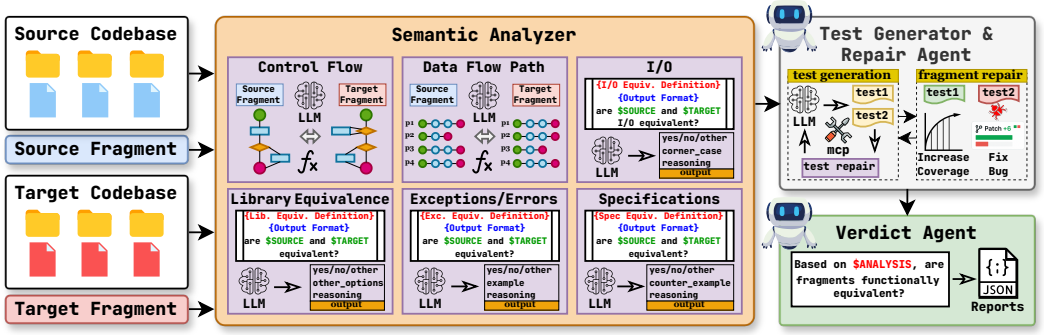

Fig. 2. Overview of MatchFixAgent.

it automatically generates a patch that uses `.chars().count()` to ensure proper handling of both ASCII and Unicode characters (b3).

Limitation 3: Repair. Current translation repair techniques solely rely on simple feedback-driven approaches, which have proven inadequate in practical settings. Skel [73], Oxidizer [86], and Syzygy [53] utilize multi-turn iterative prompting techniques. AlphaTrans [29] adopts an execution trace-based reprompting strategy. For instance, consider scenario (c1), where running the `test` method sequentially invokes methods `m1`, `m2`, and `m3`. If a translation bug is present in fragment `m2`, AlphaTrans reprompts all executed fragments individually without considering inter-fragment dependencies. This approach, while potentially resolving the bug in `m2`, risks introducing new functional bugs in previously correct fragments, such as `m1`. To overcome this limitation, MatchFixAgent uses code analysis and an LLM agent (c2). The analyses expose dependencies such as the one between `m1` and `m2` and the agent interacts with the execution environment to execute, validate, and iteratively refine its generated patches.

3 MatchFixAgent

In this section, we discuss MatchFixAgent and its three primary components in more detail. We first provide a high-level overview, then go into more details.

3.1 Overview

Figure 2 gives an overview of MatchFixAgent, which consists of three main components: (1) the Semantic Analyzer (§3.2), (2) the Test Generator & Repair Agent (§3.3), and (3) the Verdict Agent (§3.4). MatchFixAgent's main inputs are: a translation pair (a source function and its translation), the source project, and the source project's translation. MatchFixAgent's outputs are: an equivalence verdict, a natural language report, and an optional patch that repairs the translation if the translation was found to be not equivalent. In addition, MatchFixAgent is configured with an LLM, a set of tools for the agents, and a timeout.

Details of MatchFixAgent's algorithm are shown in Algorithm 1. First, the Semantic Analyzer is executed, which uses an LLM to analyze different semantic properties of the source function and its translation. We decompose this task into six sub-tasks for the LLM, which are executed independently. Each task analyzes a different semantic property of the source and translation, namely control flow (§3.2.1), data flow (§3.2.2), input and output mapping (§3.2.3), library API usage (§3.2.4), exception and error handling (§3.2.5), and specifications (§3.2.6). This sub-task architectures help to keep the LLM focused, and improves overall reliability and consistency [6]. Each sub-task outputs a report that summarizes differences between the source and translation for the given semantic property.

```
============ Source Code ============
def max(a, b):
    """
    The function max(a, b) returns the
    greater of the two input values a
    and b. If a is greater than b, it
    returns a; otherwise, it returns a.
    """
    if a > b:
        return a
    else:
        return b
```

```
========= Control Flow Graph =========
Node 0: PARAMETERS
    parameter: a, parameter: b
    → Node 1: ENTRY
Node 1: ENTRY
    → Node 2: if a > b:
Node 2:
    if a > b:
    → Node 3 [label="a > b"]:
        return a
    → Node 4 [label="(a <= b)"]:
        return b
```

```
========== Data Flow Paths ==========
Variable: a
    Path 1:
        CFG Node Path: 0 → 1 → 2
    Path 2:
        CFG Node Path: 0 → 1 → 2 → 3

Variable: b
    Path 1:
        CFG Node Path: 0 → 1 → 2
    Path 2:
        CFG Node Path: 0 → 1 → 2 → 4
```

Fig. 3. CFG and DFP structures extracted by the Semantic Analyzer component in MatchFixAgent.

These reports are then fed to the Test Generator and Repair Agent, which uses an off-the-shelf LLM coding agent, such as Claude Code [56] or Codex [64], to write executable test cases that may reveal inequivalent behavior. If the agent discovers inequivalent behavior, the agent also attempts to write a patch to repair the translation. This component outputs a boolean equivalence verdict, a set of tests for both the source project and translation, and an optional patch.

Finally, the Verdict Agent takes the outputs from the Semantic Analyzer and Test Generator and Repair Agent, which validates the claims made in those outputs, and provides a final verdict on the functional equivalence between the source and translation. The output includes a final boolean equivalence verdict, an overall summary, and the outputs from the Test Generator and Repair Agent.

Algorithm 1: MatchFixAgent

```
Input : sourceProject, sourceFunc, translatedProject,
        translatedFunc, LLM, tools, timeout
Output: validationRepairReport
transPair ← [ sourceFunc, translatedFunc ]
Function async semAnalyzer (transPair, LLM):
    cfgSrc, dfSrc ← build_cfg (sourceFunc)
    cfgTgt, dfTgt ← build_cfg (translatedFunc)
    return
      { controlFlowAnalyzer (cfgSrc, cfgTgt, transPair, LLM),
        dataFlowPathAnalyzer (dfSrc, dfTgt, transPair, LLM),
        ioAnalyzer (transPair, LLM),
        libraryAnalyzer (transPair, LLM),
        exceptionAnalyzer (transPair, LLM),
        specAnalyzer (transPair, LLM) }
await semAnalysis ← semAnalyzer (cfgSrc, dfSrc, cfgTgt, dfTgt)
testRepair ← testGenRepairAgent (prompt, LLM, tools,
  timeout)
verdict ← verdictAgent (semAnalysis, testRepair, LLM)
validationRepairReport ← semAnalysis ∪ testRepair ∪ verdict
return validationRepairReport
```

3.2 Semantic Analyzer

The Semantic Analyzer takes as input the translation pair and an LLM. It first computes a control flow graph (CFG) and data flow graph (DFG) (described in Sections 3.2.1 and 3.2.2 respectively), and then calls six sub-analyzers in parallel, each of which analyzes a different semantic property of the translation pair. Each sub-analyzer invokes the LLM with a custom prompt describing the analysis to perform. The prompts are relatively simple and short. The prompt first defines a role for the LLM ("`You are an expert in...`"), a general definition of functional equivalence, and a specific definition of equivalence for the semantic property. It then instructs the analyzer to output an equivalence verdict and explanation for the specific semantic property it analyzed. In addition, certain analyzers output examples to demonstrate inequivalence. The final output of the Semantic Analyzer is a 6-tuple containing a JSON-formatted output of each sub-analyzer. The following subsections provide more details on the prompts of the six sub-analyzers.

3.2.1 Control Flow Analyzer. The *Control Flow Analyzer* is prompted to analyze the the control flow structures of the source and translation are equivalent, looking for inequivalences like reordered conditions, missing branches, or altered loop termination criteria. To aid this task, we provide the LLM with textual representations of the source and translation's CFGs. An example is shown in Figure 3. To compute the CFG, we use Tree-Sitter [72] to construct an abstract syntax tree of the function, and then extract basic blocks and control flow structures. Tree-sitter supports 165+ PLs,

making this process PL-agnostic. Each PL supported by MatchFixAgent required approximately 280 lines of code, making it very easy to support many PLs.

To improve reliability and reduce costs, the control flow analyzer first (symbolically) computes a similarity score between the CFGs, and, if it falls above a threshold, it immediately returns an equivalent verdict without invoking the LLM. The overall procedure is shown in Algorithm 2. The analyzer abstracts each graph into canonical forms (lines 1–9) capturing node types (e.g., conditionals, loops, exception handlers) and edge types (control transfer relationships), then computes the structural similarity score based on the Jaccard index [12] (lines 10–12). We use 0.7 as the threshold. This results in approximately 25% of LLM invocations being skipped in our experiments, making this threshold relatively stringent.

Algorithm 2: Control Flow Analyzer

```
   Input  : cfgSource, cfgTarget, fragments, model
   Output : cfgAnalysis
1  Function abstractGraph (cfg):
2      for (u, v) with edge ∈ cfg do
3          uType, vType ← classifyNode (u), classifyNode (v)
4          edgeType ← classifyEdge (edge)
5          nodes ← nodes ∪ uType ∪ vType
6          edges ← edges ∪ ⟨ uType, edgeType, vType ⟩
7      return nodes, edges
8  sourceNodes, sourceEdges ← abstractGraph (cfgSource)
9  targetNodes, targetEdges ← abstractGraph (cfgTarget)
10 nodeSim ← jaccardSimilarity (sourceNodes, targetNodes)
11 edgeSim ← jaccardSimilarity (sourceEdges, targetEdges)
12 similarity ← ( 0.5 × nodeSim) + ( 0.5 × edgeSim)
13 if similarity ⩾ threshold = 0.7 then
14     cfgAnalysis ← { "is_equivalent": "yes" }
15 else
16     cfgAnalysis ← LLM (cfgSource, cfgTarget, fragments, model)
17 return cfgAnalysis
```

3.2.2 Data Flow Analyzer. The *Data Flow Analyzer* is prompted to evaluate whether the flow of data within the source and translation is equivalent, looking for issues like unused variables. To aid the LLM, we provide textual representations of the source and translation's data flow graphs (DFGs). An example is shown in Figure 3. We keep our data flow computation extremely simple. For each statement in the AST, we extract variable names, label them as a def or a use, and associate them with a CFG node. We then extract def-use chains. This is primarily a syntactic analysis. We do not handle challenging problems such as aliasing, concurrency, or context sensitivity. The per-PL implementation effort is approximately 280 lines of code based on the six PLs supported by MatchFixAgent.

Similar to our control flow analyzer, we compute a similarity score between the DFGs, and short-circuit if it falls above a threshold. The analyzer, show in Algorithm 3, first extracts *def-use* chains for parameters and local variables, capturing how data values are defined, propagated, and consumed. The extracted paths are compared using edit distance [42] as the similarity measure (lines 1–13). We again use 0.7 as the threshold, which results in approximately 35% of LLM invocations being skipped.

3.2.3 IO Analysis. The *IO Analyzer* is prompted to evaluate whether the observable input-output behavior of the source and target fragments is semantically equivalent. The prompt includes an IO equivalence definition, which assess five dimensions: (1) semantic equivalence of accepted inputs, (2) consistency of produced outputs, (3) preservation of side effects (e.g., file operations, network calls, or global state modifications), (4) uniform handling of edge cases, and (5) similarity in performance-critical complexity. The LLM is prompted also prompted to produce a plausible input that would trigger dissimilar IO behavior if it believes the translation is inequivalent. This methodology catches inequivalences such as differing error messages, inconsistent encoding assumptions, or missing side effects—often overlooked by structural analyses (§3.2.1,§3.2.2) alone.

3.2.4 Library Analyzer. The *Library API Analyzer* is prompted to consider the behavior of external library APIs called in the source and translation, and evaluate whether their differences result in

inequivalent behavior. This analyzer primarily detects subtle differences between similar library APIs in the source and translation. It provides suggestions to fix inequivalent behavior as well.

3.2.5 Exception & Error Analyzer. The *Exception and Error Handling Analyzer* is prompted to validate whether error detection, exception raising, and error recovery mechanisms in the source and target code fragments are functionally equivalent. The prompt include five dimensions for equivalence: (1) detecting and handling the same error conditions, (2) using semantically equivalent exception/error types, (3) producing equivalent error messages or codes, (4) preserving consistent recovery mechanisms, and (5) propagating errors in equivalent ways. If neither fragment implements explicit error handling, the analyzer deems them equivalent for this dimension. Otherwise, it statically identifies exception constructs (e.g., `try-catch`, `throw`, `return`-error patterns) and uses LLM reasoning to compare semantics. For instance, if the source raises a specific `FileNotFoundError` while the target raises a generic `IOException`, the discrepancy is flagged, as it may affect upstream handling. Where differences exist, the analyzer also recommends target-language error handling constructs that align with the source's semantics.

Algorithm 3: Data Flow Path Analyzer

```
Input  : dfSource, dfTarget, fragments, model
Output : dfAnalysis
Function computeEditDistance (srcPath, tgtPath):
    sim_a ← 0
    foreach xPath ∈ srcPath do
        best ← 0
        foreach yPath ∈ tgtPath do
            score ← jaccardSimilarity (xPath, yPath)
            best ← max (best, score)
        sim_a ← sim_a + best
    sim_a ← sim_a / | srcPath |
    sim_b ← "repeat loop with srcPath and tgtPath swapped"
    return (sim_a + sim_b) / 2
srcPath, tgtPath ← getVariablePaths (dfSource, dfTarget)
similarity ← computeEditDistance (srcPath, tgtPath)
if similarity ⩾ threshold = 0.7 then
    dfAnalysis ← { "is_equivalent": "yes" }
else
    dfAnalysis ← LLM (dfSource, dfTarget, fragments, model)
return dfAnalysis
```

3.2.6 Specifications Analyzer. The *Specification Analyzer* is prompted assesses whether the source and target code fragments adhere to the same explicit or implicit functional specifications. The prompt includes Specification equivalence definition which state that the source and translation should: (1) fulfill the same documented or inferred functional requirements, (2) satisfy identical pre-conditions and post-conditions, (3) maintain the same invariants, and (4) handle the same input domain, including edge cases. The LLM is instructed to extract available specifications from function signatures, type annotations, docstrings, and relevant comments, or, when no formal documentation exists, infer behavioral contracts from code semantics. The LLM is asked to compare the contracts of the source and translation. For example, if the source specifies "`returns 1 on success, 0 on failure`" and the target returns `Boolean` values, the inconsistency is flagged. In such cases, the LLM is instructed to produce a formalized specification that reconciles both implementations and provides counterexamples demonstrating behavior mismatch.

3.3 Test Generator & Repair Agent

The *Test Generator & Repair Agent* uses an off-the-shelf LLM coding agent and the reports from the Semantic Analyzer to write and execute tests that demonstrate functional (in)equivalence. This agent helps catch hallucinations and confirm the claims in the Semantic Analyzer reports. The prompt for the agent is shown in Figure 4, which includes a definition of equivalence, instructions to write tests in both the source and target language that test the (in)equivalence of the translation, and finally instructions to repair the translation if it is not equivalent. We use Claude Code [56] as the agent for most of our experiments, which comes with a set of tools out of the box, namely, reading + writing files, executing arbitrary shell commands, and searching the web. The agent

```
<fragment_details>
   <source_fragment_details> <path to source file> <implementation of source fragment> </source_fragment_details>
   <target_fragment_details> <path to target file> <implementation of target fragment> </target_fragment_details>
</fragment_details>

<instruction>
   You are an expert agent specializing in test generation and code repair. Based on the analysis from multiple expert agents
   regarding functional equivalence between $SOURCE_LANGUAGE and $TARGET_LANGUAGE implementations of the given method/function,
   your task is to generate tests and repair the target implementation, if necessary.
   <functional_equivalence_definition>
      Two code fragments in different programming languages are considered functionally equivalent if, when executed on the same
      input, they always have identical program states at all corresponding points reachable by program execution, and they both
      produce the same output upon termination.
   </functional_equivalence_definition>
   <rules_and_general_notes> 1. Consider the Semantic Analyzer results ..., 2. Generate tests ... </rules_and_general_notes>
</instruction>

<semantic_analysis_results> {"control_flow": <>, "data_flow": <>, "io": <>, "lib_api": <>, ...} </semantic_analysis_results>

<final_response_format> {"is_equivalent": <>, "explanation": <>, "tests": <>, "patch": <>, ...} </final_response_format>
```

Fig. 4. Prompt structure of Test Generator and Repair Agent.

outputs an overall equivalence verdict, a set of tests in both the source and target language, and a translation patch if the agent believed the translation was not equivalent.

3.4 Verdict Agent

The final component of MatchFixAgent is the Verdict Agent, which produces a definitive assessment of the translation's correctness by synthesizing the information from the previous two stages. The Verdict Agent takes as input the semantic analysis report and the test execution + repair report. It leverages another LLM agent to consolidate these results into a final verdict. This agent's primary job is to (1) confirm the results of the Test Generator & Repair Agent, and (2) to produce a condensed summary of the results, which is useful for end-users.

4 Evaluation

We evaluate MatchFixAgent and answer the following research questions:

- **RQ1:** *Effectiveness of MatchFixAgent in Translation Validation.* To what extent can MatchFixAgent automatically validate repository-level code translation? How does MatchFixAgent compare against existing validation systems?
- **RQ2:** *Effectiveness of MatchFixAgent in Translation Repair*. How effective is MatchFixAgent in repairing translation bugs from real-world projects? How does the repair component compare with existing tools?
- **RQ3:** *Development Cost and Adaptability.* How does the development cost and adaptability of MatchFixAgent compare to existing work? Does it work with other LLMs and agents?
- **RQ4:** *Ablation Study.* How do the semantic analyzer and test generator components contribute to MatchFixAgent's effectiveness? Can a standalone code agent perform similarly to MatchFixAgent?

4.1 Experimental Setup

4.1.1 Benchmark. We evaluate MatchFixAgent on benchmarks used in prior work on automated repository-level translation. Each benchmark problem is a translation pair: the source function and the corresponding translation. The task for each benchmark problem is to give a verdict on the functional equivalence of pairs in two different PLs, and repair translation in case of equivalence.

[2]Some projects in Skel (e.g., `colorsys`) are part of a bigger project [66]. The reported Star and Fork numbers belong to that bigger project.

Table 1. Details of benchmarks[2] from existing techniques used in MatchFixAgent. **LoC:** Lines of code in the source project.

Tool	Project	Source Language	Target Language	Total # Trans. Pairs	LoC	Test Coverage (%)	Stars	Forks
Oxidizer [86]	checkdigit [71]	Go	Rust	29	428	86.2	111	8
	go-edlib [9]			24	639	100	517	27
	histogram [16]			19	314	68.4	176	31
	nameparts [51]			15	413	100	43	5
	stats [18]			53	1241	98.1	2989	170
	textrank [7]			52	1132	100	217	22
AlphaTrans [29]	cli [20]	Java	Python	273	37841	100	372	201
	csv [21]			235	33072	100	392	278
	fileupload [22]			192	3567	100	246	185
	validator [23]			646	41605	95.5	216	164
Skel [73]	bst [3]	Python	JavaScript	19	123	100	203000	47000
	colorsys [65]			8	120	96.3	67900	32300
	heapq [67]			22	189	78.1	67900	32300
	html [68]			44	684	65.3	67900	32300
	mathgen [78]			81	735	93.8	711	183
	rbt [4]			27	366	88.1	203000	47000
	strsim [40]			64	654	20.8	1014	125
	toml [50]			72	1206	62.5	1126	192
RustRepoTrans [47]	charset [32]	Python	Rust	33	4231	100	672	56
	deltachat [13]	C		125	23116	98.4	306	28
	iceberg-java [24]	Java		25	592793	100	7700	2700
	iceberg-py [25]	Python		44	49746	97.7	805	327
	crypto-c [19]	C		20	5922	100	36	15
	crypto-java [26]	Java		97	110261	100	2	7
Total				**2219**	**910398**	**89.6**	**627351**	**195624**

Our subjects are open-source repository-level translations with equivalence verdicts available[3] from the peer-reviewed literature[4]. We make selections across a diverse set of PL pairs.

Table 1 summarizes our subject translation pairs. We collect subjects from three recent repository-level code translation techniques [29, 73, 86]. We did not include Syzygy [53] because its artifact does not provide a validation system for individual functions (it only provides end-to-end tests). These works generated translations of real-world open source GitHub projects, and performed equivalence validation at the individual function level. Since AlphaTrans [29] is evaluated on a large number of functions, we randomly sample 1346 of its total 4643 translation pairs[5].

To demonstrate the adaptability of MatchFixAgent to more PL pairs, we also collect subjects from RustRepoTrans [47], a benchmark consisting of human-written translations into Rust and unit tests. We exclude translation pairs collected from the libp2p [38] projects in RustRepoTrans due to the presence of non-deterministic flaky tests that may result in false negatives, i.e., functional inequivalence while translation is correct, unfairly biasing comparison in favor of MatchFixAgent. In total, we collect 2,219 translation pairs with over 900K lines of code from 24 projects and in 6 different PL pairs.

4.1.2 LLMs. Major software engineering leaderboards [8, 55] have shown that Claude Sonnet [54] outperforms other proprietary LLMs, such as OpenAI GPT-4o [63] and Google Gemini Pro [58]. Therefore, we use Anthropic's `Claude 3.7 Sonnet` [54] and Claude Code (1.0.51) [56] as the main LLM and agent in all our experiments. To show the adaptability of MatchFixAgent to different LLMs and agentic frameworks, we repeat a subset of experiments using OpenAI `o4-mini-2025-04-16` [45]

[3]We exclude rule-based transpilers as they do not propose a validation mechanism, i.e., their translation is (theoretically) correct by construction.

[4]This criterion excludes techniques such as RustMap [11] and C2SaferRust [44].

[5]AlphaTrans translated both application and test code. In this work, we only included the application code translation pairs. While reviewing their artifacts, we also noted 11 translation pairs to be dead code and excluded them.

Table 2. Effectiveness of MatchFixAgent in translation validation compared to existing techniques. **EQ:** Equivalent, **NEQ:** Not Equivalent, **VF:** Validation Failure, **Agreement**: number and percentage of translation pairs where MatchFixAgent's and the existing tool's verdicts agree, **Disagreement**: percentage of disagreements ruled in favor of **Tool** and MatchFixAgent (**Ours**). **VF**s are excluded from **Agreement** and **Disagreement** calculations.

Tool	Project	Total # Trans. Pairs	Tool Validation			MatchFixAgent			Agreement	Disagreement	
			EQ	NEQ	VF	EQ	NEQ	VF		Tool	Ours
Oxidizer	checkdigit	29	21 (72.4)	8 (27.6)	0 (0)	22 (75.9)	7 (24.1)	0 (0)	24 (82.8)	0.0	100
	go-edlib	24	18 (75)	6 (25)	0 (0)	16 (66.7)	8 (33.3)	0 (0)	14 (58.3)	11.1	88.9
	histogram	19	12 (63.2)	7 (36.8)	0 (0)	11 (57.9)	7 (36.8)	1 (5.3)	8 (44.4)	20.0	80.0
	nameparts	15	9 (60)	6 (40)	0 (0)	12 (80)	3 (20)	0 (0)	12 (80)	33.3	66.7
	stats	53	38 (71.7)	14 (26.4)	1 (1.9)	37 (69.8)	16 (30.2)	0 (0)	35 (67.3)	0.0	100
	textrank	52	40 (76.9)	12 (23.1)	0 (0)	34 (65.4)	18 (34.6)	0 (0)	28 (53.8)	42.9	57.1
Total		**192**	**138 (71.9)**	**53 (27.6)**	**1 (0.5)**	**132 (68.8)**	**59 (30.7)**	**1 (0.5)**	**121 (63.7)**	**15.9**	**84.1**
AlphaTrans	cli	273	210 (76.9)	24 (8.8)	39 (14.3)	210 (76.9)	60 (22)	3 (1.1)	176 (76.2)	30.0	70.0
	csv	235	97 (41.3)	61 (26)	77 (32.8)	185 (78.7)	49 (20.9)	1 (0.4)	108 (68.8)	30.0	70.0
	fileupload	192	19 (9.9)	1 (0.5)	172 (89.6)	144 (75)	48 (25)	0 (0)	16 (80)	25.0	75.0
	validator	646	247 (38.2)	103 (15.9)	296 (45.8)	483 (74.8)	163 (25.2)	0 (0)	225 (64.3)	20.0	80.0
Total		**1346**	**573 (42.6)**	**189 (14)**	**584 (43.4)**	**1022 (75.9)**	**320 (23.8)**	**4 (0.3)**	**525 (69.3)**	**26.5**	**73.5**
Skel	bst	19	19 (100)	0 (0)	0 (0)	14 (73.7)	5 (26.3)	0 (0)	14 (73.7)	20.0	80.0
	colorsys	8	8 (100)	0 (0)	0 (0)	7 (87.5)	1 (12.5)	0 (0)	7 (87.5)	0.0	100
	heapq	22	19 (86.4)	3 (13.6)	0 (0)	12 (54.5)	10 (45.5)	0 (0)	13 (59.1)	50.0	50.0
	html	44	40 (90.9)	2 (4.5)	2 (4.5)	35 (79.5)	9 (20.5)	0 (0)	33 (78.6)	66.7	33.3
	mathgen	81	77 (95.1)	4 (4.9)	0 (0)	65 (80.2)	16 (19.8)	0 (0)	67 (82.7)	50.0	50.0
	rbt	27	26 (96.3)	0 (0)	1 (3.7)	23 (85.2)	4 (14.8)	0 (0)	22 (84.6)	75.0	25.0
	strsim	64	50 (78.1)	0 (0)	14 (21.9)	56 (87.5)	8 (12.5)	0 (0)	44 (88)	40.0	60.0
	toml	72	37 (51.4)	10 (13.9)	25 (34.7)	49 (68.1)	22 (30.6)	1 (1.4)	33 (71.7)	40.0	60.0
Total		**337**	**276 (81.9)**	**19 (5.6)**	**42 (12.5)**	**261 (77.4)**	**75 (22.3)**	**1 (0.3)**	**233 (79.3)**	**46.5**	**53.5**
RustRepoTrans	charset	33	20 (60.6)	13 (39.4)	0 (0)	14 (42.4)	19 (57.6)	0 (0)	25 (75.8)	75.0	25.0
	deltachat	125	54 (43.2)	69 (55.2)	2 (1.6)	39 (31.2)	84 (67.2)	2 (1.6)	92 (76)	90.0	10.0
	iceberg-java	25	9 (36)	16 (64)	0 (0)	3 (12)	16 (64)	6 (24)	15 (78.9)	100	0.0
	iceberg-py	44	15 (34.1)	28 (63.6)	1 (2.3)	11 (25)	29 (65.9)	4 (9.1)	35 (89.7)	100	0.0
	crypto-c	20	16 (80)	4 (20)	0 (0)	7 (35)	13 (65)	0 (0)	11 (55)	66.7	33.3
	crypto-java	97	39 (40.2)	58 (59.8)	0 (0)	30 (30.9)	67 (69.1)	0 (0)	86 (88.7)	100	0.0
Total		**344**	**153 (44.5)**	**188 (54.7)**	**3 (0.9)**	**104 (30.2)**	**228 (66.3)**	**12 (3.5)**	**264 (80.2)**	**87.5**	**12.5**
Total		**2219**	**1140 (51.4)**	**449 (20.2)**	**630 (28.4)**	**1519 (68.5)**	**682 (30.7)**	**18 (0.8)**	**1143 (72.8)**	**39.3**	**60.7**

and Codex [64] (§4.4). To support future research and external validation, MatchFixAgent logs the inputs, intermediate agent interactions, tool execution results, and outputs of the LLM, and supports visualizing and inspecting these logs. MatchFixAgent terminates within the budget of 1,000 seconds. We empirically set this timeout after analyzing the execution time of 300 samples.

4.1.3 Competing Validation & Repair Tools. We compare MatchFixAgent's validation technique with the automated validation techniques proposed by Skel [73], Oxidizer [86], and AlphaTrans [29]. Except RustRepoTrans, other approaches do not explicitly report the repair results, as the repair process is interleaved with translation in the loop. For those techniques [29, 73, 86], we compare MatchFixAgent repair results with their final translation success.

4.1.4 MatchFixAgent Implementation. We implement our structure-based semantic analysis on top of Tree-Sitter [72], as it supports 165+ languages, including six PLs we target in this study. For running tests and validating patches, MatchFixAgent uses Rust 1.87.0 [70], Python 3.12.9 [69], Java 21.0.7 [60], Node 22.16.0 [62], GCC 7.3.1 [57], and Go 1.24.4 [59].

4.2 RQ1: Effectiveness of MatchFixAgent in Translation Validation

To answer this RQ, we run MatchFixAgent on each translation pair to obtain an equivalence verdict, and compare it with the verdict of existing tools (§4.2.1). We then investigate disagreements between verdicts to determine which verdict is correct (§4.2.2).

4.2.1 Translation Validation. The columns under **Tool Validation** and **MatchFixAgent** in Table 2 summarize the equivalence verdicts for the competing validation tool and MatchFixAgent, respectively. There are three types of equivalence verdicts: (1) Equivalent (**EQ**) indicating the source function and its translation are equivalent, (2) Not Equivalent (**NEQ**) indicating they are inequivalent, and (3) Validation Failure (**VF**) indicating that the tool failed to provide a verdict. Competing tools can fail to provide a verdict if (1) the source project did not have unit tests covering the function, or (2) the competing tool's language interoperability mechanism crashes before providing verdict. MatchFixAgent can fail to provide a verdict if the timeout limit is reached. The **Agreement** column shows the number and percentage where both MatchFixAgent's and others verdicts agree or disagree. The **Disagreement** column shows the result of our human investigation on disagreements. The **Tool** sub-column shows the percentage of disagreements where the existing tool's verdict was correct, and **Ours** sub-column shows the same metric for MatchFixAgent. On average, MatchFixAgent takes 309 seconds to produce a verdict, with an average cost of $1.22 and a total cost of $2,710.45.

These results demonstrate effectiveness of MatchFixAgent in providing an equivalence verdict: MatchFixAgent provides verdicts for 99.7% of translation pairs from AlphaTrans and Skel, whereas these techniques report verdicts for only 56.6% and 87.5% of their studied translation pairs, respectively. In addition, MatchFixAgent's verdicts show a high level of agreement with all prior work, ranging from 63.7% to 80.2%.

4.2.2 Analysis of the Disagreements. Both MatchFixAgent and existing validation approaches are prone to false positives (i.e. the verdict is equivalent but the translation is not) and false negatives. To determine false positives and false negatives, we perform a manual investigation.

We first categorize disagreements into two cases: D_1, where MatchFixAgent produced a *not equivalent* verdict and the other disagreed; and D_2, where MatchFixAgent produced an *equivalent* verdict and the other disagreed. Due to large number of studied translation pairs, we randomly sample five instances of both D_1 and D_2 from each of the 24 projects. If a project had fewer than five disagreements, we considered all instances without sampling. Two authors[6] independently reviewed disagreements to verify whether *the not equivalent verdict* was correct (by MatchFixAgent in D_1 and by others in D_2). If the not equivalent verdict is correct, the reviewer rules it in favor of the tool that said not equivalent, otherwise they rule it favor of the tool that said equivalent. During the investigation, 3 disagreements with Oxidizer were due to the tool's function mocking not being enabled, and were unable to enable it. In addition, 11 disagreements with RustRepoTrans were due to the correct translation not being "1:1", or in other words, the correct translation is not functionally equivalent to the source function. These disagreements would have been unfairly ruled in favor of MatchFixAgent, and so were filtered out, leaving us with 145 disagreement cases ($D_1 = 92$, $D_2 = 53$). When the reviewer's resolution conflicted (18.6% of cases), they met with each other and agreed on the final resolution.

The **Disagreement** columns in Table 2 summarize the result of our human investigation. The **Tool** column shows the percentage of disagreements ruled in favor of the existing validation tool, and the **Ours** columns shows the percentage ruled in favor of MatchFixAgent. The disagreement resolutions show that MatchFixAgent 's verdicts are often more accurate than existing automated validation tools. MatchFixAgent's verdicts are significantly more accurate than Oxidizer and AlphaTrans— disagreements are ruled in favor of MatchFixAgent in 84.1% and 73.5% of cases, respectively. Compared to Skel, MatchFixAgent shows similar accuracy (53.5%). On RustRepoTrans, MatchFixAgent's accuracy fairs worse (12.5%), which is due to the relative complexity of its translation pairs.

[6]The selected authors have total experience of 17 years in academia and 4.5 years in industry.

Existing validation approaches produced 88 incorrect verdicts, 80 of which fell into three categories: (1) **Inadequate Unit Tests (42 of cases)**, (2) **Excessively Strict Equivalence Definition (11 of cases)**, and (3) **Language Interoperability Bug (27 of cases)**. A language interoperability bug means that the tool's process for converting concrete inputs in the source language to the target language did not preserve equivalence. For example, Oxidizer's conversion of `Go runes` (which represent a Unicode character) to `Rust chars` resulted in different Unicode characters.

On the benchmarks associated with automated validation tools (Skel, Oxidizer, AlphaTrans), MatchFixAgent produced 36 incorrect verdicts, 34 of which fell into three categories: (1) **Hallucination (23 cases)**, (2) **Inadequate Unit Tests (4 cases)** (meaning MatchFixAgent missed an input that would demonstrate inequivalence), (3) **Infeasible Input (7 cases)**. An infeasible input means that the LLM discovered an input where the source function and translation produce different outputs, but the input can never occur when the project is used as intended. Infeasible inputs often involve directly initializing private class/struct members, or calling private helper methods in unintended ways.

On RustRepoTrans's benchmarks, MatchFixAgent produced 21 incorrect verdicts. Two of the main causes are similar to the other benchmarks: (1) **Hallucination (7 cases)** and (2) **Inadequate Unit Tests (6 cases)**. The increased rates of these two causes are due to the relative complexity and size of RustRepoTrans's projects. The other major cause is **Excessively Strict Equivalence Definition (6 cases)**. As previously mentioned, RustRepoTrans's translations include refactors to make the translation more idiomatic, which creates ambiguity around the proper definition equivalence.

4.2.3 Representative Examples of Incorrect Verdicts. The following code snippet shows an example of **Inadequate Unit Tests** on a translation pair from Oxidizer. The functions both calculate the edit distance between two strings. The functions are not equivalent because `Go`'s `len()` function counts Unicode characters, whereas `Rust`'s `.len()` function counts bytes. Oxidizer incorrectly validates the `Rust` translation as equivalent because the source project's unit tests do not cover non-ASCII inputs. However, MatchFixAgent successfully generates a test with non-ASCII inputs demonstrating they are not equivalent.

```
------------------- GO SOURCE CODE -------------------
func LCSEditDistance(str1 string, str2 string)
 ↪  int {

    // ... if conditions ...
    lcs := LCS(s1, s2)
    return (len([]rune(s1)) - lcs) +
    ↪  (len([]rune(s2)) - lcs)
}
```

```
------------------- RUST TRANSLATION -------------------
pub fn lcs_edit_distance(str1: &str, str2: &str) ->
 ↪  Result<i32> {
    // ... if conditions ...
    let lcs_len = lcs(str1, str2)?;
    let edit_distance = (str1.len() as i32 - lcs_len)
    ↪  + (str2.len() as i32 - lcs_len);
    Ok(edit_distance)
}
```

The next code snippet demonstrates an example of an **Infeasible Input** taken from the `textrank` project. The below `Go` function inserts a value into the map `ranks.SentenceMap` (respectively, `ranks.sentence_map` for the `Rust` translation). MatchFixAgent discovers that these functions return different values when the map is initially `{ 1 : "SomeString" }` (the `Go` returns `0` while the `Rust` returns 1). However, when the `textrank` is used properly via its public interface, this initial state for the map cannot occur. The map will always contain keys from 1 to n, where n is the number of map entries. Under this precondition, the functions are equivalent.

```
------------------- GO SOURCE CODE -------------------
func addSentence(ranks *Rank, sentence
 ↪  ParsedSentence) int {,
    ranks.SentenceMap[len(ranks.SentenceMap)] =
    ↪  sentence.GetOriginal()
    return len(ranks.SentenceMap) - 1

}
```

```
------------------- RUST TRANSLATION -------------------
pub(crate) fn add_sentence(ranks: &mut Rank, sentence:
 ↪  ParsedSentence) -> Result<i32, Error> {
    let sentence_id = ranks.sentence_map.len() as i32;
    ranks.sentence_map.insert(sentence_id,
    ↪  sentence.original.clone());
    Ok(sentence_id)
}
```

The next code snippet demonstrates an example of an **Excessively Strict Equivalence Definition** taken from the `deltachat-core` project. Both the `C` and `Rust` function retrieve a field `blobdir`.

MatchFixAgent states that these are not equivalent because (1) the Rust function does not perform a null check, and (2) the C function returns a copy of blobdir whereas the Rust returns a reference. While this is true, the translation follows Rust's idioms, and a developer would not care about these differences. Rust's type system prevents blobdir from ever being null, and the Rust translation returns an *immutable* reference, preventing the caller from modifying the return value.

```c
------------------- C SOURCE CODE --------------------
char* dc_get_blobdir(const dc_context_t* context) {
    if (context==NULL ||
    ↪  context->magic!=DC_CONTEXT_MAGIC) {
        return dc_strdup(NULL);
    }
    return dc_strdup(context->blobdir);
}
```

```rust
------------------- RUST TRANSLATION -------------------
pub fn get_blobdir(&self) -> &Path {


    &self.inner.blobdir


}
```

Summary. Compared with existing automated validation approaches (Skel, Oxidizer, and AlphaTrans), MatchFixAgent is more reliable at producing equivalence verdicts. MatchFixAgent provides verdicts for 99.2% of benchmarks, compared to 71.6% for existing approaches. In addition, MatchFixAgent's verdicts are more accurate than existing approaches. MatchFixAgent's verdicts agree on 72.8% of benchmarks, and on 60.7% of disagreements, human investigation shows MatchFixAgent's verdict is correct. On the more challenging benchmarks of RustRepoTrans, MatchFixAgent's verdict still show a high agreement rate, but the disagreement cases reveal for improvement.

4.3 RQ2: Effectiveness of MatchFixAgent in Translation Repair

We investigate the effectiveness of MatchFixAgent in automated repair of translation bugs, and its ability to improve code coverage of existing projects. Table 3 shows the results of this research question. To fairly evaluate the effectiveness of existing tools and MatchFixAgent in translation repair, we extract a subset of translations where both techniques generated a patch.

In total, $\frac{265}{2219}$ (11.9%) buggy translations were considered for our study. To validate patches, we used original project tests that previously failed on buggy translations, and only considered a patch correct when all failing tests passed. We did not use generated tests by MatchFixAgent to avoid bias in our evaluation. Column *Tool Repaired* shows the number of buggy translations repaired by existing techniques. Only $\frac{49}{265}$ (18.5%) bugs have been repaired by prior techniques. Except for RustRepoTrans [47], other code translation tools failed to generate correct patches to repair translation bugs. Skel [73] reprompts the same LLM for repairing bugs, but then requires a user to manually provide a fix. AlphaTrans [29] and Oxidizer [86] generates patches in the loop, however, no effectiveness were reported in their papers, and we could not repair any translation bugs using their tools. Moreover, patches by AlphaTrans and Oxidizer could not be validated mostly because of limitations in their validation system. For example, the following code snippets show instance 11 from the project commons-validator in AlphaTrans. The GraalVM-based validation system in AlphaTrans does not validate this translation as functionally equivalent, although our manual investigation indicates that the Python translation is correct. Therefore, the limitation in AlphaTrans is mostly due to its validation system being unable to validate LLM patches.

```java
------------------ JAVA SOURCE CODE ------------------
public boolean isOn(long flag) {
    return (this.flags & flag) == flag;
}
```

```python
----------------- PYTHON TRANSLATION -----------------
def isOn(self, flag: int) -> bool:

    return (self.__flags & flag) == flag
```

Column *MatchFixAgent Repaired* shows the number of translation bugs successfully repaired by our approach. In total, MatchFixAgent repaired $\frac{134}{265}$ (50.6%) of translation bugs, 32.1% more than existing reprompting-based techniques. Given the limitations in validation system of AlphaTrans discussed earlier, we manually investigate and validate patches from this tool. The following code

Table 3. Effectiveness of MatchFixAgent in translation repair compared against existing techniques. **NEQ:** Not Equivalent, **NR:** Not Reported/Repaired.

Tool	Project	# Total Trans. Pair	Tool NEQ ∩ MatchFixAgent NEQ	Tool Repaired	MatchFixAgent Repaired	Disagreement Repaired	Coverage (Improvement) %
Oxidizer	checkdigit	29	5 (17.2)	NR	5 (100)	2 (100)	86.2 (0)
	go-edlib	24	2 (8.3)	NR	1 (50)	4 (100)	100 (0)
	histogram	19	2 (10.5)	NR	1 (50)	2 (66.7)	68.4 (0)
	nameparts	15	3 (20)	NR	1 (33.3)	0 (0)	100 (0)
	stats	53	6 (11.3)	NR	6 (100)	5 (100)	100 (↑**1.9**)
	textrank	52	3 (5.8)	NR	3 (100)	2 (100)	100 (0)
Total		**192**	**21 (10.9)**	**0 (0)**	**17 (81)**	**15 (93.8)**	**92.4 (↑0.3)**
AlphaTrans	cli	273	9 (3.3)	NR	7 (77.8)	3 (100)	100 (↑**6.2**)
	csv	235	20 (8.5)	NR	16 (80)	2 (100)	100 (↑**11.9**)
	fileupload	192	0 (0)	-	-	2 (100)	98.9 (↑**78.1**)
	validator	646	34 (5.3)	NR	23 (67.6)	3 (100)	99.3 (↑**36.8**)
Total		**1346**	**63 (4.7)**	**0 (0)**	**46 (73)**	**10 (100)**	**99.6 (↑33.3)**
Skel	bst	19	0 (0)	-	-	4 (100)	100 (0)
	colorsys	8	0 (0)	-	-	1 (100)	100 (0)
	heapq	22	2 (9.1)	NR	1 (50)	3 (100)	100 (0)
	html	44	1 (2.3)	NR	0 (0)	2 (100)	100 (↑**4.5**)
	mathgen	81	3 (3.7)	NR	1 (33.3)	2 (66.7)	100 (0)
	rbt	27	0 (0)	-	-	1 (100)	100 (↑**3.7**)
	strsim	64	0 (0)	-	-	3 (100)	100 (↑**21.9**)
	toml	72	5 (6.9)	NR	3 (60)	3 (100)	100 (↑**34.7**)
Total		**337**	**11 (3.3)**	**0 (0)**	**5 (45.5)**	**19 (95)**	**100 (↑8.1)**
RustRepo Trans	charset	33	12 (36.4)	5 (41.7)	7 (58.3)	1 (100)	100 (0)
	deltachat	125	60 (48)	10 (16.7)	11 (18.3)	1 (100)	100 (↑**1.6**)
	iceberg-java	25	12 (48)	1 (8.3)	1 (8.3)	0 (0)	100 (0)
	iceberg-py	44	25 (56.8)	4 (16)	6 (24)	0 (0)	100 (↑**2.3**)
	crypto-c	20	4 (20)	1 (25)	2 (50)	1 (100)	100 (0)
	crypto-java	97	57 (58.8)	28 (49.1)	39 (68.4)	0 (0)	100 (0)
Total		**344**	**170 (49.4)**	**49 (28.8)**	**66 (38.8)**	**3 (100)**	**100 (↑0.6)**
Total		**2219**	**265 (11.9)**	**49 (18.5)**	**134 (50.6)**	**47 (95.9)**	**98.1 (↑ 8.5)**

snippets demonstrate instance `276` from the project `commons-validator` in AlphaTrans which its reprompting-based repairing could not generate a correct patch. By contrast, MatchFixAgent successfully repairs this translation bug with the help of its Library Analyzer (§3.2.4). The report generated by this semantic analyzer indicate `"... the standard Python datetime.date class does not have a SHORT attribute ..."` which is correct. The Test Generator & Repair Agent (§3.3) then leverages this analysis and successfully generate a patch by replacing `SHORT` with constant `3`.

```
------------------ JAVA SOURCE CODE ------------------
public static DateValidator DateValidator1() {
    return new DateValidator(true, DateFormat.SHORT);
}
```

```
------------------ PYTHON TRANSLATION ------------------
def DateValidator1() -> DateValidator:
- return DateValidator(True, datetime.date.SHORT)
+ return DateValidator(True, 3)
```

Column *Disagreement Repaired* show the number of disagreements from RQ1 (§4.2) which MatchFixAgent determined as *not equivalent* and successfully generated a correct patch. Of the 49 disagreements resolved in favor of MatchFixAgent, we further asked our manual investigators if the generated patch was correct or not. On average, $\frac{47}{49}$ (95.9%) of patches were validated as correct. Notice that this column cannot be directly compared with existing techniques, because they validated disagreements as functionally correct and did not generate any patches. Moreover, Column *Coverage* indicate the overall coverage for subject projects and the total improvement as a result of tests generated by MatchFixAgent. The generated tests help improve code coverage by 8.5% (from 89.6% to 98.1%). Project `commons-fileupload` from AlphaTrans sees the most improvement (78.1%), with most of its translated fragments now validated with MatchFixAgent tests.

Summary. MatchFixAgent patches fix 50.6% of translation bugs, 32.1% more than existing repair techniques. Its generated target PL tests addresses the inadequate test suite limitation in prior work and help improve code coverage by 8.5%.

4.4 RQ3: Development Cost and Adaptability

In this RQ, we demonstrate the development cost of MatchFixAgent (§4.4.1) and its adaptability with other LLMs and agentic frameworks (§4.4.2).

4.4.1 Development Cost. Figure 5 shows the development cost of MatchFixAgent against existing techniques. We define cost as the tool's total lines of code (LoC). As illustrated in the figure, developing MatchFixAgent is cheap and the initial version only consists of 1,650 LoC, supporting 6 different PLs. Its dependence only on the Tree-Sitter [72] parser makes it easy to support more languages using only 280 LoC. By contrast, other tools, such as, Skel [73], AlphaTrans [29], and Oxidizer [86] only support *one* PL pair and require significant engineering effort to adapt to more languages. More precisely, MatchFixAgent is ×2.3, ×6.6, and ×11.6 cheaper than Skel, AlphaTrans, and Oxidizer, respectively. The static nature of MatchFixAgent makes it cost-effective and scalable, achieving better performance and revealing major limitations in existing tools.

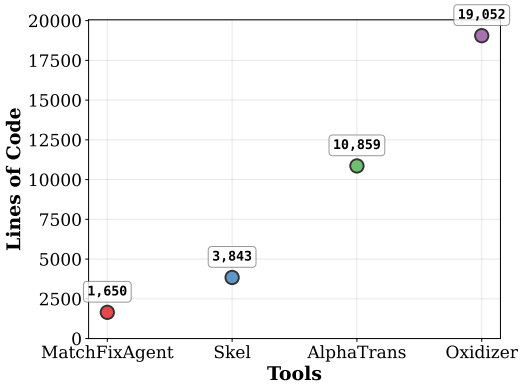

Fig. 5. Development cost of MatchFixAgent compared against existing tools.

4.4.2 Adaptability. The architecture of MatchFixAgent is largely independent of any specific LLM or agent framework, making it easy to extend and integrate with more LLMs. In this research question, we investigate the extent to which replacing Anthropic's Claude 3.7 Sonnet with OpenAI `o4-mini-2025-04-16` [45], and Claude Code with Codex [64] agent, impacts the performance of MatchFixAgent. Due to the limited cost budget, we randomly sampled 96 instances from all subject projects. While sampling, we controlled for equal contribution of equivalent and non-equivalent translations, resulting in 58 and 38 samples for each, respectively.

Figure 6 illustrates the results of this study. Our analysis indicates that MatchFixAgent with Claude Code and Codex agrees 73% of the time against existing validation systems. Moreover, we also analyzed both agents' behavior in terms of problem understanding and finding a solution. Our investigation of agent trajectories (footprint of agent actions) shows OpenAI's Codex makes fewer actions to explore the codebase, and attempts to provide a decision faster. By contrast, Anthropic's Claude Code agent first plans and reasons thoroughly about its tasks, specifically called `TODO List` by the agent, and then takes specific actions to perform each of its planned tasks.

Summary. MatchFixAgent is up to ×11.6 times cheaper to develop than existing techniques. Also, it can be easily adapted to other LLMs and agent frameworks, e.g., OpenAI.

4.5 RQ4: Ablation Study

We present two ablation studies to highlight the contribution of the semantic analyzer and the test generator agent in MatchFixAgent.

4.5.1 Impact of Semantic Analyzer and Test Generator Agent. To investigate the importance of semantic analyzer and test generator agent in MatchFixAgent, we evaluated a standalone baseline agent that uses the same LLM and agent framework, e.g., `Claude Sonnet 3.7` [54] and Claude Code [56]. In order to perform a controlled study, we created a sample from the original dataset

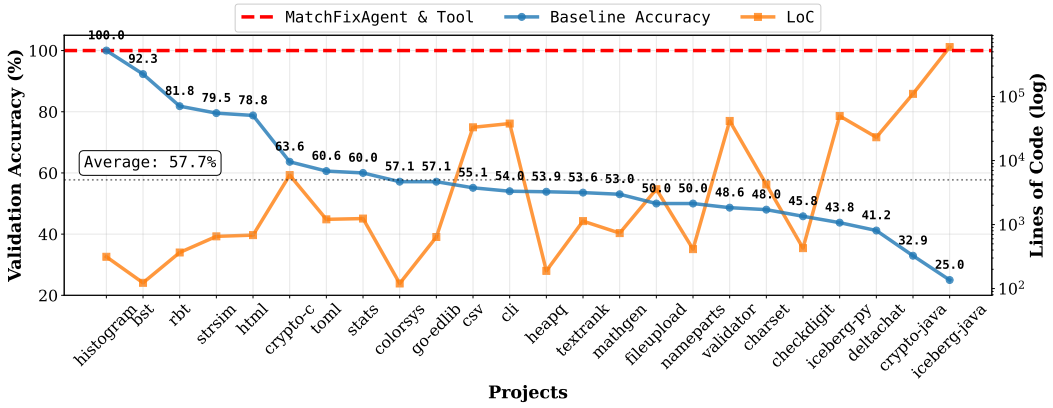

Fig. 7. Impact of semantic analyzer and test generator agent in MatchFixAgent. A standalone baseline agent struggles validating translations as the projects grow in size.

where existing tools and MatchFixAgent verdicts agree with each other, meaning we collected all non-dispute instances. In total, 1,091 instances, 862 equivalent, and 229 non-equivalent translations were selected.

Figure 7 shows the result of this ablation study. Accuracy indicates the ratio of baseline verdicts that agree with the existing tool and MatchFixAgent. On average, the validation accuracy drops by 42.3%, illustrating the importance of MatchFixAgent's semantic analyzer and test generator components. Across all projects, the baseline agent only reproduced MatchFixAgent and tool results in the `histogram` project from Oxidizer [86], achieving 100% validation correctness. For the remaining projects, the agent's accuracy dropped as low as 25% in `iceberg-java`, which is the project with the largest number of lines of code. This study confirms that without semantic guidance and in-the-loop test generation, the LLM agents often fail to distinguish subtle differences and validate functional equivalence, especially when projects become larger and more restrictive languages, such as `Rust` are involved in the translation.

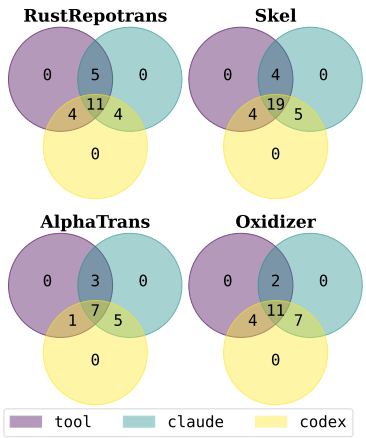

Fig. 6. Agreement and dispute cases between tool validation system and MatchFixAgent with Claude and Codex agents.

4.5.2 Impact of Semantic Analyzer. We perform another study by removing only the semantic analysis results when prompting the test generator and verdict agents. We use the same set with 1,091 instances from the previous ablation to perform this study. Figure 8 illustrates the result of our second ablation. The results indicate that the performance of MatchFixAgent significantly drops by 39.7% without the six semantic analyses. Furthermore, we observe that the test generator agent without semantic analyzer is more costly and on average spends 3.9% (66.3K instead of 63.7K), 6.2% (136K instead of 128K), 4.2% (9.4M instead of 9.0M), and 7.5% (61.28h instead of 56.71h) more turns/interactions, input tokens, output tokens, and time, respectively. This is expected since the purpose of semantic analysis is to provide the necessary context for the test generator agent, without the need for the agent to explore and figure it out for

itself. Since each semantic analysis is a *single* call to the LLM, it is much cheaper than the agents that iteratively analyze the codebase.

Summary. Removing the semantic analyzer and test generator significantly drops the effectiveness of MatchFixAgent by 42.3%. Moreover, removing the semantic analyzer alone decreases the accuracy by 39.7%, and increases the test generator cost, on average, by 5.4%.

5 Related Work

Translation Validation and Repair. Existing automated translation validation techniques either rely on test execution [29, 47, 49, 53, 81, 85, 86, 88], formal methods [43, 82], or fuzzing [17] for translation validation. Abid et al. [1] and AlphaTrans [29] leverage GraalVM [46] and language interoperability to execute code in both source and target PL for translation validation. Other tools like Oxidizer [86] and Syzygy [53] instrument programs to extract input-output pairs and use in target PL for validation. Skel [73] validates translations through test execution by converting Python tests to JavaScript simply through string replacement. This suffices since source Python tests are simple function calls and value comparisons. For automated translation repair, most tools [29, 47, 49, 53, 73, 85, 86] rely on simple reprompting of LLMs with execution feedback, which has proven ineffective. Specifically, AlphaTrans [29] performs independent reprompting of suspicious fragments based on execution trace, while Skel [73] requires a user for manually repairing translation bugs.

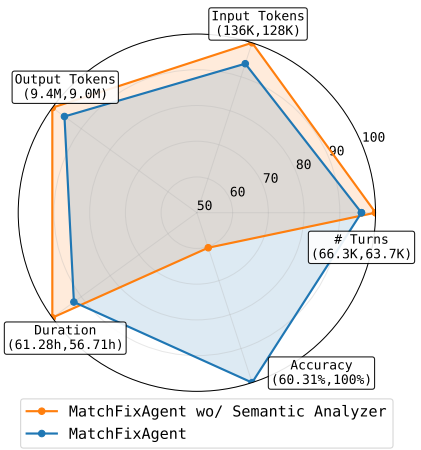

Fig. 8. Removing the semantic analyzer decreases the effectiveness of MatchFixAgent, while increasing token consumption, number of turns, and processing time.

LLM Agents. With the increasing prominence of agent-based frameworks [39, 79], recent research and industrial efforts have turned towards leveraging these frameworks to address various software engineering tasks [14, 33, 84]. SWE-agent [83] introduces a specialized agent-computer interface (ACI) facilitating agent interaction with code repositories via file reading, editing, and execution of bash commands. AutoCodeRover [87], which provides LLM agents with specialized code-search APIs, enables iterative retrieval and localization of code segments associated with bugs. SpecRover [52] enhances AutoCodeRover by emphasizing specification inference, generating function summaries, and providing targeted feedback at crucial agent execution stages. Agentless [80] further shows simple LLM agents can fix real-world bugs without using excessive tools and model complex environment behavior. Besides these state-of-the-art frameworks, numerous additional agent-based approaches exist both in open-source [2, 10, 48, 61, 77] and commercial products [5, 15, 74].

6 Threats to Validity

Similar to prior techniques, MatchFixAgent comes with some limitations and threats to the validity. In this section, we discuss how we mitigated various threats.

Internal Validity. There are two main threats to internal validity. First, we only run experiments once. Since LLMs are inherently non-deterministic, running experiments again may produce different results. While it is highly *likely* some individual equivalence verdicts and repair results

would change if experiments were run again, it is highly *unlikely* the aggregate metrics we report would change significantly given the large number of translation samples we use (2,219 pairs). Second, our human investigation does not assess ground truth equivalence. We only assess whether an inequivalent verdict was correct, but we do not analyze the correctness of equivalent verdicts. While this means we don't have any measure of true accuracy of MatchFixAgent, we still can claim MatchFixAgent is more accurate than existing automated validation techniques.

External Validity. One main external threat is the generalizability of our approach. Our validation and repair system is very generic and can be extended to more PL pairs with minimal engineering effort. Also, the majority of tools that we used, for example, Tree-Sitter [72] can support a large set of PLs. To mitigate external validity, we built the initial version of MatchFixAgent with six PLs.

Construct Validity. In order to minimize construct validity, MatchFixAgent is built on well-known and rigorously tested tools, e.g., Tree-Sitter, Claude Code, and Codex.

7 Conclusion

In this work, we presented MatchFixAgent, a *language-agnostic* neuro-symbolic technique which combines the power of program analysis and LLM agents for autonomous repository-level translation validation and repair. MatchFixAgent performs various semantic analyses—including control-flow and data-flow analyses—to systematically generate targeted tests, enabling demonstration of functional equivalence or detection of semantic bugs. Through rigorous evaluation on multiple benchmarks and different language-pairs, we show MatchFixAgent is effective in automatically validating and repairing translation pairs. It further generates high-quality reports that can be used by end-users for better understanding of translated programs and the validation process. Our manual investigation of generated reports reveals significant limitation of existing techniques in code translation validation and repair. MatchFixAgent is cost-effective and scalable, it only requires on average 280 lines of code to support more PLs and it validates each instance in approximately 5 minutes. To the best of our knowledge, MatchFixAgent is the first approach that can effectively validate and repair translations in repository-level across multiple PLs.

References

[1] Muhammad Salman Abid, Mrigank Pawagi, Sugam Adhikari, Xuyan Cheng, Ryed Badr, Md Wahiduzzaman, Vedant Rathi, Ronghui Qi, Choiyin Li, Lu Liu, Rohit Sai Naidu, Licheng Lin, Que Liu, Asif Zubayer Palak, Mehzabin Haque, Xinyu Chen, Darko Marinov, and Saikat Dutta. 2024. GlueTest: Testing Code Translation via Language Interoperability. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 612–617. doi:10.1109/ICSME58944.2024.00061

[2] Aider AI. 2025. AI pair programming in your terminal. https://aider.chat/

[3] The Algorithms. 2025. All Algorithms implemented in Python. https://github.com/TheAlgorithms/Python/blob/master/data_structures/binary_tree/binary_search_tree_recursive.py

[4] The Algorithms. 2025. All Algorithms implemented in Python. https://github.com/TheAlgorithms/Python/blob/master/data_structures/binary_tree/red_black_tree.py

[5] Amazon. 2025. Amazon Q Developer. https://aws.amazon.com/q/developer/

[6] Anthropic. 2025. Building Effective AI Agents. https://www.anthropic.com/engineering/building-effective-agents.

[7] David Belicza. 2025. TextRank on Go. https://github.com/DavidBelicza/TextRank

[8] The SWE bench Team. 2025. SWE-bench Leaderboard. https://www.swebench.com/

[9] Hugo Bollon. 2025. Go-edlib : Edit distance and string comparison library. https://github.com/hbollon/go-edlib

[10] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. Repairagent: An autonomous, llm-based agent for program repair. *arXiv preprint arXiv:2403.17134* (2024).

[11] Xuemeng Cai, Jiakun Liu, Xiping Huang, Yijun Yu, Haitao Wu, Chunmiao Li, Bo Wang, Imam Nur Bani Yusuf, and Lingxiao Jiang. 2025. RustMap: Towards Project-Scale C-to-Rust Migration via Program Analysis and LLM. *arXiv preprint arXiv:2503.17741* (2025).

[12] Sung-Hyuk Cha. [n. d.]. Comprehensive survey on distance/similarity measures between probability density functions. *City* 1, 2 ([n. d.]), 1.

[13] Delta Chat. 2025. Delta.Chat C-Library with e2e chat-over-email functionality & Python bindings. https://github.com/deltachat/deltachat-core
[14] Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. 2024. Introducing SWE-bench Verified. https://openai.com/index/introducing-swe-bench-verified/
[15] Cognition. 2025. Introducing Devin, the first AI software engineer. https://cognition.ai/blog/introducing-devin
[16] Vivid Cortex. 2025. gohistogram - Histograms in Go. https://github.com/VividCortex/gohistogram
[17] Hasan Ferit Eniser, Hanliang Zhang, Cristina David, Meng Wang, Maria Christakis, Brandon Paulsen, Joey Dodds, and Daniel Kroening. 2024. Towards translating real-world code with llms: A study of translating to rust. *arXiv preprint arXiv:2405.11514* (2024).
[18] Montana Flynn. 2025. Stats - Golang Statistics Package. https://github.com/montanaflynn/stats
[19] The Apache Software Foundation. 2025. AMCL - Apache Milagro Crypto Library. https://github.com/apache/incubator-milagro-crypto-c
[20] The Apache Software Foundation. 2025. Apache Commons CLI. https://github.com/apache/commons-cli
[21] The Apache Software Foundation. 2025. Apache Commons CSV. https://github.com/apache/commons-csv
[22] The Apache Software Foundation. 2025. Apache Commons FileUpload. https://github.com/apache/commons-fileupload
[23] The Apache Software Foundation. 2025. Apache Commons Validator. https://github.com/apache/commons-validator
[24] The Apache Software Foundation. 2025. Apache Iceberg. https://github.com/apache/iceberg
[25] The Apache Software Foundation. 2025. Apache PyIceberg. https://github.com/apache/iceberg-python
[26] The Apache Software Foundation. 2025. MCJL - Milagro Crypto Java Library. https://github.com/apache/incubator-milagro-java
[27] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. 2008. Automated whitebox fuzz testing.. In *Ndss*, Vol. 8. 151–166.
[28] André Hora, Romain Robbes, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, and Marco Tulio Valente. 2015. How do developers react to API evolution? The Pharo ecosystem case. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 251–260. doi:10.1109/ICSM.2015.7332471
[29] Ali Reza Ibrahimzada, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. 2025. AlphaTrans: A Neuro-Symbolic Compositional Approach for Repository-Level Code Translation and Validation. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE109 (June 2025), 23 pages. doi:10.1145/3729379
[30] Suman Jain and Inderveer Chana. 2015. Modernization of Legacy Systems: A Generalised Roadmap. In *Proceedings of the Sixth International Conference on Computer and Communication Technology 2015* (Allahabad, India) *(ICCCT '15)*. Association for Computing Machinery, New York, NY, USA, 62–67. doi:10.1145/2818567.2818579
[31] Pooyan Jamshidi, Aakash Ahmad, and Claus Pahl. 2013. Cloud Migration Research: A Systematic Review. *IEEE Transactions on Cloud Computing* 1, 2 (2013), 142–157. doi:10.1109/TCC.2013.10
[32] Jawah. 2025. Charset Normalizer: Truly universal encoding detector in pure Python. https://github.com/jawah/charset_normalizer
[33] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=VTF8yNQM66
[34] Ravi Khadka, Belfrit V. Batlajery, Amir M. Saeidi, Slinger Jansen, and Jurriaan Hage. 2014. How do professionals perceive legacy systems and software modernization?. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 36–47. doi:10.1145/2568225.2568318
[35] Musawwer Khan, Islam Ali, Wasif Nisar, Muhammad Qaiser Saleem, Ali S Ahmed, Haysam E Elamin, Waqar Mehmood, and Muhammad Shafiq. 2022. Modernization Framework to Enhance the Security of Legacy Information Systems. *Intelligent Automation & Soft Computing* 32, 1 (2022), 543–555. doi:10.32604/iasc.2022.016120
[36] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2123–2138.
[37] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? *Empirical Softw. Engg.* 23, 1 (Feb. 2018), 384–417. doi:10.1007/s10664-017-9521-5
[38] libp2p. 2025. The Python implementation of the libp2p networking stack. https://github.com/libp2p/py-libp2p
[39] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024. Large language model-based agents for software engineering: A survey. *arXiv preprint arXiv:2409.02977* (2024).
[40] ZhouYang Luo. 2025. A library implementing different string similarity and distance measures using Python. https://github.com/luozhouyang/python-string-similarity/tree/master/strsimpy
[41] Nickil Maveli, Antonio Vergari, and Shay B Cohen. 2025. What can Large Language Models Capture about Code Functional Equivalence?. In *Findings of the Association for Computational Linguistics: NAACL 2025*, Luis Chiruzzo,

Alan Ritter, and Lu Wang (Eds.). Association for Computational Linguistics, Albuquerque, New Mexico, 6865–6903. doi:10.18653/v1/2025.findings-naacl.382

[42] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. 2009. *Levenshtein Distance: Information theory, Computer science, String (computer science), String metric, Damerau?Levenshtein distance, Spell checker, Hamming distance.* Alpha Press.

[43] Vikram Nitin, Rahul Krishna, and Baishakhi Ray. 2024. Spectra: Enhancing the code translation ability of language models by generating multi-modal specifications. *arXiv preprint arXiv:2405.18574* (2024).

[44] Vikram Nitin, Rahul Krishna, Luiz Lemos do Valle, and Baishakhi Ray. 2025. C2SaferRust: Transforming C Projects into Safer Rust with NeuroSymbolic Techniques. *arXiv preprint arXiv:2501.14257* (2025).

[45] OpenAI. 2025. Introducing OpenAI o3 and o4-mini. https://openai.com/index/introducing-o3-and-o4-mini/.

[46] Oracle. 2025. GraalVM. https://www.graalvm.org.

[47] Guangsheng Ou, Mingwei Liu, Yuxuan Chen, Xin Peng, and Zibin Zheng. 2024. Repository-level Code Translation Benchmark Targeting Rust. *arXiv preprint arXiv:2411.13990* (2024).

[48] Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. 2025. RepoGraph: Enhancing AI Software Engineering with Repository-level Code Graph. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=dw9VUsSHGB

[49] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 82, 13 pages. doi:10.1145/3597503.3639226

[50] Will Pearson. 2025. Python lib for TOML. https://github.com/uiri/toml/tree/master/toml

[51] James Polera. 2025. gonameparts. https://github.com/polera/gonameparts

[52] Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. 2025. SpecRover: Code Intent Extraction via LLMs. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 963–974. doi:10.1109/ICSE55347.2025.00080

[53] Manish Shetty, Naman Jain, Adwait Godbole, Sanjit A Seshia, and Koushik Sen. 2024. Syzygy: Dual Code-Test C to (safe) Rust Translation using LLMs and Dynamic Analysis. *arXiv preprint arXiv:2412.14234* (2024).

[54] The Anthropic Team. 2025. Claude. https://www.anthropic.com/claude

[55] The BigCodeBench Team. 2025. BigCodeBench Leaderboard. https://bigcode-bench.github.io/

[56] The Claude Code Team. 2025. Claude Code. https://github.com/anthropics/claude-code

[57] The GNU Team. 2025. C Compiler. https://gcc.gnu.org/

[58] The Google Deepmind Team. 2025. Gemini Pro. https://deepmind.google/models/gemini/pro/

[59] The Go Language Team. 2025. Go Language. https://go.dev/

[60] The Java Language Team. 2025. Java Language. https://www.java.com/en/

[61] The Moatless Tools Team. 2025. Moatless Tools. https://github.com/aorwall/moatless-tools

[62] The NodeJS Team. 2025. NodeJS. https://nodejs.org/en

[63] The OpenAI Team. 2025. GPT-4o. https://openai.com/index/hello-gpt-4o/

[64] The OpenAI Team. 2025. OpenAI Codex CLI. https://github.com/openai/codex

[65] The Python Team. 2025. Conversion functions between RGB and other color systems. https://github.com/python/cpython/blob/3.13/Lib/colorsys.py

[66] The Python Team. 2025. CPython. https://github.com/python/cpython.

[67] The Python Team. 2025. Heap queue algorithm (a.k.a. priority queue). https://github.com/python/cpython/blob/3.13/Lib/heapq.py

[68] The Python Team. 2025. A parser for HTML and XHTML. https://github.com/python/cpython/blob/3.13/Lib/html/parser.py

[69] The Python Language Team. 2025. Python Language. https://www.python.org/

[70] The Rust Language Team. 2025. Rust Language. https://www.rust-lang.org/

[71] Osamu Tonomori. 2025. Checkdigit. https://github.com/osamingo/checkdigit

[72] Tree-Sitter. 2025. Tree-Sitter Library. https://tree-sitter.github.io/tree-sitter/

[73] Bo Wang, Tianyu Li, Ruishi Li, Umang Mathur, and Prateek Saxena. 2025. Program Skeletons for Automated Program Translation. *Proc. ACM Program. Lang.* 9, PLDI, Article 184 (June 2025), 25 pages. doi:10.1145/3729287

[74] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2025. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=OJd3ayDDoF

[75] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. 2020. An Empirical Study of Usages, Updates and Risks of Third-Party Libraries in Java Projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 35–45. doi:10.1109/ICSME46990.2020.00014

[76] Anjiang Wei, Jiannan Cao, Ran Li, Hongyu Chen, Yuhui Zhang, Ziheng Wang, Yuan Liu, Thiago SFX Teixeira, Diyi Yang, Ke Wang, et al. 2025. EquiBench: Benchmarking Large Language Models' Understanding of Program Semantics via Equivalence Checking. *arXiv preprint arXiv:2502.12466* (2025).

[77] Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I Wang. 2025. SWE-RL: Advancing LLM Reasoning via Reinforcement Learning on Open Software Evolution. *arXiv preprint arXiv:2502.18449* (2025).

[78] Luke Weiler. 2025. Basic Math. https://github.com/lukew3/mathgenerator/blob/main/mathgenerator/basic_math.py

[79] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, Zhangyue Yin, Shihan Dou, Rongxiang Weng, Wenjuan Qin, Yongyan Zheng, Xipeng Qiu, Xuanjing Huang, Qi Zhang, and Tao Gui. 2025. The rise and potential of large language model based agents: a survey. *Science China Information Sciences* 68, 2 (17 Jan 2025), 121101. doi:10.1007/s11432-024-4222-0

[80] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Demystifying LLM-Based Software Engineering Agents. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE037 (June 2025), 24 pages. doi:10.1145/3715754

[81] Pengyu Xue, Linhao Wu, Zhen Yang, Chengyi Wang, Xiang Li, Yuxiang Zhang, Jia Li, Ruikai Jin, Yifei Pei, Zhaoyan Shen, Xiran Lyu, and Jacky Wai Keung. 2025. ClassEval-T: Evaluating Large Language Models in Class-Level Code Translation. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA063 (June 2025), 24 pages. doi:10.1145/3728940

[82] Aidan ZH Yang, Yoshiki Takashima, Brandon Paulsen, Josiah Dodds, and Daniel Kroening. 2024. VERT: Verified Equivalent Rust Transpilation with Large Language Models as Few-shot Learners. *arXiv preprint arXiv:2404.18852* (2024).

[83] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=mXpq6ut8J3

[84] John Yang, Carlos E Jimenez, Alex L Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R Narasimhan, Diyi Yang, Sida Wang, and Ofir Press. 2025. SWE-bench Multimodal: Do AI Systems Generalize to Visual Software Domains?. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=riTiq3i21b

[85] Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024. Exploring and Unleashing the Power of Large Language Models in Automated Code Translation. *Proc. ACM Softw. Eng.* 1, FSE, Article 71 (July 2024), 24 pages. doi:10.1145/3660778

[86] Hanliang Zhang, Cristina David, Meng Wang, Brandon Paulsen, and Daniel Kroening. 2025. Scalable, Validated Code Translation of Entire Projects using Large Language Models. *Proc. ACM Program. Lang.* 9, PLDI, Article 212 (June 2025), 26 pages. doi:10.1145/3729315

[87] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1592–1604. doi:10.1145/3650212.3680384

[88] 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* (Clarion Hotel Trondheim, Trondheim, Norway) *(FSE Companion '25)*. Association for Computing Machinery, New York, NY, USA, 162–173. doi:10.1145/3696630.3728542