Multi-Agent Approach to Code Translation

Parsa Youssefpour
Supervised by: Prof. Eldan Cohen
Department of Mechanical And Industrial Engineering
University of Toronto

Abstract

Accurate and semantically faithful code translation remains a key challenge in software engineering, particularly for organizations migrating legacy systems written in low-resource programming languages. While large language models (LLMs) have improved automated translation, they frequently produce syntactically invalid or semantically incorrect outputs due to brittle reasoning and a lack of verification mechanisms.

This work investigates whether multi-agent systems can enhance translation quality by decomposing the task into modular subtasks such as code analysis, translation, evaluation, and regeneration. In this report, four architectures were implemented and evaluated: a baseline one shot GPT-4o translator and three multi-agent systems incorporating Analyzer, Evaluator, and Regenerator Agents, with one hybrid variant combining components from two of the architectures.

Evaluation on a subset of the GeeksforGeeks Python–Java dataset shows that multi-agent systems improve Python-to-Java accuracy by up to 78%, a 32% absolute gain over the baseline, primarily by reducing compilation errors through structured reasoning and iterative validation. For Java-to-Python translation, where baseline accuracy is already high, only the hybrid variant outperforms the baseline.

These results highlight the benefits of explicit structured reasoning, AST analysis integration, modular task separation, and repeated code evaluation for improving translation reliability, supporting the potential of multi-agent systems as a viable framework for industrial code migration.

Introduction

Accurate translation of code between programming languages remains a critical challenge in software engineering. Effective translation involves not only syntactic conversion but also careful preservation of the semantics of the input code. Translated code must maintain logical correctness, handle language specific constructs appropriately, and robustly manage edge cases. Even minor inaccuracies in translation can lead to significant semantic errors, causing runtime or compilation issues.

A reliable code translation has significant practical value in modern software development. Organizations often need to migrate old or outdated codebases into newer programming languages, a process that can be costly, time consuming, and error prone when done manually (Browne, 2016). Additionally, modern software applications frequently integrate multiple programming languages and platforms, creating a need for seamless interoperability. Effective automated translation simplifies these tasks by reducing manual effort, improving consistency, and enabling developers to reuse existing software more efficiently. High quality translation tools therefore help organizations save time, lower costs, reduce errors, and accelerate overall software development and innovation.

Recent advances in large language models (LLMs) have notably enhanced automated code translation. Trained on extensive, diverse code datasets, these models demonstrate impressive zero shot translation capabilities (Yadav and Mondal, 2025). However, despite their syntactic strengths, LLMs frequently fall short in preserving semantic equivalence, especially when dealing with complex logic or subtle edge cases. Such limitations are exacerbated by model biases, hallucinations, brittle reasoning, and the absence of built-in verification processes (Maveli et al., 2025).

The recent emergence of multi-agent systems offers a promising direction to address these challenges. Instead of relying on a single model to perform the entire task, multi-agent frameworks divide the problem into smaller, specialized sub-tasks, with different agents responsible for different parts of the process (Ng, 2024). In the context of code translation, agents can focus individually on analyzing the input code, generating an initial translation, creating and running unit tests, detecting errors, and refining the output. By breaking down the task and allowing agents to specialize, the system can systematically verify and improve translations at each stage. This collaborative approach not only helps catch errors that a single model might miss but also makes the translation process more robust, reliable, and interpretable.

This paper investigates whether multi-agent systems can improve translation accuracy and reliability compared to a baseline one shot LLM translator. Specifically, it introduces and evaluates three multi-agent architectures. First, a Translator with a unit test verifier to ensure robust code validation through testing. Second, an Analyzer combined with a Translator and unit test verifier, enhancing translation robustness via initial code analysis and subsequent testing. Finally, a Deeper Analyzer coupled with a Translator and an LLM based evaluator that emphasizes semantic validation with reduced reliance on conventional unit testing. We systematically benchmark these architectures against a baseline model across Python-to-Java and Java-to-Python translation tasks, demonstrating their capability to substantially enhance translation accuracy and reliability.

Background

Early research on automating code translation focused on rule based systems and transpilers that mapped syntax across languages (Miller, 2022). While effective for simple cases, these approaches struggled to preserve deeper semantic structures and adapt to language-specific constructs, limiting their scalability (Costa-Jussà et al., 2012). The rise of large language models (LLMs) marked a major shift, offering a data driven alternative. Pretrained on diverse multilingual code corpora, models such as OpenAI's Codex Chen et al., 2021 and Meta's TransCoder Q. Sun et al., 2024 demonstrated strong zero shot and few shot translation capabilities across many programming languages. However, despite their syntactic fluency, LLM based translators frequently produce semantically incorrect outputs, particularly for programs with complex control flows or subtle logic.

To address these limitations, UniTrans Yang et al., 2024 introduced a framework for improving LLM based code translation through iterative test driven repair. UniTrans first generates input-output test cases for the source program, then translates the code using an LLM. After translation, the system executes the translated program on the test cases. If discrepancies or compilation errors occur, UniTrans uses error messages and test case failures to guide the LLM in repairing the code. This process is repeated iteratively until the translated program passes all tests or a maximum number of iterations is reached. UniTrans achieves a computational accuracy of 85.0% for Java-to-Python translation and 56.2% for Python-to-Java translation using GPT-3.5 level models.

Building on these ideas, TRANSAGENT Yuan et al., 2024 proposes a more structured, multi-agent approach to enhance translation robustness and efficiency. Instead of relying on one big repair loop, TRANSAGENT splits the translation process into four specialized agents: an Initial Code Translator, a Syntax Error Fixer, a Code Aligner, and a Semantic Error Fixer. First, the Initial Code Translator generates both the target code and test cases. If syntax errors are detected, the Syntax Error Fixer creates a fixing plan and applies corrections iteratively. Once syntax correctness is achieved, the Code Aligner maps blocks between the source and target programs using control-flow analysis. Finally, the Semantic Error Fixer finds runtime behavioral mismatches at the block level and corrects them. This modular design enables more detailed and systematic error detection and correction compared to UniTrans's whole program fixing strategy.

Through experimentation, TRANSAGENT demonstrates substantially higher translation accuracy than UniTrans. For Python-to-Java translation, TRANSAGENT improves computational accuracy from 56.2% (UniTrans) to 89.5%, and for Java-to-Python translation, from 85.0% to 93.2%. TRANSAGENT also achieves higher sample efficiency, with most programs correctly translated within two fixing iterations, and introduced a novel block level mapping mechanism that significantly improves alignment accuracy over prior approaches such as TransMap B. Wang et al., 2023.

It is worth noting that most prior systems, including UniTrans and TRANSAGENT, were evaluated using GPT-3.5 models. GPT-3.5 achieves approximately 49.3% pass@1 on HumanEval Murdza, 2024 and 19.9% calibrated pass@1 on BigCodeBench Face, 2024. In contrast, this work uses GPT-4o, OpenAI's latest general-purpose model, which demonstrates stronger coding performance: approximately 65% pass@1 on HumanEval and 61.1% on BigCodeBench (Yu et al., 2024, Zhuo, 2024). By building on a more capable LLM foundation, this study isolates the specific impact of multi-agent system design and evaluates its effectiveness in enhancing LLM-based code translation. It aims to demonstrate that, with continued advancements in LLMs and the integration of multi-agent architectures, code translation can become more accurate, reliable, and robust.

Methodology

General Architecture

The general architecture evaluated in this work consists of three to four specialized agents, each composed of LLMs and external tools assigned to specific sub tasks. These agents include:

- **Translator Agent**: Responsible for performing the initial code translation from the source to the target language.
- **Evaluator Agent**: Responsible for evaluating the correctness of the translated code, typically through automated unit tests or execution based validation.
- **Regenerator Agent**: Activated when the Evaluator Agent detects errors, this agent attempts to re-translate or repair the code. In certain architectures, the Regenerator Agent is also invoked during intermediate steps to improve partial translations.
- **Analyzer Agent**: Used in some architectures, the Analyzer Agent is used to assist other agents by conducting in depth code analysis, leveraging prompt reasoning techniques such as chain-of-thought prompting to enhance decision making and improve the task's accuracy, and by extension the general translation accuracy of the model.

A schematic overview of the general multi-agent architecture is shown in Figure 1. In the standard flow, the Translator Agent produces an initial translation, which is then passed to the Evaluator Agent for validation. If any issues are detected, the code is sent to the Regenerator Agent for correction. This validation-regeneration loop continues until the Evaluator Agent confirms that the code passes all tests or no further improvements are possible. In some architectures, the Analyzer Agent is used at various stages to provide additional reasoning support to the other agents. All architectures are implemented using the LangGraph framework for flexible agent orchestration and communication (LangChain-Team, 2024). The main LLM used for the agents is GPT-4o, chosen for its performance on software engineering tasks and its popularity (Murdza, 2024, Face, 2024) . The prompts used for the agents used various proven techniques to maximize prompt performance, and they were formatted in a way best suited for Open AI's model based on OpenAI, 2023 and Sahoo et al., 2025.

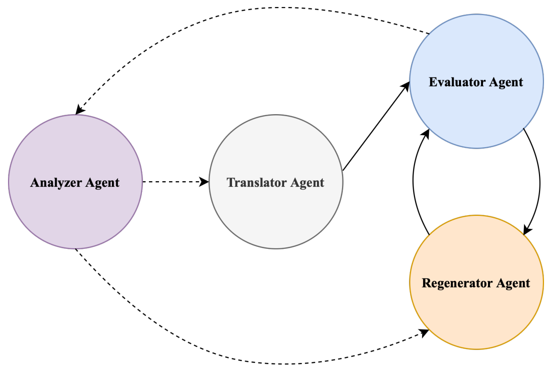


Figure 1: General architecture of the multi-agent systems used for code translation.

Baseline Architecture

The baseline architecture consists of a single pass through the Translator Agent without any additional validation, analysis, or regeneration. The Translator Agent is implemented using GPT-4o, OpenAI's latest general-purpose model, which is among the top performing general purpose models for code generation tasks (at the time when this research project began).

The Translator Agent receives three inputs: the source language of the input code, the target language for translation, and the code snippet to be translated. Within the prompt, a role assignment strategy is employed, which has been shown to improve LLM agent performance on complex tasks(Yehudai et al., 2025). Additional prompt engineering techniques are applied, including explicit instructions specifying output expectations and a list of translation specific notes tailored to common differences between Python and Java. Finally, an output formatting guideline is provided to make the evaluation and automated testing easier.

The output of the Translator Agent is a one shot translation of the input code into the desired target language. No post processing, error checking, or iterative correction is performed in the baseline. The full translation prompt and Configuration details are available in the accompanying GitHub repository, with access information provided in Appendix A.

Multi-Agent Architecture #1

The first architecture in this work uses three agents: a Translator Agent, an Evaluator Agent, and a Regenerator Agent. At a high level, the Translator Agent is identical to the one used in the baseline model, performing a direct one shot translation of the input code into the target language. The Evaluator Agent validates the translated code by generating 20 test inputs based on the source code, and then running the input code on these generated inputs, and saving the resulting input-output pairs into a JSON file. The translated function is then evaluated against these expected outputs. If evaluation fails either due to runtime errors or incorrect outputs, the Regenerator Agent intervenes to repair the translated code or the supporting script. The Regenerator Agent also assists whenever issues arise in the script generation or execution phases in the sub-tasks of the Evaluator Agent. Through this evaluation and regeneration process the problems with the translation can be detected and used for generating a more robust translation.

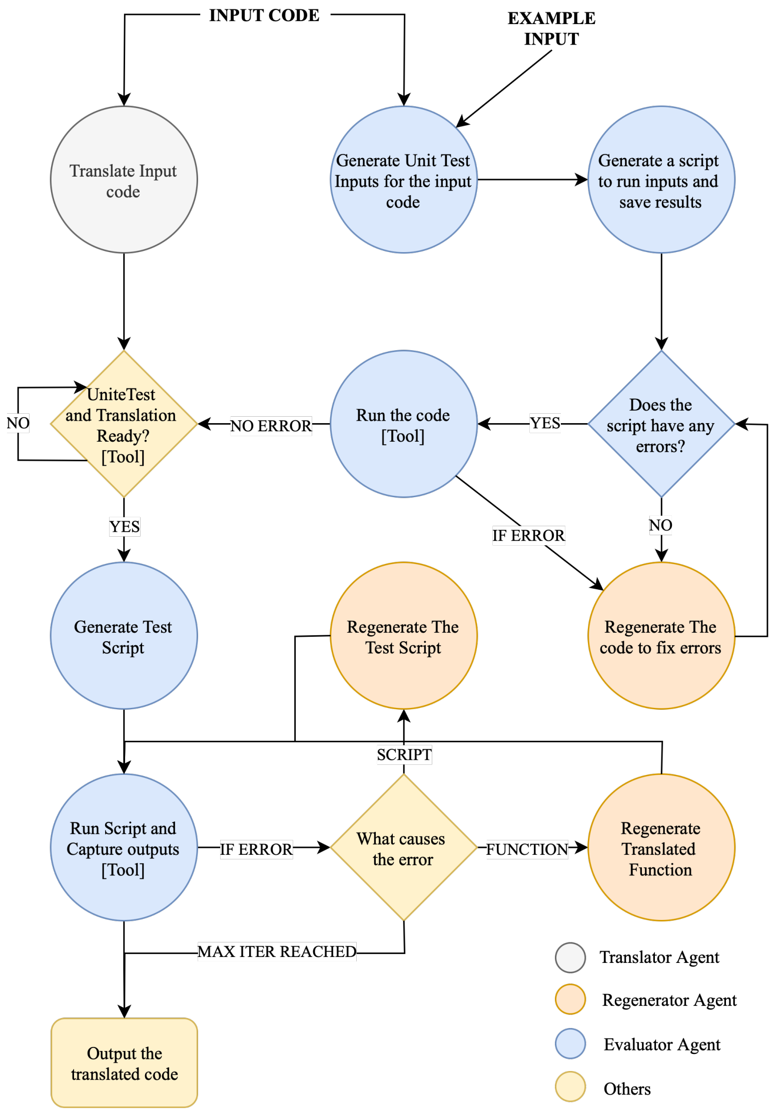

INPUT CODE
EXAMPLE INPUT
Translate Input code
Generate Unit Test Inputs for the input code
Generate a script to run inputs and save results
UniteTest and Translation Ready? [Tool]
NO
NO ERROR
Run the code [Tool]
YES
Does the script have any errors?
NO
IF ERROR
YES
Generate Test Script
Regenerate The Test Script
Regenerate The code to fix errors
SCRIPT
Run Script and Capture outputs [Tool]
IF ERROR
What causes the error
FUNCTION
Regenerate Translated Function
MAX ITER REACHED
Output the translated code
Translator Agent
Regenerator Agent
Evaluator Agent
Others

Figure 2: Multi-Agent Architecture #1: Translation, Evaluation, and Regeneration Workflow.

The complete multi-agent architecture is illustrated in Figure 2. The system is designed to initialize two parallel workflows: the Translator Agent begins translating the input code, while the Evaluator Agent simultaneously starts preparing the test data infrastructure. Once the Evaluator agent has completed its initial tasks and the Translation Agent has completed producing an initial translation, the evaluation and regeneration loop proceeds.

Upon receiving the input code and a set of sample inputs to the input code, the Evaluator Agent first generates 20 diverse test inputs. This step is performed by an LLM using a chain-of-thought prompting technique, which encourages step by step reasoning to produce more robust and varied inputs (Wei et al., 2023). These inputs are designed to better explore the behavior of the input function across a wide range of cases, increasing the likelihood of exposing translation errors.

Next, the Evaluator Agent invokes another LLM to generate a script that executes the input function on the generated inputs and saves the results into a JSON file. To assist with the script generation and improve robustness, the LLM is provided with a starter script as an initial template. The use of an LLM is necessary because the number, types, and structures of the input parameters can vary significantly across different functions. Manually writing a fully general execution script would require modifying the input code, which risks altering it semantically. Furthermore, serializing diverse data types, particularly in statically typed languages like Java, introduces additional complexity. By using an LLM, the starter script is dynamically adapted to the input specifications without modifying the input code.

As a safeguard, the generated script is automatically checked for syntactic correctness and logical consistency by an LLM. If errors are detected during script generation, the script and the corresponding error information are passed to the Regenerator Agent, which attempts to repair the faulty script. Once the script passes the LLM check, the agent proceeds to execute it. If the script produces an error during execution, or fails to generate a JSON file as expected, it is again passed along with the associated error to the Regenerator Agent for further repair. Once a valid execution script is obtained, the agent runs the script which passes the 20 generated inputs through the original input code, and the resulting input-output mappings are saved into a JSON file.

After the JSON dataset is successfully created, the Evaluator Agent generates a second script to test the translated function. This evaluation script randomly samples 5 out of the 20 generated test cases for each evaluation round. Using only 5 samples at a time allows the system to select different subsets of test cases after each regeneration attempt, ensuring that the regenerated function does not overfit to a static set of examples. This strategy promotes the creation of more generalized and robust translations.

Similar to the earlier script generation step, an LLM generates the test script based on a starter code, modifying it to work with the expected types and structures of the generated inputs and the translated function. The translated function is then evaluated by running the 5 randomly selected inputs through it, and the outputs are compared against the expected outputs from the original input code for each respective input used this test script. If execution errors or test failures occur, the system diagnoses the source of the error. An LLM determines whether the problem lies in the translated function or in the supporting testing script. Based on this diagnosis, the appropriate component is sent to the Regenerator Agent for targeted repair. This evaluation-regeneration loop continues iteratively. After each regeneration attempt, a new random subset of 5 test cases is selected, and the process repeats. The loop terminates when the translated function passes all tests or when a maximum of three regeneration iterations is reached. The maximum iteration limit ensures bounded computational overhead.

Finally, if successful, the output of the system is the corrected translated code. If, after three regenerations, the function still fails to pass the Evaluator Agent, the output would be the latest translated code alongside detailed error information to aid manual debugging.

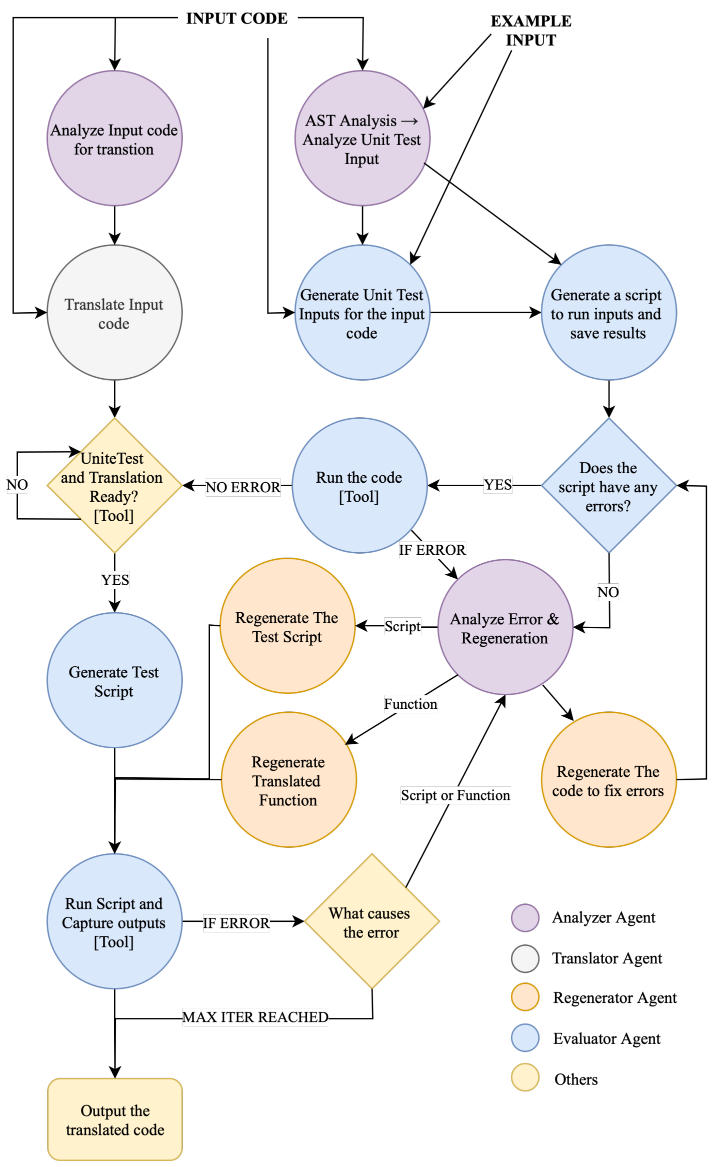
The full codebase for this architecture, including prompts, configurations, and agent implementations, is available in the accompanying GitHub repository (Appendix A).

Multi-Agent Architecture #2

The second multi-agent architecture builds on the Multi-Agent Architecture #1 by introducing an additional agent: the Analyzer Agent, alongside the existing Translator, Evaluator, and Regenerator Agents (Figure 3). The goal of the Analyzer Agent is to allow each sub-task to reason through its objective and strategize before execution, following the chain-of-thought prompting methodology Wei et al., 2023 to improve task performance. The Analyzer Agent has 3 tasks to help improve the performance of the architecture.

Task 1: Code Analysis Prior to Translation

The Analyzer Agent's primary role is to examine the input code prior to translation. It generates a detailed reasoning chain that includes a high-level summary of the function, a step-by-step walkthrough of the code's logic and control flow, identification of key variables, and an analysis of its memory and time complexity. Additionally, it produces a list of translation-specific concerns to consider when converting to the target language. Finally, it outputs a pseudocode version of the input function, tailored for the target language using the chain-of-code prompting technique introduced by "–li2024chaincodereasoninglanguage}. This comprehensive analysis is passed to the Translator Agent to enable a more informed and accurate translation.



INPUT CODE
EXAMPLE INPUT
Analyze Input code for transtion
AST Analysis → Analyze Unit Test Input
Translate Input code
Generate Unit Test Inputs for the input code
Generate a script to run inputs and save results
UniteTest and Translation Ready? [Tool]
NO
NO ERROR
Run the code [Tool]
YES
Does the script have any errors?
IF ERROR
YES
NO
Regenerate The Test Script
Script
Analyze Error & Regeneration
Generate Test Script
Function
Regenerate Translated Function
Script or Function
Regenerate The code to fix errors
Run Script and Capture outputs [Tool]
IF ERROR
What causes the error
MAX ITER REACHED
Output the translated code
Analyzer Agent
Translator Agent
Regenerator Agent
Evaluator Agent
Others

Figure 3: Multi-Agent Architecture #2: Multi-Agent System with Analyzer Agent for Pre-Analysis and Error Diagnosis. (Any component not labeled Tool is an LLM)

Task 2: Input and Script Analysis for Unit Test Generation

The second responsibility of the Analyzer Agent is to enhance the input generation process used for testing the translated code and to assist in generating the script that constructs the input-output JSON file. To do this, the Analyzer performs an Abstract Syntax Tree (AST) analysis of the input code's structure in a JSON format W. Sun et al., 2023. Using the AST results alongside the original input code and sample inputs, it generates two reports: one for input generation and another for script generation.

The unit test input generation report includes:

- Descriptions of expected input types, shapes, and constraints.
- Identification of normal and edge case behaviors.
- Values to avoid in order to prevent runtime errors.
- A step-by-step strategy for generating safe, diverse, and meaningful inputs.

The script generation report includes:

- Instructions for executing the input function with generated inputs.

- Guidelines for capturing, structuring, and organizing outputs.
- Strategies for handling exceptions, serialization issues, and ensuring file safety.
- A list of potential pitfalls to watch out for.

This analysis helps the Evaluator Agent receives high-quality inputs and scripts, reducing the likelihood of errors caused by differences between programming languages. The input generation report is sent to the input generator LLM, and the script generation report is sent to the script generator LLM to create the input-output JSON file. The rest of the workflow, aside from the regeneration process, remains the same as in Multi-Agent Architecture #1.

Task 3: Error Analysis for Regeneration

The final responsibility of the Analyzer Agent is error diagnosis and recovery support. When a script generation task encounters an error. Whether during LLM validation checks for the input-output pair JSON file generation script, any errors encountered by the execution of this script, or any errors during the test script execution to test the translated code. The Analyzer Agent takes the faulty script and error message and generates an error analysis report. This report includes:

- A diagnosis of the root cause of the error.
- A step-by-step reasoning of how and why the error occurs.
- One to three proposed solutions to address the error.

The Regenerator Agent uses this report to perform a more informed and targeted regeneration of the faulty component.

Summary

Overall, apart from the addition of the Analyzer Agent, the rest of the architecture remains identical to Multi-Agent Architecture #1. The addition of this agent to the architecture can be seen in Figure 3. The Analyzer Agent supports the Translator, Evaluator, and Regenerator Agents by enhancing reasoning, planning, and recovery processes, trying to improve the overall robustness and translation quality. The full implementation details of this architecture are available in the accompanying GitHub repository (Appendix A).

Multi-Agent Architecture #3

The third and final architecture introduces a streamlined yet more analytically intensive workflow. It includes four agents: a Deep Analyzer Agent, a Translator Agent, an Evaluator Agent, and a Regenerator Agent. Unlike the previous two designs, this architecture focuses on performing a comprehensive pre-translation analysis and a simpler, critique based evaluation process using a separate LLM model. Like the previous architectures if a problem with the translated code is detected by the Evaluator Agent, the Regenerator Agent attempts to fix the identified errors and send it back to the Evaluator Agent for re-evaluation. The process continues until the Evaluator Agent finds no issues with the translated code. The overall flow is illustrated in Figure 4.

Deep Analyzer Agent

The process begins with the Deep Analyzer Agent, which first runs an AST analysis on the input code. It then combines the AST output with the input code and sample inputs to produce a structured report. This report includes four sections: a plain English summary of the input code's purpose, a detailed step-by-step walkthrough of its logic and control flow, a translation watchlist highlighting potential cross language pitfalls, and a pseudocode version of the input code written in the target language's idiom (C. Li et al., 2024).

While the basic Analyzer Agent also produces a semi-structured output, it relies on implicit chain-of-thought reasoning, where the model is expected to reason internally and provide a coherent answer without being explicitly guided through the process. In contrast, the Deep Analyzer Agent uses an explicit and structured chain of thought prompting (SCoT) strategy J. Li et al., 2023. It guides the model to articulate its reasoning step by step, systematically address edge cases and any concerns regarding the target language, and perform a self-consistency check X. Wang et al., 2023 to ensure alignment between the AST structure, input code, and the sample inputs.

Incorporating AST analysis and sample inputs alongside the raw input code gives the model a more comprehensive view of the function, enabling deeper understanding of its structure and behavior. This, in turn, can help improve the quality and reliability of the analysis passed on to the Translator Agent.

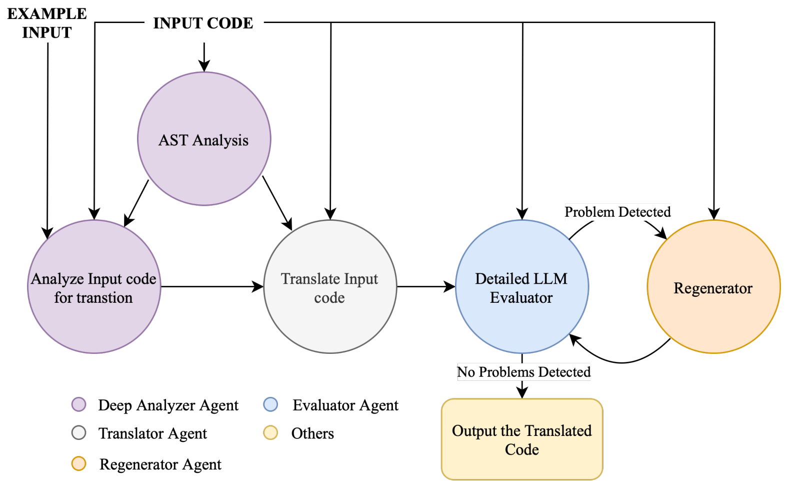


Figure 4: Multi-Agent Architecture #3: Deep Analysis and Criticism-Based Evaluation Loop

Table 1 breaks down the main differences between the Deep and basic Analyzer agents.

Features / Techniques	Deep Analyzer	Basic Analyzer
Structured Output	Yes (multi-layered with constraints)	Yes (simpler and lighter)
Chain of Thought Reasoning (CoT)	Explicit and guided	Implicit and unguided
Structured Reasoning (SCoT)	Enforced via section breakdown	Not fully enforced
Self-Consistency Check	Included	Not included
Inputs	AST + Input code + sample inputs	Input code

Table 1: Deep Analyzer and Basic Analyzer based on key prompt engineering attributes.

By explicitly guiding both the reasoning process and the output structure using techniques such as structured chain of thought and self-consistency checks, the Deep Analyzer tries to help, generates more accurate, idiomatic, and robust intermediate representations. These outputs, along with the original input code, sample input, and the AST analysis, are then passed to the Translator Agent for downstream translation.

Translator Agent

The Translator Agent in this architecture is similar to the ones used previously but benefits from the enhanced context provided by the Deep Analyzer and the AST analysis. It produces a first-pass initial translation which is then sent to the Evaluator Agent for evaluation and validation.

Evaluator Agent

The Evaluator Agent in this architecture is a critique LLM. In order to reduce bias and improve objectivity, the Evaluator Agent in this system is implemented using Claude 3.7 Sonnet by Anthropic. This model is one of the best general purpose LLMs for coding achieving a 62.3% of SWE bench mark and 82.4% pass@1 on the HumanEval benchmark (Anthropic, 2024). Similar to the Deep Analyzer Agent, the Evaluator Agent uses a multi-phase prompt engineered for a rigorous and reliable assessment.

It first conducts a chain-of-thought analysis, reasoning through the logic of both the input and translated functions to build a structural and semantic understanding. If multiple interpretations are possible, it performs a self-consistency check to identify the most plausible one (X. Wang et al., 2023). It then executes a structured chain of verification procedure that compares the translated code with the original based on input types, control flow, variable usage, output behavior, and exception handling (J. Li et al., 2023).

The prompt also incorporates contrastive thinking: the model is asked to describe what a bad translation might look like and whether the current one avoids those errors (Chia et al., 2023). The Evaluator Agent produces a concise report including this diagnostic reasoning, a checklist result, and a final verdict on whether the translated code needs to be regenerated or not.

Regenerator Agent

If issues are detected, the translated code, the input code, and the evaluator report are passed to the Regenerator Agent. The Regenerator attempts to correct the translation based on the inputs, and sends the new version of the translated code back to the Evaluator. This loop continues until the Evaluator Agent confirms that the translation passes all checks, or until the maximum number of iterations (set to five) is reached. At that point, the latest version of the translated code is returned as the final output.

Summary

Architecture 3 emphasizes deeper reasoning, explicit reflection, and LLM agent feedback loops. By combining AST based analysis, structured self-consistency checking, and independent LLM evaluation, this architecture aims to produce higher quality translations, compared to the baseline, while maintaining a simpler flow, compared to Multi-Agents #1 and #2. It trades execution based evaluation for LLM model driven critique, allowing for a more lightweight evaluation loop and not dependent on the generation of correct and quality unit test.

Similar to the other Multi-Agent Architectures, the full implementation of this architecture can be found in the GitHub repository shared in Appendix A.

Evaluation

Evaluation Method

To evaluate the proposed multi-agent architectures, I used the GeeksforGeeks parallel Python–Java dataset originally employed in the TransCoder evaluation Group, 2021. This dataset contains over 600 parallel functions written in both Python and Java. However, due to computational constraints, only a subset of 50 functions were selected for evaluation. The selection was based on factors such as function complexity, diversity of programming techniques (e.g., recursion), and a range of functional objectives. The complete list of selected test files is provided in Appendix B.

Each sample in the dataset consists of a function followed by a corresponding set of unit tests. An example is shown in Appendix C. As illustrated in Figure 5, both the function and its unit tests are extracted and passed to the multi-agent system under evaluation. The function serves as the input code, while the unit tests act as sample inputs. In addition to generating a translated function, the system records metadata such as intermediate outputs and the number of regenerations.

The translated function is then appended to the parallel test file to generate an evaluation script. This script compares the outputs of the original and translated functions using the dataset's unit tests. An example of such a script is shown in Appendix D. Each translated function is evaluated on 10 unit tests, and the number of passed cases is recorded.

Finally, the evaluation results and metadata is stored in a CSV file. This file generally includes the test file name, input and output languages, number of passed tests, a boolean indicating whether all tests passed, number of regeneration attempts, whether the iteration cap was reached, the final translated code, the generated unit test evaluation output, and the complete execution history of the multi-agent translation and unit test generation process (all of the sub-task outputs). This may vary based on the multi-agent (e.g. the baseline does not record the number of regenerations as it doesn't have that feature)

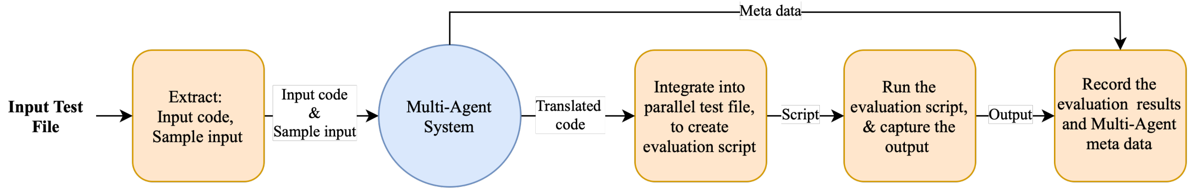

Meta data
Input Test File
Extract: Input code, Sample input
Input code & Sample input
Multi-Agent System
Translated code
Integrate into parallel test file, to create evaluation script
Script
Run the evaluation script, & capture the output
Output
Record the evaluation results and Multi-Agent meta data

Figure 5: Multi-Agent Evaluation Work Flow

Code Translation Evaluation

The described evaluation was performed on the baseline, the three proposed multi-agent architectures and Multi-Agent #2 with the Deeper Analyzer (DA) form architecture #3. This architecture is exactly the same as the Multi-Agent #2 , except that the input code analysis is done by the deep analyzer agent. The result of the computational accuracy analysis for both Java-to-Python and Python-to-Java is displayed in Table 2 along with the number of passed and failed test. The full csv output of all of the evaluations can be found in the GitHub link shared in Appendix A.

The baseline one-shot architecture performs reasonably well on Java-to-Python translation tasks, but exhibits significant performance limitations in the Python-to-Java direction, achieving only a 46% accuracy rate. As detailed in Table 3, 42% of the failures in Python-to-Java are attributed to script-level issues, primarily compilation errors caused by missing semicolons, incorrect class declarations, or improper type handling. In contrast, only 12% of failures in this direction are due to logical or semantic mismatches in output. These results indicate that the baseline architecture lacks robustness when translating into statically typed languages like Java, where minor syntactic errors are sufficient to prevent successful execution. Conversely, Java-to-Python translations benefit from Python's dynamic typing and more permissive syntax, allowing many of the baseline translations to execute correctly even in the presence of minor structural inconsistencies. Finally, the one Java-to-Python translation that resulted in a script-level issue was analyzed and determined to have originated from a syntactic error in the reference function within the dataset. As a result, this error persists across all architectures.

	Baseline	**Multi-Agent #1**	**Multi-Agent #2**	**Multi-Agent #3**	**Multi-Agent #2 + DA**
Python to Java	**46%** Passed: 23 Failed: 27	**68%** Passed: 34 Failed: 16	**68%** Passed: 34 Failed: 16	**78%** Passed: 39 Failed: 11	**68%** Passed: 34 Failed: 16
Java to Python	**84%** Passed: 42 Failed: 8	**82%** Passed: 41 Failed: 9	**80%** Passed: 40 Failed: 10	**84%** Passed: 42 Failed: 8	**86%** Passed: 43 Failed: 7

Table 2: Comparison of success rates across different agent architectures for code translation tasks.

With Multi-Agent #1, the introduction of a unit test evaluation and regeneration loop led to a substantial improvement in Python-to-Java translation, increasing overall accuracy by 21% compared to the baseline. In contrast, Java-to-Python performance slightly declined by 2%. As shown in Table 3, this improvement in Python-to-Java was primarily driven by a significant reduction in compilation errors, which dropped from 42% to 14%. These corrected errors typically involved missing semicolons, incorrect public class declarations, absent imports, and return type mismatches. The regeneration loop successfully identified these issues during compilation or through failing generated unit tests and guided the translator to produce corrected code. Additionally, some simple logical errors such as off by one mistakes in loop conditions, were resolved when exposed by simple generated unit test cases .

However, deeper semantic issues remained unresolved as demonstrated by output mismatch as shown in Table 3. In Python-to-Java, logical errors (output mismatch) persisted when the unit tests failed to cover edge cases, such as handling of empty inputs, boundary values, or specific numerical constraints. As a result, some incorrect implementations passed the test suite and were marked as successful, despite failing final evaluation. In Java-to-Python, where python's compilation is less strict, the unit test loop had limited utility. Errors related to incorrect algorithmic behavior, misaligned return values, and placeholder logic often persisted, as the generated unit tests could not catch these errors. While Multi-Agent #1 proved highly effective at resolving surface level compilation issues, particularly for Java's stricter language constraints, but it was insufficient for identifying and correcting deeper semantic and logical issues, especially in Python.

	Baseline	**Multi-Agent #1**	**Multi-Agent #2**	**Multi-Agent #3**	**Multi-Agent #2 + DA**
Compilation Error (Java to Python)	**2%** (Error: 1)	**2%** (Error: 1)	**2%** (Error: 1)	**2%** (Error: 1)	**2%** (Error: 1)
Compilation Error (Python to Java)	**42%** (Error: 21)	**14%** (Error: 7)	**24%** (Error: 12)	**10%** (Error: 5)	**16%** (Error: 8)
Output Mismatch (Java to Python)	**14%** (Error: 7)	**16%** (Error: 8)	**18%** (Error: 9)	**14%** (Error: 7)	**12%** (Error: 6)
Output Mismatch (Python to Java)	**12%** (Error: 6)	**18%** (Error: 9)	**8%** (Error: 4)	**12%** (Error: 6)	**16%** (Error: 8)

Table 3: Script / Compilation errors and unit test mismatches / Logical/ Semantic errors for each system.

Multi-Agent #2 introduces an Analyzer agent aimed at enhancing the reasoning capabilities of three key components from Multi-Agent #1 architecture: the Translator, the Regenerator, and the unit test generation stage of the Evaluator agents. The Analyzer's purpose is to help the LLMs reason better by providing high level context and guidance. However, as shown in Table 2, this added reasoning did not lead to consistent performance improvements. Python-to-Java accuracy remained effectively unchanged compared to Multi-Agent #1, while Java-to-Python accuracy dropped to 80%, down from 82% in Multi-Agent #1 and 84% in the baseline. The average number of regeneration steps also increased (Table 4), indicating reduced efficiency in resolving translation errors.

In the Python-to-Java direction, the percentage of compilation failures increased from 14% to 24%. Based on analysis of the multi-agent execution history, this increase is attributed to the use of implicit, semi-structured chain-of-thought reasoning in the Analyzer agent. Markdown headings, pseudocode, and descriptive comments of the Analyzer Agent's outputs frequently leaked into the Translator's output, resulting in syntactically invalid Java code. These failures often persisted across regeneration attempts, as the underlying analysis remained unchanged. Although the percentage of logical errors decreased from 18% to 8%, this shift does not indicate improved semantic correctness. In most cases, files that previously failed due to logic errors in Multi-Agent #1 now failed earlier due to compilation issues introduced by analysis leakage.

In the Java-to-Python direction, semantic failures increased from 16% to 18%. Again, the semi-structured reasoning provided by the Analyzer failed to enhance unit test quality and, in some cases, introduced ambiguity that negatively affected translation reliability. Review of the execution history revealed frequent cases where the Translator Agent produced minimal or placeholder logic, such as hardcoded return values, to pass generated unit tests. These examples often passed the Evaluator Agent but failed to yield robust or semantically correct translations, contributing to the observed decline in overall translation accuracy.

	Multi-Agent #1	**Multi-Agent #2**	**Multi-Agent #3**	**Multi-Agent #2 + DA**
Avg. Iteration (Java to Python)	0.47	0.53	0.13	0.24
Avg. Iteration (Python to Java)	0.56	0.88	0.10	0.82

Table 4: Average number of regenerations/iterations for code translation across multi-agent architectures.

Multi-Agent #3 combines a more robust Analyzer Agent, featuring structured and explicit reasoning, with a simplified architecture that replaces the unit test loop with an LLM-based critique mechanism. This configuration demonstrated the best overall performance in Python-to-Java translation, achieving 78% accuracy as shown in Table 2, marking a 32% improvement over the baseline. In the Java-to-Python direction, Multi-Agent #3 outperformed both Multi-Agent #1 and #2, matching the baseline's accuracy of 84%. The error breakdown in Table 3 shows a significant reduction in compilation errors for Python-to-Java translations, while maintaining a low rate of semantic failures. For Java-to-Python, the distribution of error types closely aligns with the baseline, with no increase in script or logic errors.

As shown in Table 4, Multi-Agent #3 also achieved the lowest average number of regeneration steps across all models. The vast majority of test files passed the Evaluator Agent on the first attempt, without the need for regeneration. In the worst case, a file required four regenerations, and a few required two to one. A detailed review of the execution history revealed that the structured output from the Analyzer Agent effectively prevented leakage of evaluation content, such as comments, and reasoning steps, into the translated code and observed in Multi-Agent #2. This demonstrates that the structured Deep Analyzer Agent, combined with direct integration of AST analysis into the Translator, enabled a significantly more robust and reliable translation process compared to the previous architectures.

Building on the strong performance of the Deep Analyzer agent, the input code analysis component of the Analyzer Agent in Multi-Agent #2 was replaced with the Deep Analyzer (DA) to evaluate its effectiveness. It is important to note that, unlike in Multi-Agent #3, the AST analysis was not provided directly to the Translator Agent. As shown in Table 2, this modified architecture (Multi-Agent #2 + DA) achieved the highest Java-to-Python accuracy across all configurations, reaching 86%, which represents a 2% improvement over the baseline. In contrast, Python-to-Java translation accuracy remained unchanged at 68%, consistent with the performance of both Multi-Agent #1 and #2. By analyzing the specific error types in Table 3, this architecture demonstrated a reduction in semantic errors in Java-to-Python translation, a 4% improvement over Multi-Agent #2 and a 2% gain compared to the baseline. Showing a slight improvement in understanding the semantics of the input code. Compilation errors for Python-to-Java also decreased by 8% compared to Multi-Agent #2, reflecting improved robustness to syntax-related issues. Additionally, as shown in Table 4, the average number of regeneration steps decreased in both directions: from 0.53 to 0.24 for Java-to-Python, and from 0.88 to 0.82 for Python-to-Java.

Unit Test Generation Evaluation

To evaluate the effectiveness of unit test generation across architectures, a confusion matrix analysis was conducted (Table 5). This analysis compares the outcomes of the evaluator generated unit tests with the final results of the multi-agent evaluation. A true positive (TP) indicates that a translation passed both the generated unit tests and the final evaluation. A false positive (FP) means the translation passed the generated tests but failed final evaluation, suggesting test cases were too shallow or incomplete. A false negative (FN) occurs when the translation failed the generated test but ultimately succeeded the evaluation, which may indicate overly strict or misaligned test logic. A true negative (TN) means both the unit test and evaluation correctly rejected the translation.

Results show that Multi-Agent #1, which uses a basic unit test loop, achieves a 72% TP rate in Java-to-Python translation, but with a 16% FP rate, indicating limited test coverage of edge cases. Multi-Agent #2, which introduces the Analyzer Agent for both translation and test generation, performs slightly worse with a 68% TP rate and a higher 12% FN rate, suggesting that the added reasoning introduced ambiguity without improving test quality. In contrast, Multi-Agent #2 + Deep Analysis (DA) retains the same unit test generator as Multi-Agent #2, but improves translation performance through more structured input code analysis. This results in a 78% TP rate, despite unchanged test quality, along with a reduction in both FP and FN rates. The improvement is therefore attributed to better initial translations that are more likely to pass final evaluation when correct, rather than enhanced unit test discrimination.

In Python-to-Java translation, the TP rate improves from 54% in Multi-Agent #2 to 58% in the DA variant, again with identical analyzer for the input generator. FN remains constant at 10% across all models, while FP decreases slightly, reinforcing the idea that stronger translation, not test generation, is responsible for performance gains. TN rates remain low in all configurations and directions, reflecting the difficulty of generating test inputs that reliably trigger Evaluator Agent failures.

These findings highlight two key takeaways:

- Shallow or generic test cases often allow incorrect translations to pass, resulting in a high false positive (FP) rate.
- Even when the unit test generation process is unchanged, improved analysis during translation, such as using the structured Deep Analyzer, can lead to better alignment with evaluator outcomes.

	Multi-Agent #1	**Multi-Agent #2**	**Multi-Agent #2 + DA**
True Positive	**72%**	**68%**	**78%**
(Java to Python)	Count: 36	Count: 34	Count: 39
True Negative	**2%**	**4%**	**0%**
(Java to Python)	Count: 1	Count: 2	Count: 0
False Positive	**16%**	**16%**	**14%**
(Java to Python)	Count: 8	Count: 8	Count: 7
False Negative	**6%**	**12%**	**8%**
(Java to Python)	Count: 3	Count: 6	Count: 4
True Positive	**62%**	**54%**	**58%**
(Python to Java)	Count: 31	Count: 27	Count: 29
True Negative	**12%**	**16%**	**14%**
(Python to Java)	Count: 6	Count: 8	Count: 7
False Positive	**20%**	**20%**	**18%**
(Python to Java)	Count: 10	Count: 10	Count: 9
False Negative	**10%**	**10%**	**10%**
(Python to Java)	Count: 5	Count: 5	Count: 5

Table 5: Confusion matrix analysis of unit test generation results across agent architectures.

Discussion

Analysis

This study explored the design and performance of various multi-agent architectures for code translation, specifically targeting the Python-to-Java and Java-to-Python directions. The experiments reveal two distinct classes of failure modes: (1) syntactic issues, particularly prevalent in Python-to-Java translations, due to Java's strict compilation requirements, and (2) semantic issues, where the translated function fails to preserve the original logic, as reflected in output mismatches during evaluation.

Multi-Agent #1 demonstrated that a validation and regeneration loop built around unit test execution can significantly reduce syntactic errors in Python-to-Java translation. The architecture uses generated unit tests to validate translated code and employs an LLM-based Regenerator Agent to correct identified issues. This proved highly effective at fixing surface level syntax problems such as missing semicolons, invalid class declarations, and return type mismatches. However, deeper logical flaws remained largely unresolved. The Evaluator Agent often failed to produce diverse or rigorous unit tests, limiting its ability to expose semantic errors or edge cases as it was demonstrated in the Unit Test Generation Evaluation section. As a result, many incorrect implementations still passed the Evaluator Agent and were marked as successful.

In contrast, Java-to-Python translation saw less benefit from this approach. Because Python is dynamically typed and does not enforce strict compilation, syntactic issues were rare. The architecture's dependence on shallow unit tests sometimes led to overfitting or failure to detect logical bugs. In fact, the performance of Multi-Agent #1 and #2 in this direction slightly declined compared to the baseline. This suggests that when the initial translation is weak and unit test quality is low, iterative feedback can reinforce incorrect logic rather than correct it.

The quality of the Translator Agent's initial translation emerged as a key determinant of success. This is evident in the two best performing models: Multi-Agent #3 for Python-to-Java and Multi-Agent #2 + DA for Java-to-Python. Both achieved high accuracy with minimal regeneration steps. In Multi-Agent #2 + DA, the Deep Analyzer produced a more accurate initial reasoning chain, leading to better translation outcomes. Similarly, in Multi-Agent #3, the direct integration of AST analysis into the Translator prompt enabled highly accurate first-pass initial translation, reducing the need for regeneration.

These results suggest a clear distinction between what is required to resolve different error types. Compilation failures can be addressed effectively through regeneration loops and well-constructed unit tests. However, through deeper reasoning, structural awareness, and overall better understanding of the input code, with tools such as AST analysis, both the compilation and semantic errors can be avoided or resolved. Architectures using structured, explicit reasoning and AST based input such as in Multi-Agent #3, outperformed those relying on implicit reasoning, as seen in Multi-Agent #2. In the latter, markdown artifacts and unstructured comments often leaked into the translation, introducing new syntactic failures.

Although Multi-Agent #2 + DA used the same Deep Analyzer as Multi-Agent #3, it did not provide the Translator Agent with direct access to the AST analysis outputs. This likely contributed to its weaker performance in Python-to-Java translation compared to Multi-Agent #3. Furthermore, Multi-Agent #3 used Claude 3.7 Sonnet as the Evaluator Agent, one of the strongest LLMs for code reasoning, which provided high-quality critique, helping to identify subtle issues that a unit test evaluator might miss. Despite this, the architecture required very few regeneration steps, reinforcing the value of AST analysis in generating a strong initial translations.

It is also worth noting that Java-to-Python translations began with a strong baseline accuracy of 84%. This ceiling effect limited the observable improvement space for multi-agent systems in that direction. In contrast, Python-to-Java translations had much more room for gain, making them more sensitive to multi-agent enhancements.

Finally, these experiments reinforce the limitations of unstructured, implicit chain-of-thought prompting in complex, multi-agent workflows. While it may enhance interpretability, it often introduces ambiguity and prompt contamination, which degrade the performance of downstream agents. Structured, concise, and task-specific reasoning, particularly when enriched with AST analysis, proves more effective for guiding translation.

The analysis provided valuable engineering insights into the design of multi-agent architectures for code translation systems:

- Evaluation loops are effective at correcting syntactic issues, particularly in statically typed target languages like Java.
- Structured analysis, when passed directly to the Translator Agent (e.g., via AST), improves translation quality and reduces reliance on regeneration.
- Implicit reasoning increases prompt contamination and introduces noise that degrades translation fidelity.

- Structured and explicit chain-of-thought prompting, combined with scoped outputs and self-consistency checks, improves both reasoning quality and translation accuracy.
- LLMs in this experiment were not consistently able to generate robust unit tests, which in some cases led to false confidence and reduced overall translation reliability.
- Providing AST analysis directly to the Translator Agent, in a structured format such as JSON, was shown to be highly effective for achieving high first-pass translation accuracy.

Limitations and Future Work

While this study provides meaningful insights into multi-agent architectures for code translation, several limitations affect the scope, reliability, and generalizability of the findings.

First, the evaluation was limited to 50 parallel Python and Java functions selected from a larger dataset. Although care was taken to ensure diversity in logic and complexity, broader evaluation across more samples and application domains is needed to validate the generality of these results.

Second, budget and runtime constraints imposed practical limitations on system design and experimentation. In particular, regeneration loops were capped at a fixed number of iterations to comply with compute and API limits. This may have prevented full convergence in some cases, especially for more complex tasks, potentially underestimating the effectiveness of iterative refinement strategies.

Third, all experiments were focused on Python-to-Java and Java-to-Python translation. While these language pairs span dynamic and statically typed paradigms, further testing on additional languages, such as C++, TypeScript, or Go, would better evaluate the generality of multi-agent insights across differing type systems and idiomatic constructs.

Additionally, the translation and evaluation loop relied entirely on automated validation. Incorporating human in the loop feedback in future work could improve alignment with developer expectations and uncover qualitative issues not captured by automated metrics.

Future work may also explore the following directions:

- Evaluate individual aspects of the proposed architecture and in various combinations to better understand the contribution of each component.
- Pass AST representations directly to the Translator Agent in Multi-Agent #2 + DA to better assess the contribution of the AST analysis.
- Improve unit test generation to expose edge cases, increase test robustness and evaluate how well designed unit test affect code translation in the proposed architectures.
- Decompose the Analyzer Agent into smaller, specialized reasoning agents or apply reinforcement learning techniques (e.g., RLHF) to improve reasoning.
- Extend the framework to multi function, multi file or project level translation tasks where coordination across modules is required.

Ultimately, this work lays a foundation for building modular, interpretable, and adaptive multi-agent systems for code translation. However, further research is required to evaluate their scalability, robustness, and applicability across diverse programming ecosystems and real world development workflows.

Conclusion

This work explored the design and evaluation of multi-agent architectures for the task of code translation between Python and Java. Five distinct configurations were implemented and analyzed, ranging from a baseline one-shot model to systems leveraging structured analysis, iterative regeneration, and critique based evaluation. The results show that multi-agent systems can substantially improve translation quality, particularly when translating into statically typed languages like Java where syntax violations are more likely to cause failures.

Architectures that employed regeneration loops and validation against generated unit tests (e.g., Multi-Agent #1) demonstrated strong gains in resolving syntactic issues. However, their ability to catch and correct deeper semantic errors was limited by the quality of the generated tests. Introducing unstructured reasoning through an Analyzer Agent (Multi-Agent #2) led to performance regressions due to prompt contamination and ambiguity. In contrast, more structured approaches that leveraged AST analysis and explicit structured chain-of-thought reasoning (Multi-Agent #3) outperformed all others, achieving the highest accuracy with minimal regeneration, demonstrating the power of structured reasoning.

From a broader perspective, this work addresses a real and pressing challenge faced by large organizations: translating legacy systems written in outdated or low resource languages into modern ecosystems. In practice, legacy programming languages often suffer from limited training data compared to modern languages, reducing the reliability of traditional end-to-end LLM-based translation methods. Multi-agent systems offer a promising alternative by allowing LLMs to focus on narrower subtasks such as translation, validation, analysis, and correction, thereby compensating for gaps in training data.

Ultimately, this work shows that multi-agent systems can serve as a viable path forward for robust code translation in industrial settings. By combining structured analysis, agent specialization, and iterative refinement, these systems enable more accurate and scalable translation workflows, even under conditions of limited data availability for legacy source languages.

Acknowledgments

I would like to thank Professor Eldan Cohen for his valuable guidance and support throughout this project.

References

Anthropic. (2024). Introducing claude 3.5 sonnet [Accessed: 2025-04-29]. https://www.anthropic.com/news/claude-3-7-sonnet

Browne, J. (2016, June). Comparing the cost of rewrite to migration [Accessed: 2025-05-02]. https://www.mobilize.net/blog/comparing-the-cost-of-rewrite-to-migration

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., … Zaremba, W. (2021). Evaluating large language models trained on code. https://arxiv.org/abs/2107.03374

Chia, Y. K., Chen, G., Tuan, L. A., Poria, S., & Bing, L. (2023). Contrastive chain-of-thought prompting. https://arxiv.org/abs/2311.09277

Costa-Jussà, M. R., Farrús, M., Mariño, J. B., & Fonollosa, J. A. R. (2012). Study and comparison of rule-based and statistical catalan-spanish machine translation systems [Accessed: 2025-05-02]. *Computing and Informatics*, *31*(2), 245–270. https://repositori.upf.edu/bitstreams/8951cb08-54b8-44f9-a0ec-6aa892042218/download

Face, H. (2024). Bigcodebench leaderboard [Accessed: 2024-04-27]. https://huggingface.co/blog/leaderboard-bigcodebench

Group, B. C. N. (2021). Transcoder [Accessed: 2025-04-30].

LangChain-Team. (2024). Langgraph: Building multi-agent workflows with llms [Accessed: 2024-04-27]. https://github.com/langchain-ai/langgraph

Li, C., Liang, J., Zeng, A., Chen, X., Hausman, K., Sadigh, D., Levine, S., Fei-Fei, L., Xia, F., & Ichter, B. (2024). Chain of code: Reasoning with a language model-augmented code emulator. https://arxiv.org/abs/2312.04474

Li, J., Li, G., Li, Y., & Jin, Z. (2023). Structured chain-of-thought prompting for code generation. https://arxiv.org/abs/2305.06599

Maveli, N., Vergari, A., & Cohen, S. B. (2025). What can large language models capture about code functional equivalence? *Findings of the Association for Computational Linguistics: NAACL 2025*, 6865–6903. https://aclanthology.org/2025.findings-naacl.382/

Miller, B. (2022, September). *The evolution of machine translation: A timeline* [Accessed: 2025-05-02]. https://www.languageintelligence.com/post/the-evolution-of-machine-translation-a-timeline

Murdza, J. (2024). Humaneval results - gpt-3.5 turbo [Accessed: 2024-04-27].

Ng, A. (2024, March). Four ai agent strategies that improve gpt-4 and gpt-3.5 performance [Accessed: 2025-05-02]. https://www.deeplearning.ai/the-batch/how-agents-can-improve-llm-performance/

OpenAI. (2023). Best practices for prompt engineering with the openai api [Accessed: 2024-04-28]. https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-the-openai-api

Sahoo, P., Singh, A. K., Saha, S., Jain, V., Mondal, S., & Chadha, A. (2025). A systematic survey of prompt engineering in large language models: Techniques and applications. https://arxiv.org/abs/2402.07927

Sun, Q., Chen, N., Wang, J., Li, X., & Gao, M. (2024). Transcoder: Towards unified transferable code representation learning inspired by human skills. https://arxiv.org/abs/2306.07285

Sun, W., Fang, C., Miao, Y., You, Y., Yuan, M., Chen, Y., Zhang, Q., Guo, A., Chen, X., Liu, Y., & Chen, Z. (2023). Abstract syntax tree for programming language understanding and representation: How far are we? https://arxiv.org/abs/2312.00413

Wang, B., Li, R., Li, M., & Saxena, P. (2023). Transmap: Pinpointing mistakes in neural code translation. *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23)*, 1–13. https://doi.org/10.1145/3611643.3616322

Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A., & Zhou, D. (2023). Self-consistency improves chain of thought reasoning in language models. https://arxiv.org/abs/2203.11171

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., & Zhou, D. (2023). Chain-of-thought prompting elicits reasoning in large language models. https://arxiv.org/abs/2201.11903

Yadav, D., & Mondal, S. (2025, January). Evaluating pre-trained large language models on zero shot prompts for parallelization of source code [Accessed: 2025-05-02]. https://doi.org/10.2139/ssrn.5114036

Yang, Z., Liu, F., Yu, Z., Keung, J. W., Li, J., Liu, S., Hong, Y., Ma, X., Jin, Z., & Li, G. (2024). Exploring and unleashing the power of large language models in automated code translation. https://arxiv.org/abs/2404.14646

Yehudai, A., Eden, L., Li, A., Uziel, G., Zhao, Y., Bar-Haim, R., Cohan, A., & Shmueli-Scheuer, M. (2025). Survey on evaluation of llm-based agents. https://arxiv.org/abs/2503.16416

Yu, Z., Zhao, Y., Cohan, A., & Zhang, X.-P. (2024). Humaneval pro and mbpp pro: Evaluating large language models on self-invoking code generation. https://arxiv.org/abs/2412.21199

Yuan, Z., Chen, W., Wang, H., Yu, K., Peng, X., & Lou, Y. (2024). Transagent: An llm-based multi-agent system for code translation. https://arxiv.org/abs/2409.19894

Zhuo, T. Y. (2024, June). *Bigcodebench: The next generation of humaneval* [Accessed: 2025-05-02]. https://huggingface.co/blog/leaderboard-bigcodebench

Appendix

A. Github Repository

The accompanying GitHub repository contains the full implementation and evaluation results for all architectures discussed in this paper: https://github.com/Pyoussefpour/multi-agent-code-translation/tree/main

It should be noted that: The Evaluator Agent is split into two code files: Unit Test Generator Agent, and Evaluator Tool for the evaluations for some cases more than 50 test file were evaluated, but the reported numbers in this paper are based on the first 50 tests.

B. Test Files used for Evaluation

File Names
ANALYSIS_OF_ALGORITHMS_SET_2_ASYMPTOTIC_ANALYSIS
BELL_NUMBERS_NUMBER_OF_WAYS_TO_PARTITION_A_SET
BINARY_SEARCH
CHECK_GIVEN_SENTENCE_GIVEN_SET_SIMPLE_GRAMMER_RULES
CHECK_IF_ALL_THE_ELEMENTS_CAN_BE_MADE_OF_SAME_PARITY_BY_INVERTING_ADJACENT_ELEMENTS
CHECK_WHETHER_TWO_STRINGS_ARE_ANAGRAM_OF_EACH_OTHER
CHOCOLATE_DISTRIBUTION_PROBLEM
CONSTRUCT_LEXICOGRAPHICALLY_SMALLEST_PALINDROME
CONVERTING_ONE_STRING_USING_APPEND_DELETE_LAST_OPERATIONS
COUNT_NUMBER_OF_SOLUTIONS_OF_X2_1_MOD_P_IN_GIVEN_RANGE
COUNT_OF_OCCURRENCES_OF_A_101_PATTERN_IN_A_STRING
COUNT_OPERATIONS_MAKE_STRINGAB_FREE
COUNT_PALINDROME_SUB_STRINGS_STRING
COUNT_POSSIBLE_WAYS_TO_CONSTRUCT_BUILDINGS
COUNT_SUBARRAYS_EQUAL_NUMBER_1S_0S
DISTRIBUTING_M_ITEMS_CIRCLE_SIZE_N_STARTING_K_TH_POSITION
DYNAMIC_PROGRAMMING_SET_17_PALINDROME_PARTITIONING_1
FIND_A_ROTATION_WITH_MAXIMUM_HAMMING_DISTANCE
FIND_DIFFERENCE_BETWEEN_SUMS_OF_TWO_DIAGONALS
FIND_EQUAL_POINT_STRING_BRACKETS
FIND_INDEX_OF_AN_EXTRA_ELEMENT_PRESENT_IN_ONE_SORTED_ARRAY_1
FIND_POSITION_GIVEN_NUMBER_AMONG_NUMBERS_MADE_4_7
FIND_ROTATION_COUNT_ROTATED_SORTED_ARRAY_1
FIND_SUM_NODES_GIVEN_PERFECT_BINARY_TREE_1
FIND_SUM_NON_REPEATING_DISTINCT_ELEMENTS_ARRAY
FIND_THREE_ELEMENT_FROM_DIFFERENT_THREE_ARRAYS_SUCH_THAT_THAT_A_B_C_K_1
FRIENDS_PAIRING_PROBLEM
FUNCTION_COPY_STRING_ITERATIVE_RECURSIVE_1
GOLD_MINE_PROBLEM
GOOGLE_CASE_GIVEN_SENTENCE
HARDY_RAMANUJAN_THEOREM
HIGHWAY_BILLBOARD_PROBLEM
HOW_TO_CHECK_IF_A_GIVEN_ARRAY_REPRESENTS_A_BINARY_HEAP
HYPERCUBE_GRAPH
LARGEST_SUBARRAY_WITH_EQUAL_NUMBER_OF_0S_AND_1S_1
LONGEST_PALINDROME_SUBSEQUENCE_SPACE
LONGEST_SUBSEQUENCE_SUCH_THAT_DIFFERENCE_BETWEEN_ADJACENTS_IS_ONE
MARKOV_MATRIX
MAXIMUM_AVERAGE_SUM_PARTITION_ARRAY
MAXIMUM_NUMBER_2X2_SQUARES_CAN_FIT_INSIDE_RIGHT_ISOSCELES_TRIANGLE
MAXIMUM_PRODUCT_INCREASING_SUBSEQUENCE
MAXIMUM_PRODUCT_OF_4_ADJACENT_ELEMENTS_IN_MATRIX
MAXIMUM_PROFIT_BY_BUYING_AND_SELLING_A_SHARE_AT_MOST_K_TIMES_1
MAXIMUM_PROFIT_BY_BUYING_AND_SELLING_A_SHARE_AT_MOST_TWICE
MAXIMUM_SUM_PAIRS_SPECIFIC_DIFFERENCE_1
MEDIAN_OF_TWO_SORTED_ARRAYS
MINIMUM_COST_CUT_BOARD_SQUARES
MINIMUM_ROTATIONS_REQUIRED_GET_STRING
MINIMUM_SUM_SUBSEQUENCE_LEAST_ONE_EVERY_FOUR_CONSECUTIVE_ELEMENTS_PICKED
MINIMUM_SWAPS_REQUIRED_BRING_ELEMENTS_LESS_EQUAL_K_TOGETHER

Table 6: List of test file used in evaluation.

C. Test File Structure

The following represent what the test file in the GeeksforGeeks data set looks like.

```python
def f_gold(str1, str2, k):
    if (len(str1) + len(str2)) < k:
        return True
    commonLength = 0
    for i in range(0, min(len(str1), len(str2)), 1):
        if str1[i] == str2[i]:
            commonLength += 1
        else:
            break
    if (k - len(str1) - len(str2) + 2 * commonLength) % 2 == 0:
        return True
    return False

# TOFILL

if __name__ == '__main__':
    param = [
        ('ZNHGro','jAdbtDUYQu',3,),
        ('382880806774','65565',10,),
        ('0','00100010100',2,),
        ('1xHTRFCTSQ','sViXYE',89,),
        ('6399914758','780990121',9,),
        ('01100011100000','0100',0,),
        ('WkGqlob','NpQVdXzEtUZy',6,),
        ('46974006151','74438',11,),
        ('1001001','1000010',15,),
        ('IJQ','nFOHAeYEAp',42,)
    ]
    n_success = 0
    for i, parameters_set in enumerate(param):
        if f_filled(*parameters_set) == f_gold(*parameters_set):
            n_success += 1
    print("#Results: %i, %i" % (n_success, len(param)))
```

For Translation, the f_gold function is passed as the input code and the rest of the code is passed as sample inputs.

```python
def f_gold(str1, str2, k):
    if (len(str1) + len(str2)) < k:
        return True
    commonLength = 0
    for i in range(0, min(len(str1), len(str2)), 1):
        if str1[i] == str2[i]:
            commonLength += 1
        else:
            break
    if (k - len(str1) - len(str2) + 2 * commonLength) % 2 == 0:
        return True
    return False
```

D. Evaluation Script

For the multi-agent's translation evolution, the translated code replaces the #TOFILL (or //TOFILL in Java) in the prallel test file to create script like the following. When executed it evaluates the translated function against the parallel function using the provided unit tests by the dataset.

In the following example the input function was in Java and it was translated to python and the following is the parallel python file with the translated code added to it.

```python
def f_gold(str1, str2, k):
    if (len(str1) + len(str2)) < k:
        return True
    commonLength = 0
    for i in range(0, min(len(str1), len(str2)), 1):
        if str1[i] == str2[i]:
            commonLength += 1
        else:
            break
    if (k - len(str1) - len(str2) + 2 * commonLength) % 2 == 0:
        return True
    return False


# Translated Function

def can_transform_with_k_ops(str1, str2, k):
    # If the total length is less than k, you can delete all and rebuild
    if len(str1) + len(str2) < k:
        return True

    # Count common prefix length
    common_prefix_len = 0
    for i in range(min(len(str1), len(str2))):
        if str1[i] == str2[i]:
            common_prefix_len += 1
        else:
            break

    # Compute how many actual operations are needed:
    # (delete rest of str1 + add rest of str2)
    min_ops_needed = len(str1) + len(str2) - 2 * common_prefix_len

    # Check if the difference can be covered exactly or by extra redundant ops
    return (k - min_ops_needed) % 2 == 0 and k >= min_ops_needed

# Translated Function


if __name__ == '__main__':
    param = [
        ('ZNHGro','jAdbtDUYQu',3,),
        ('382880806774','65565',10,),
        ('0','00100010100',2,),
        ('lxHTRFCTSQ','sViXYE',89,),
        ('6399914758','780990121',9,),
        ('01100011100000','0100',0,),
        ('WkGqlob','NpQVdXzEtUZy',6,),
        ('46974006151','74438',11,),
        ('1001001','1000010',15,),
        ('IJQ','nFOHAeYEAp',42,)
    ]
    n_success = 0
    for i, parameters_set in enumerate(param):
        if f_filled(*parameters_set) == f_gold(*parameters_set):
            n_success += 1
    print("#Results: %i, %i" % (n_success, len(param)))
```