Enhancing Repository-Level Code Translation using RAG and Mock Testing

Kaiyao Ke
kaiyaok2@illinois.edu
University of Illinois Urbana-Champaign
Urbana, Illinois, USA

Abstract

We identify two key limitations of AlphaTrans, a state-of-the-art neuro-symbolic approach for repository-level code translation and validation. First, its simplistic type translation mechanism often produces incorrect type mappings and requires manual intervention. Second, its reliance on GraalVM for in-isolation validation demands extensive configuration to support new types. We address these challenges by enhancing type resolution using contextual and documentation-aware prompting, and by introducing a mocking-based validation framework that automatically verifies outputs and side effects of method-level translations. The implementation of our solution is publicly available at https://github.com/kaiyaok2/CS510-Project-Team15

Keywords

Code Translation, Mock Testing, RAG

ACM Reference Format:
Kaiyao Ke. 2025. Enhancing Repository-Level Code Translation using RAG and Mock Testing. In . ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

We highlight two key limitations of AlphaTrans [6] and describe how our tool addresses each of them.

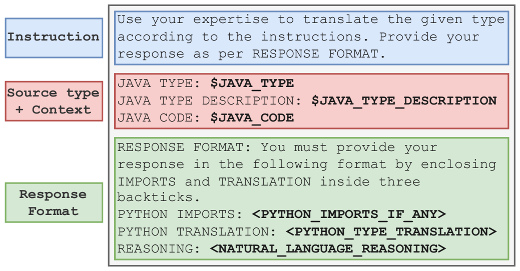
AlphaTrans Overview. AlphaTrans [6] is a state-of-the-art neuro-symbolic approach that combines static analysis with the generative capabilities of large language models (LLMs) to automate repository-level code translation and validation. To avoid exceeding context limits, translation is performed at the method level. Before translation begins, AlphaTrans analyzes the source project and constructs a structural skeleton in the target language. This skeleton uses type hints to maintain consistency and serves as a framework for compositional translation. Equivalent types across source and target languages are suggested by an LLM, which is then prompted to translate individual code fragments—method by method—in reverse call order, progressively "filling in" the skeleton. Beyond syntactic translation, AlphaTrans performs two forms of validation: runtime verification using GraalVM and execution of translated unit tests. While AlphaTrans achieves strong results in repository-level code translation, it encounters two major limitations:

Challenge 1: Type Translation. Before translating individual code fragments, AlphaTrans must resolve type mappings between source and target languages, as types affect how fragments are ultimately translated. However, its current approach is simplistic and relies heavily on manual effort to build a universal type map. For example, it prompts the LLM with only the raw Java type (e.g., `java.util.Collections$UnmodifiableCollection`) to generate an equivalent Python type. This often results in incorrect mappings—such as translating to a mutable `List` in Python—failing to preserve the semantics of immutability. To address this, we enhance type resolution by prompting the LLM with (i) the source type, (ii) code context where the type is used, and (iii) online-crawled type documentation. Furthermore, unlike AlphaTrans, we support import resolution by specifying a response format in our prompts. We also require the model to explain its reasoning, improving interpretability and accuracy of type resolution.

Challenge 2: Isolated Translation Validation. AlphaTrans validates translated code fragments in isolation using GraalVM [10], leveraging cross-language interoperability. However, this approach requires significant manual configuration to support previously unseen types. For instance, its glue-code generation does not scale well and must be adapted manually for types like `Map<String, String>`. In contrast, we propose a mocking-based validation workflow that automatically supports all resolved types via the type translation module. Our method replaces all callees invoked by the focal method with mocks, ensuring in-isolation validation. At the method-invocation level, it verifies not only the returned output, but also all possible side effects including exceptions, parameter mutations, instance field changes, and static field modifications—enabling more comprehensive and automated validation.

2 Context-Aware RAG-Based Type Resolution

Automated type resolution is a longstanding and complex challenge [5, 12], traditionally addressed through symbolic rule-based methods [1, 4, 7], and more recently with large language models (LLMs) [6, 9, 11]. Unlike AlphaTrans [6] and other repository-level code translation systems [9, 11], which typically rely on minimal prompting or manual inspection of translated types, we leverage both API documentation and code usage context to guide type translation. Our prompting strategy uses Retrieval-Augmented Generation (RAG) [8], combining the raw type, code fragment, and retrieved documentation to help the LLM identify semantically equivalent types in the target language.


Instruction
Source type + Context
Response Format

```
Use your expertise to translate the given type
according to the instructions. Provide your
response as per RESPONSE FORMAT.

JAVA TYPE: $JAVA_TYPE
JAVA TYPE DESCRIPTION: $JAVA_TYPE_DESCRIPTION
JAVA CODE: $JAVA_CODE

RESPONSE FORMAT: You must provide your
response in the following format by enclosing
IMPORTS and TRANSLATION inside three
backticks.
PYTHON IMPORTS: <PYTHON_IMPORTS_IF_ANY>
PYTHON TRANSLATION: <PYTHON_TYPE_TRANSLATION>
REASONING: <NATURAL_LANGUAGE_REASONING>
```

Figure 1: Context-Aware Type Resolution prompt template.

Additionally, we prompt the LLM to include a natural language explanation for the chosen translation, which is later used to improve transparency and support fragment-level translation. Figure 1 illustrates the prompt template used for Context-Aware Type Resolution, which includes both API and code-based context.

Algorithm 1: Context-Aware Type Resolution

```
   Inputs: Subject Projects $Ps$
   Output: Context-Aware Type Map $CTM$
1  foreach $project \in Ps$ do
2      $types \leftarrow$ getProjectTypes($project$);
3      foreach $type \in types$ do
4          $doc \leftarrow$ crawlTypeDocumentation($type$);
5          $code \leftarrow$ getTypeCodeSnippet($type$);
6          $translation \leftarrow$ translate&Validate($type, doc, code$);
7          if $unsuccessful(translation)$ then
8              $translation \leftarrow$ resolveGlobally($type$);
9          $CTM[project][type] \leftarrow translation$;
10 return $CTM$
```

Algorithm 1 outlines the workflow for resolving types. It begins by extracting all source-language types from the project (lines 1–2). Application-specific types are resolved without LLM involvement, while standard library or third-party types are processed through the full translation pipeline. For each unresolved type, we retrieve API documentation, collect the surrounding code context, and prompt the LLM to generate a valid target-language type along with import statements and reasoning (lines 3–6).

To guard against hallucinations—e.g., references to nonexistent Python types—we perform syntactic and runtime validation of the LLM's response. If validation fails, a feedback loop is used to refine the query. When translation fails despite multiple attempts, the system searches other translated projects for inferred mappings (lines 7–8). As a fallback, unresolved types are conservatively mapped to `object`.

Finally, all validated translations and their corresponding rationales are stored in the *Context-Aware Type Map*, a centralized structure used by the mocking-based validation pipeline we'll introduce in the next section.

Algorithm 2: Mocking-Based Validation Workflow

```
   Input  : Java Unit Tests $J$, Repository Method Set $R$
   Output : Mock-Based Python Test Suite $T$
1  $T \leftarrow [\,]$;
2  foreach $test \in J$ do
3      $L \leftarrow$ runWithAspectJandCustomSerializer($test$);
4      $focal\_methods \leftarrow$ extractAllInvocations($L$);
5      foreach $focal \in focal\_methods$ do
6          $C \leftarrow$ getDirectCallees($focal, R$);
7          $test\_case \leftarrow$ initTestFile($focal$);
           // Mock direct callees of the focal method
8          foreach $c \in C$ do
9              $M_c \leftarrow$ {
10                 output$_c$,
11                 exception$_c$,
12                 paramModifications$_c$,
13                 instanceModifications$_c$
14                 staticFieldsModifications$_c$
15             };
16             injectMock(test_case, c, $M_c$);
           // Set up input of the focal method
17         $S_{init} \leftarrow$ {staticFields, focalInstance, focalParams};
18         injectCall(test_case, $focal$, $S_{init}$);
           // Verify focal method's output & side effects
19         $Expected \leftarrow$ {
20             output$_{focal}$,
21             exception$_{focal}$,
22             paramModifications$_{focal}$,
23             instanceModifications$_{focal}$,
24             staticFieldsModifications$_{focal}$
25         };
26         injectVerifications(test_case, $Expected$);
27         append(T, test_case);
28 return $T$
```

3 Mocking-based In-isolation Validation

The mocking-based validation pipeline automates the generation of Python tests to verify the correctness of translated Java methods in complete isolation. Algorithm 2 shows its overall workflow. It begins by analyzing AspectJ-generated execution logs to extract detailed input-output behavior and side effects for each method invocation in the call chain of a Java unit test (lines 1–5). Based on this information, it constructs a Python test case that calls one translated focal method directly (lines 17–18), while mocking all other repository-level methods it depends on (lines 6–16). Each mock simulates the original method's return values and side effects, including updates to global static fields and thrown exceptions. The test then reconstructs the initial program state, invokes the translated focal method, and verifies the correctness of return values, instance and parameter mutations, and global state changes using recursive equality checks (lines 19–26). All Python objects used for mocking or equality checks are constructed based on a custom serialization of Java objects, which supports both standard library types and repository-defined classes. This enables reproducible, in-isolation validation of each translated method's semantic equivalence to its original Java counterpart.

3.1 A Custom Serialization Workflow for Java Objects

This workflow revolves around a `CustomToStringConverter` class that serializes arbitrary Java objects into JSON. For each object, this process captures the essential data needed to reconstruct an equivalent Python object for mocking or equality checks. It distinguishes between internal repository types and external library classes by comparing the class's package prefix with its own, enabling fine-grained introspection for internal types and grouped handling for common external types (e.g., applying the same strategy to all `Map` implementations).

Standard library types are grouped for consistent handling. Primitive types are serialized by recording their type name and string value. Compound types—such as collections, maps, and properties—are serialized element-wise through recursive traversal, preserving structural integrity. The intuition is to record all information needed to construct an equivalent Python object. For example, types like `HashMap` are flattened into JSON arrays of key-value pairs; for I/O-related types like `ByteArrayInputStream`, internal buffers and positions are extracted using reflection without consuming the stream, thus avoiding side effects. Mutable objects are tagged with identity information to allow accurate reference tracking during reconstruction in Python. Special cases, such as enums (excluding types like `TimeUnit`), are handled via reflection to extract meaningful internal state; when such details are unavailable, the ordinal is used as a fallback.

For repository-defined classes, the converter introspects fields across the class hierarchy. It distinguishes between static and instance fields, skips synthetic and transient members, and includes visibility metadata when relevant. To resolve naming conflicts or ambiguity, it records the declaring class where appropriate. Edge cases also receive tailored handling. For instance, repository-level subclasses of I/O classes (e.g., `BufferedReader`, `FilterInputStream`) are treated with dedicated logic that accounts for both inherited behavior and additional custom fields. For iterators from inner classes, the converter detects outer references (via the synthetic `this$0` field), serializing the enclosing context while avoiding infinite recursion through cached identity tracking. This ensures consistent, cycle-free serialization even in the presence of deeply nested or self-referential structures.

3.2 AspectJ-Based Runtime Interception

The system employs an AspectJ-based instrumentation layer to capture detailed behavioral traces of method executions. This mechanism intercepts all public method calls within repository-defined classes and records input-output (IO) pairs, side effects on mutable parameters, changes to instance state (for non-static methods), and any global state modifications resulting from static field mutations. The AspectJ advice uses a broad pointcut targeting repository packages, ensuring comprehensive coverage while avoiding synthetic or standard library method invocation.

Upon method entry, the interceptor captures a deep copy of all input parameters using the same serialization logic employed by the `CustomToStringConverter`. If the method is an instance method, it also records a snapshot of the `this` object's internal state prior to execution. Static fields from all repository-defined classes are introspected via reflection and their values deep-copied to capture a consistent view of global state before invocation.

After the method completes—or throws an exception—the system captures a second snapshot of the parameters, the return value (or thrown exception), the updated receiver state (if applicable), and the new global static field state. Comparing pre- and post-execution snapshots allows the system to detect in-place mutations, not only to method arguments and instance state but also to the wider program context via static fields. All captured information is serialized to a trace structure, which can later be used to reconstruct method behavior, assess purity, and support Python test generation.

To maintain fidelity and avoid semantic interference, intercepted executions are proceeds unmodified after AspectJ logging. For instance, exceptions are re-thrown after being recorded, preserving the program's natural control flow. The advice layer is designed to be non-invasive and dynamically pluggable, making it suitable for integration into Maven-based test executions without modifying the source code under test. Its design supports both whole-program capture and filtered tracing of specific modules, depending on the analysis goals.

3.3 Deserialization and Equality Verification Workflow

The system reconstructs Python objects from JSON-based logs generated during Java program execution using a custom recursive deserialization pipeline. Each JSON structure represents an object captured by the AspectJ tracer and serialized via `CustomToStringConverter`. The deserialization process begins by resolving the object's declared Java type, which is mapped to the corresponding Python class using a universal type map. This ensures that both repository-specific classes and Java types (e.g., `java.util.*`) are mapped to appropriate Python standard library equivalents or preloaded translated repository-level classes.

The deserializer traverses the JSON recursively, converting primitive values (e.g., integers, floats, booleans) directly, while handling compound structures like lists, tuples, dictionaries, and object fields through type-specific deserialization. To preserve pointer (or reference) identity and manage aliasing, a global dictionary of memory addresses tracks all instantiated objects, keyed by their memory addresses recorded at serialization time. This prevents duplication and allows the handling of cyclic references.

For objects representing custom Java classes, the deserialization workflow reconstructs Python instances by introspecting both static and instance fields. Java-style visibility rules are simulated by using Python name mangling for private and protected fields (e.g., transforming `private int x` in class `Foo` to `_Foo__x`). For types that inherit from Java standard library classes, the system ignores Java-specific fields to reduce noise. Special edge cases are handled on a case-by-case basis. For example, Enum types are reconstructed by recreating both `_name_` and `_value_` attributes, ensuring compatibility with Python's `Enum` semantics. Exception instances are created using low-level constructors (`__new__`) to bypass argument enforcement, while still initializing with meaningful messages when present.

Once reconstruction is complete, a recursive equality checker performs a deep semantic comparison between the expected and actual Python objects. This function supports type-specific structural equality for standard library types, including primitives, collections, enums, and I/O streams. designs one specific workflow for each group of "similar" types. For example, all I/O stream values (e.g., `BytesIO`, `StringIO`) are compared by their internal buffer content. For custom objects, the checker compares the `__dict__` of two objects recursively to ensure that all instance attributes match. Special care is taken to normalize numerical tolerances (e.g., accounting for precision loss when converting `timedelta` values) and to apply a logical comparison heuristic for string equivalence (e.g., standard string representation of a Java `HashMap` and its Python equivalent `dict` have different format) when simple string equality fails.

This entire pipeline ensures accurate behavioral equivalence between Java traces and their Python replays, while faithfully preserving object identity, mutability, and visibility semantics.

3.4 Automated Mock-Based Test Generation

A mock test generation workflow is designed to automate the in-isolation validation of each method invoked by executing Java unit tests. For each method invocation, a mock test is generated to run its translated Python method to verify if its I/O and side effects match the expected behavior recorded during the execution of the Java unit test. The main workflow is divided into several steps:

(1) **Reading the Full Log from AspectJ Interception**: The workflow reads the logs (containing all I/O pairs and side effects of each method invocation) to locate the relevant test method. It searches for a focal method by scanning through the lines of the file. It retrieves all relevant subsections in the log, including information about the I/O pairs and side effects of the method itself (used to compare against results of real execution) and information about the I/O pairs and side effects of other repository-level methods it directly calls (used to set up mocks to replace actual invocation of these methods, ensuring that the validation of one method translation operates purely in isolation).

(2) **Mock Decorators Insertion**: For each method that requires mocking, the workflow adds mock decorators. It checks whether the method is a constructor (e.g., `<init>` in Java) or a normal method, and applies the appropriate mock decorator to replace the constructor or method with a mocked version. The decorators are inserted above the test method with correct indentation, ensuring that the patching is done only once per method. If the invocation is the focal method, mocking is bypassed, with the translated real method called directly. Otherwise, mocking is used to simulate the output and side effects of the specific method. Side effects include updating static fields, handling argument changes, and simulating exceptions if required.

(3) **Method Body Construction**: For each method, the script generates the necessary logic inside the test method. This includes:
   - Handling static fields initialization, which mimics the Java program state before invoking the focal method.
   - Handling construction of instances (if instance method) and parameters (if any).
   - Invoke the focal method. Notice that other repository-level methods called by the focal method are already replaced by mocks as a result of the earlier mock decorators.
   - Verifying changes in the current instance (if an instance method) and the mutable input parameters (if any), and ensuring that the mock behaves as the original method.
   - Verifying static field changes (if any) in all repository-level classes and ensuring that the mock behaves as the original method.
   - Asserting that the return values, if any, match the expected results using recursive comparison.
   - Handling exceptions thrown by the method and asserting that the correct exception is raised, if any.

(4) **Final File Write**: For each focal method invocation, once the decorators, parameters, and method bodies are updated, the workflow writes all relevant lines to a new test file. This enables verifying the correctness of one method translation without invoking any other repository-level methods it depends on. When running these mock tests, the actual setup of Python objects and equivalence checks are handled by the deserialization and equality verification workflow introduced earlier.

4 Results

We integrated both RAG-based context-aware type resolution and mock-based in-isolation method translation validation into the original AlphaTrans framework [6]. As shown in Table 1, replacing raw LLM prompting with RAG-enhanced type resolution improves the validation success rate of method translations for both `commons-cli`[2] and `commons-fileupload`[3]. This demonstrates that RAG-based context retrieval—applied during type inference for parameter types, return types, field declarations, and local variables—enables more accurate and consistent Python translations of Java functions. Internally, AlphaTrans [6] decomposes Java programs into smaller, manageable fragments and constructs an equivalent skeleton project in the target language, using type hints (for method parameter types, method returned object types, and class / instance types) to maintain semantic consistency. The LLM then translates each fragment ("filling in the skeleton") in reverse call order. Enhancing this pipeline with RAG improves type resolution, which in turn leads to more functionally accurate and consistent type hints for LLMs to produce more accurate translations.

Furthermore, the integration of mocking-based validation ensures that every method fragment can be tested in isolation. On the other hand, even with extensive manual effort to write glue code for each type, GraalVM [10] still fails on certain fragments—highlighting its inability to support fully automated validation.

Overall, our contributions make repository-level code translation more robust, by improving type inference with RAG, and more scalable, by enabling automated testing through mocking.

Table 1: Comparison of Fragment Validatability and Type Resolution Accuracy

Project	Non-Validatable Methods	Validation Success %
`cli`	GraalVM: 8.9% Mocking: **0%**	Raw LLM: 76.92% RAG: **81.71%**
`fileupload`	GraalVM: 20.0% Mocking: **0%**	Raw LLM: 76.03% RAG: **90.91%**

5 Using the Software

Our toolchain integrates RAG-based type resolution and a mocking-based validation pipeline to improve the accuracy and reliability of Java-to-Python translation. The workflow consists of the following main steps:

- **Type Translation**: Java types extracted from the logs are resolved to Python types using a Retrieval-Augmented Generation (RAG) mechanism implemented in `translate_type.py`.
- **Java Instrumentation**: We insert custom Java classes (e.g., `CustomToStringConverter`, `LoggingAspect`) into the original Java project to intercept and log runtime method calls and side effects.
- **Log Generation**: The instrumented Java project is built with Maven and executed, generating logs that describe input/output values, method invocations, and internal field states.
- **Python Mocking and Testing**: Using a series of scripts such as `log_parser.py` and `script.py`, the logs are parsed and translated into Python mock-based test files, enabling per-method validation of translated functions.

To simplify adoption, we provide detailed setup and execution instructions in the README file at:

https://github.com/kaiyaok2/CS510-Project-Team15

This includes guidance for modifying a Java project's build configuration, injecting aspect-oriented logging code, executing the pipeline, and running validation tests on the translated Python code.

6 Next Steps

Looking ahead, we plan to scale our mocking-based testing pipeline to a broader set of open-source Java projects. We are also extending our type resolution framework to support *one-to-many* type mappings, where a single Java type may correspond to multiple valid Python types depending on the context. For example, a Java method returning an `ArrayList<Integer>` could be translated to either a `List[int]` or a `numpy` array in Python.

As a next step, we aim to integrate third-party libraries such as `numpy` into our type system and validation logic. We are actively collaborating with the original AlphaTrans authors to incorporate these enhancements, with the goal of submitting a joint paper to the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE 2025).

References

[1] Dante Broggi and Yi Liu. 2023. On the Interoperability of Programming Languages via Translation. In *2023 Congress in Computer Science, Computer Engineering, & Applied Computing (CSCE)*. IEEE, 2579–2585.

[2] The Apache Software Foundation. 2025. Apache Commons CLI. https://github.com/apache/commons-cli https://github.com/apache/commons-cli.

[3] The Apache Software Foundation. 2025. Apache Commons FileUpload. https://github.com/apache/commons-fileupload https://github.com/apache/commons-fileupload.

[4] Dony George, Priyanka Girase, Mahesh Gupta, Prachi Gupta, and Aakanksha Sharma. 2010. Programming language inter-conversion. *International Journal of Computer Applications* 1, 20 (2010), 68–74.

[5] Giovani Guizzo, Jie M Zhang, Federica Sarro, Christoph Treude, and Mark Harman. 2024. Mutation analysis for evaluating code translation. *Empirical Software Engineering* 29, 1 (2024), 19.

[6] Ali Reza Ibrahimzada, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Repository-level compositional code translation and validation. *arXiv preprint arXiv:2410.24117* (2024).

[7] Kevin Lano and Hanan Siala. 2024. Using model-driven engineering to automate software language translation. *Automated Software Engineering* 31, 1 (2024), 20.

[8] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems* 33 (2020), 9459–9474.

[9] 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).

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

[11] 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).

[12] Andrey A Terekhov and Chris Verhoef. 2000. The realities of language conversions. *IEEE Software* 17, 6 (2000), 111–124.