ICAM-Trans: Implicit-Context-Aware Multi-Agent Framework for Function-Level Code Translation

RUOLIN CHEN, JIEXIN WANG, LIUWEN CAO, YI CAI, and MENGCHEN ZHAO, School of Software Engineering, South China University of Technology, Guangzhou, China and Joint Guangdong-Hong Kong-Macao Research Laboratory of Big Data and Robotic Intelligence, Ministry of Education, Guangzhou, China and Key Laboratory of Big Data and Intelligent Robot (SCUT), Ministry of Education, Guangzhou, China

With the rapid advancement of Large Language Models (LLMs), code translation has become a critical yet challenging task in software engineering. Existing evaluations of this task remain unreliable due to two key limitations: the absence of specialized function-level code translation benchmarks and flawed evaluation pipelines. Specifically, current pipelines lack post-processing mechanisms to extract target code from LLMs' inherently unstable output formats, directly resulting in non-executable translations and false negatives in evaluation. To address these issues, we introduce FuncTransEval, an enhanced multilingual benchmark. It comprises 400 × 4 functions across four programming languages (Python, C++, Java, and JavaScript) and 12 bidirectional translation pairs, each paired with executable unit tests for fine-grained functional validation. Using FuncTransEval, we systematically assess LLMs' true capabilities in code translation and identify their core limitations: inadequate understanding of the implicit information such as function logic, type specifications,

This research is supported by the Science and Technology Planning Project of Guangdong Province (2025B0101120003), the Guangdong Provincial Fund for Basic and Applied Basic Research—Regional Joint Fund Project (Key Project) (2023B1515120078), National Natural Science Foundation of China (62476097, 62402185), Guangdong Provincial Natural Science Foundation for Outstanding Youth Team Project (2024B1515040010), the Fundamental Research Funds for the Central Universities, South China University of Technology (x2rjD2250190), and the Fundamental Research Funds for the Central Universities (2025ZYGXZR064).

Authors' Contact Information: Ruolin Chen (corresponding author), School of Software Engineering, South China University of Technology, Guangzhou, China and Joint Guangdong-Hong Kong-Macao Research Laboratory of Big Data and Robotic Intelligence, Ministry of Education, Guangzhou, China and Key Laboratory of Big Data and Intelligent Robot (SCUT), Ministry of Education, Guangzhou, China; e-mail: rolachen1999@163.com; Jiexin Wang, School of Software Engineering, South China University of Technology, Guangzhou, China and Joint Guangdong-Hong Kong-Macao Research Laboratory of Big Data and Robotic Intelligence, Ministry of Education, Guangzhou, China and Key Laboratory of Big Data and Intelligent Robot (SCUT), Ministry of Education, Guangzhou, China; e-mail: jiexinwang@scut.edu.cn; Liuwen Cao, School of Software Engineering, South China University of Technology, Guangzhou, China and Joint Guangdong-Hong Kong-Macao Research Laboratory of Big Data and Robotic Intelligence, Ministry of Education, Guangzhou, China and Key Laboratory of Big Data and Intelligent Robot (SCUT), Ministry of Education, Guangzhou, China; e-mail: caoliuwencc@163.com; Yi Cai, School of Software Engineering, South China University of Technology, Guangzhou, China and Joint Guangdong-Hong Kong-Macao Research Laboratory of Big Data and Robotic Intelligence, Ministry of Education, Guangzhou, China and Key Laboratory of Big Data and Intelligent Robot (SCUT), Ministry of Education, Guangzhou, China; e-mail: ycai@scut.edu.cn; Mengchen Zhao, School of Software Engineering, South China University of Technology, Guangzhou, China and Joint Guangdong-Hong Kong-Macao Research Laboratory of Big Data and Robotic Intelligence, Ministry of Education, Guangzhou, China and Key Laboratory of Big Data and Intelligent Robot (SCUT), Ministry of Education, Guangzhou, China; e-mail: zzmc@scut.edu.cn.
*For correspondence, please contact Yi Cai.



This work is licensed under Creative Commons Attribution International 4.0.

© 2026 Copyright held by the owner/author(s).
ACM 1557-7392/2026/11-ART350
https://doi.org/10.1145/3810952

and cross-language differences. To mitigate these shortcomings, we propose ICAM-Trans (Implicit-Context-Aware Multi-Agent Framework for function-level code translation). ICAM-Trans is a hierarchical, autonomous multi-agents architecture that explicitly uncovers and leverages these implicit contextual information during translation. It employs a Translation Orchestrate Agent (TOA) to autonomously coordinate four specialized Context Analysis Agents (CAAs), which conduct pre-translation analyses on different aspects. Insights from these analyses guide a Context-Aware Translation Agent (CTA) to generate semantically faithful target code. Experiments on FuncTransEval demonstrate that ICAM-Trans consistently outperforms strong baselines, validating its effectiveness in achieving high-fidelity and interpretable function-level code translation.

CCS Concepts: • **Computing methodologies** → **Artificial intelligence**; • **Software and its engineering** → **Software creation and management**;

Additional Key Words and Phrases: code translation, multi-agents, large language model, benchmark

ACM Reference format:
Ruolin Chen, Jiexin Wang, Liuwen Cao, Yi Cai, and Mengchen Zhao. 2026. ICAM-Trans: Implicit-Context-Aware Multi-Agent Framework for Function-Level Code Translation. *ACM Trans. Softw. Eng. Methodol.* 35, 11, Article 350 (November 2026), 33 pages.
https://doi.org/10.1145/3810952

1 Introduction

Code translation, which refers to converting source code from one programming language to another, has long been a key research topic in software engineering [3]. It enables developers to reuse existing codebases across languages, reducing redundant implementation efforts and significantly improving software development efficiency [8]. Beyond improving productivity, code translation plays an essential role in software modernization. By converting code from older languages to newer, more robust ones [2], it allows developers to migrate legacy systems to modern programming environments that are easier to maintain and upgrade [4, 20, 24].

With the rapid advancement of **Large Language Models (LLMs)**, they have achieved remarkable success in code generation tasks. Both general-purpose LLMs (e.g., GPT-4 [1], Llama [27]) and specialized code LLMs (e.g., Qwen2.5-Coder [10], CodeLlama [21]) have demonstrated strong capabilities in generating functional code snippets. However, when it comes to code translation tasks, their effectiveness is widely considered unsatisfactory, as highlighted in prior work like *Lost in Translation* [17]. Nonetheless, this conclusion warrants scrutiny, as existing research suffers from two critical methodological limitations that may have skewed assessments of LLM performance [15]. First, there is a lack of specialized, function-level code translation benchmarks designed to rigorously evaluate LLM capabilities in this task. Second, evaluation pipelines are inherently fragile, they lack essential post-processing mechanisms to extract executable target code from LLMs' inherently unstable output formats. These flaws directly lead to non-executable translations and false negatives in performance measurement thereby underestimating LLMs' true potential and casting doubt on the reliability of claims about their inadequacy in code translation. Given these gaps, a key question arises: what is the genuine capability of LLMs in function-level code translation when evaluated with proper datasets and robust pipelines? Addressing this question is crucial for understanding whether LLMs can be trusted for cross-language code migration and for guiding future research to enhance their translation performance.

To address these methodological gaps and establish a reliable foundation for evaluating LLMs' translation capabilities, we construct *FuncTransEval*, an enhanced multilingual benchmark specifically tailored for function-level code translation, derived from G-TransEval [11]. It contains 400

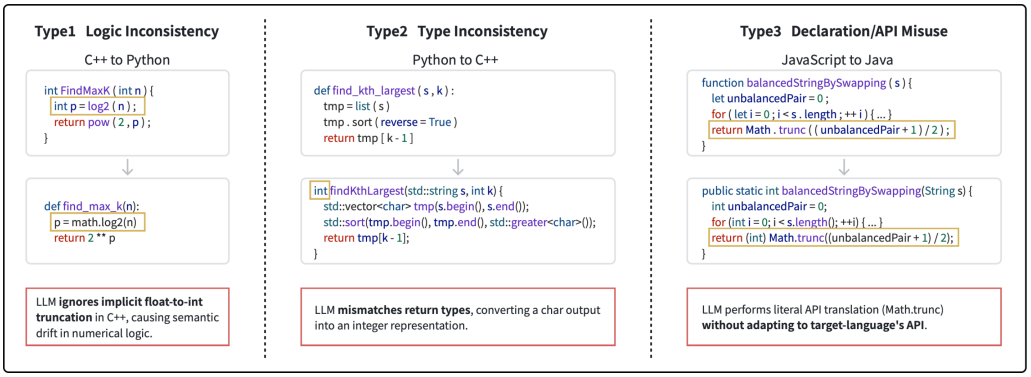


Fig. 1. Typical Errors occur in direct LLM translation.

functions per language (totaling 400×4 functions) across *Python*, *C++*, *Java*, and *JavaScript*, covering 12 bidirectional translation pairs. Each function is paired with executable unit tests, enabling fine-grained validation of both syntactic correctness and functional behavior. Complementing this curated dataset, we redesign the evaluation pipeline by integrating refined prompting strategies and structured post-processing mechanisms, critical for extracting executable code from LLMs' unstable outputs. This dual improvement provides a fairer, more trustworthy framework for comparing model performance and uncovering LLMs' true translation capabilities.

Our analysis using the reliable FuncTransEval benchmark reveals that while LLM-based translators significantly outperform non-LLM baselines (e.g., TransCoder [22] and TransCoder-ST [23]), they still suffer from critical, recurring limitations. As illustrated in Figure 1, these flaws cluster into three core categories: (1) *Logic Inconsistency*: Misinterpreting language-specific semantic nuances. For example, LLM ignores the implicit *float*-to-*int* truncation in C++, which introduces semantic drift in numerical logic (Figure 1, Type 1). (2) *Type Inconsistency*: Incorrectly inferring the type information. In dynamically typed languages (e.g., *Python*, *JavaScript*), where parameter types are not explicitly declared, the LLM may misinfer type semantics, for instance, misclassifying a variable's type during cross-language translation, which results in type mismatches between source and target code (Figure 1, Type 2). (3) *Declaration/API Misuse*: Wrongly use a non-exist Library or API. For instance, LLM fails to adapt the library/API to target-language ecosystems, generates non-existent Java functions *Math.trun* (Figure 1, Type 3). Notably, logic and type inconsistencies collectively account for nearly 85% of all translation errors, revealing a fundamental limitation: *Effective code translation requires not only syntactic alignment but also a deeper understanding of the function's implicit semantics and cross-language contextual dependencies.*

Motivated by this insight, we propose *ICAM-Trans*, an *I*mplicit-*C*ontext-*A*ware *M*ulti-agent framework for function-level code translation. ICAM-Trans is a hierarchical, autonomous multi-agents architecture that explicitly uncovers and utilizes hidden contextual information during translation. It is organized into three cooperative layers, each coordinated by one or multiple specialized agents. At the Translation Orchestration Layer, a **Translation Orchestrate Agent (TOA)** evaluates the source function and autonomously determines the optimal analysis workflow. Guided by the TOA, the Pre-Translation Analysis Layer dynamically activates the required **Context Analysis Agents (CAAs)**, including **Parameter Analysis Agent (PAA)**, **Return-value Analysis Agent (RAA)**, **Logic Analysis Agent (LAA)**, and **Language Difference Analysis Agent (DAA)** to perform targeted analyses in parallel. The results are then aggregated and passed to the Context-Aware Translation Layer, where the **Context-Aware Translation Agent (CTA)**

generates the final translated code. Through this explicit pre-translation reasoning process, ICAM-Trans effectively mitigates implicit semantic inconsistencies by integrating structured analysis with context-aware translation, thereby achieving more consistent and semantically faithful translations across programming languages. Comprehensive experiments on FuncTransEval demonstrate that ICAM-Trans consistently outperforms strong baselines across pass@1, pass@10, and **Average Computational Accuracy (ACA)**. Moreover, ablation studies verify that each agent within ICAM-Trans contributes distinct yet complementary improvements, collectively enhancing translation reliability and robustness. Our FuncTransEval dataset, the automated testing pipeline, and the complete ICAM-Trans framework implementation are opened via an anonymous repository: https://doi.org/10.5281/zenodo.19556433.

The main contributions of this work are summarized as follows:

— *A Reliable Benchmark for Function-Level Code Translation.* We construct *FuncTransEval*, an enhanced multilingual benchmark derived from G-TransEval. It includes 400×4 semantically consistent functions implemented in four programming languages (Python, C++, Java, and JavaScript), accompanied by automatically validated unit tests and standardized formatting. We integrate testing and assessment into a unified pipeline and incorporate widely adopted execution-based metrics (*pass@k* and *ACA*) to ensure convenient, comprehensive, and fair evaluation.

— *An Implicit-Context-Aware Multi-Agent Framework for Translation.* We propose *ICAM-Trans*, a novel multi-agent framework that introduces explicit pre-translation reasoning. It integrates a *TOA* to dynamically coordinate multiple specialized *CAAs*, responsible for parameter, return_value, logic, and language difference analyses, and guides a *CTA* to produce semantically faithful target code.

— *Comprehensive Evaluation and Interpretability Analysis.* Extensive experiments on FuncTransEval demonstrate that ICAM-Trans consistently outperforms strong baselines in the execution-based metrics. Ablation and case studies further reveal how each agent contributes complementary reasoning capabilities, validating the interpretability and effectiveness of the proposed framework.

2 Related Works

2.1 Evolution of Code Translation Techniques

The task of code translation aims to convert source code from one programming language to another while preserving the functional and semantic integrity of the original implementation [33]. Early research primarily relied on rule-based transformation systems that manually defined mappings between syntax constructs and grammar rules [12]. While these approaches offered deterministic behavior, they required extensive human engineering, making them difficult to generalize to complex or dynamically typed programming languages. With the emergence of Neural Machine Translation, researchers began to apply deep learning to code translation. Neural models such as RNN-based and Transformer-based architectures [2, 13, 16] introduced end-to-end learning pipelines that improved lexical and structural alignment between languages. However, these models often failed to preserve deeper program semantics. Specifically, they exhibited issues with logical consistency, readability, and maintainability—problems that stem from treating code as plain text rather than as structured, executable programs. With the explosion of transformer modeling, deep learning-based approach such as CodeBERT [7], CodeT5 [28], and TransCoder [22] further improved translation quality by capturing long-range dependencies and leveraging large pretraining corpora [7, 22, 23, 28]. Despite these advances, their success heavily depends on the availability of large, high-quality bilingual code datasets, and the associated training costs remain

prohibitive. Due to the outstanding performance of LLMs in natural language translation and code generation, researchers have explored LLM-based code translation methods in recent years. The paper published by Pan et al. [17] from IMB Research Institute in 2024 discovered that LLMs are yet to be reliably used to automate code translation.

2.2 Evolution of Code Translation Benchmarks

The evolution of code translation evaluation has progressed from leveraging general-purpose parallel corpora to developing task-specific benchmarks. Initially, researchers utilized large-scale datasets like CodeNet [19] and xCodeEval [14], which offer massive multilingual corpora. However, these datasets primarily evaluate translation at the program level, relying on standard input/output matching, and often lack the strictly verified, one-to-one semantic equivalence required for precise, fine-grained assessment. Subsequently, the community adapted multilingual function-level datasets originally designed for code generation, such as Humaneval-x [32] and MultiPL-E [5], to support cross-lingual translation tasks. While beneficial, applying generation-focused datasets directly to translation tasks introduces inherent limitations. To address these gaps, researchers proposed specialized evaluation benchmarks explicitly tailored for function-level code translation, including TransCoder-test [22], G-TransEval [11], and EvalPlus [17]. More recently, the field has expanded to evaluate higher levels of structural complexity, introducing class-level and repository-level benchmarks such as ClassEval-T [18] and RepoTransBench [29]. While these benchmarks are crucial for evaluating complex inter-method relationships, dependencies, and shared class states, they present significant challenges for isolated error diagnosis. When an LLM fails to translate a complex class, it is highly difficult to disentangle whether the root cause lies in a structural misunderstanding (e.g., mismanaging class state) or in atomic, intra-function semantic errors. Therefore, fine-grained, strictly isolated function-level benchmarks remain orthogonally essential. By stripping away external dependencies, function-level evaluation provides a controlled testbed to rigorously diagnose the core semantic translation capabilities of LLMs. Ensuring high-fidelity, semantically equivalent translations at this atomic function level provides the robust execution baseline strictly necessary before scaling to the broader complexities of real-world software migration.

2.3 LLM-Based Agents for Code Translation

Recent advances in agentic LLM architectures have opened new possibilities for improving the reasoning and adaptability of code translation systems. Unlike conventional end-to-end translation models, LLM-based agents decompose the translation process into multiple reasoning or interaction stages, enabling more structured decision-making and self-refinement. Early works such as Reflexion [25] and AgentCoder [9] explored using LLMs as self-reflective agents for code generation and debugging, where the model evaluates its own output and performs corrective iterations based on error signals. Extending this concept to translation, studies like TransAgent [31] have demonstrated that multi-step reasoning and verification can improve cross-language consistency and functional correctness. These systems often incorporate components such as planning agents for task decomposition, analysis agents for semantic alignment, and verification agents for unit-test-based correction. However, most of these approaches still rely on ad-hoc coordination rules or external execution feedback, which limits their interpretability and scalability across programming languages. This gap motivates our design of ICAM-Trans, a Context-Aware Multi-Agent Framework that integrates explicit pre-translation reasoning into the translation pipeline. By coordinating specialized analysis agents before generation, ICAM-Trans aims to achieve both interpretability and high-fidelity code translation without relying on iterative trial-and-error mechanisms.

Table 1. Code Translation Accuracy across Different Datasets and Language Pairs

Benchmark	C++	Java	JavaScript	Python	Java	JavaScript	Python	C++	JavaScript	Python	C++	Java
	Python			C++			Java			JavaScript		
PolyHumanEval	68.29	87.8	85.98	10.98	18.29	50.00	0	0	0	84.15	86.59	87.8
G-TransEval	93.00	95.50	94.75	17.50	27.50	11.25	0	0	0	90.50	91.25	90.75
FuncTransEval (Ours)	92.68	94.40	94.83	71.87	70.71	71.57	86.45	93.78	79.55	92.87	92.26	91.52

Values are represented in %.

3 Dataset Construction

3.1 Motivation for FuncTransEval

Although numerous datasets in the software engineering community can be used for code translation research, most are not designed to rigorously evaluate function-level translation capabilities of LLMs. Large-scale benchmarks such as CodeNet [19] and xCodeEval [14] primarily assess program-level correctness, where evaluation depends on complete executable programs rather than isolated function semantics. Consequently, they are not suitable for fine grained diagnosis at the function level. In contrast, multilingual function-level datasets such as PolyHumanEval [26] and HumanEval-X [32] were originally constructed for code generation tasks, not cross lingual translation. When directly repurposed for translation, their generation-oriented design can introduce systemic false negatives. Even translation specific benchmarks such as G-TransEval [11] present practical limitations, including unexecutable Python one liners and evaluation pipelines that require manual multi-round testing.

As shown in Table 1, when evaluating Qwen2.5-Coder-7B-Instruct on these existing datasets, the results exhibit severe anomalies. For example, the accuracy for translating into *Java* is consistently 0% across all source languages in both PolyHumanEval and G-TransEval. In addition, the accuracy for translating into *C++* is unexpectedly low, and the performance variance across language pairs is highly irregular. Such catastrophic outcomes do not reflect the model's actual translation capability, indicating that these benchmarks fail to provide an objective assessment of LLM-based translation performance. In contrast, FuncTransEval produces substantially more stable and reasonable evaluation results.

The abnormal results observed on existing datasets stem from three major structural and semantic limitations, as illustrated in Figure 2. *Different Encapsulation Methods During Testing.* As shown in CASE1, programming languages adopt fundamentally different encapsulation paradigms. For example, Python supports standalone functions, whereas Java requires methods to be embedded within class structures. Standard evaluation scripts may fail because the raw function body generated by the LLM cannot be executed directly within the target language testing environment. *Inconsistent Function Naming Conventions.* As shown in CASE2, many existing test harnesses rely on fixed and hard-coded function names. If an LLM preserves the original naming style during translation, such as retaining snake_case style instead of converting to camelCase style, the generated function will not match the expected signature, leading to false negatives. *Inconsistent Intrinsic Semantics.* As shown in CASE3, some translation pairs contain latent semantic inconsistencies, particularly in boundary behaviors or return types. For example, a Python implementation may return a binary integer value, while the corresponding C++ reference returns a binary representation in string form. Such discrepancies violate strict one-to-one functional equivalence and can distort translation evaluation.

To address these critical evaluation bottlenecks, we introduce *FuncTransEval*. Our benchmark resolves the first two issues by incorporating a unified test suite equipped with automated AST based

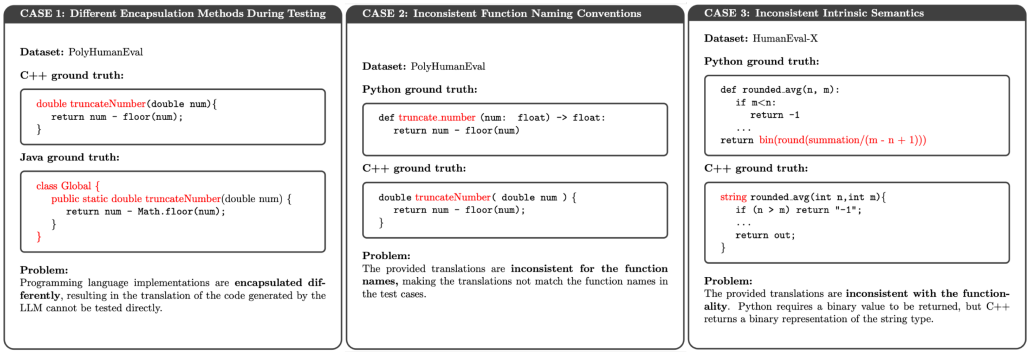


Fig. 2. The examples of the limitation of existing dataset for code translation task.

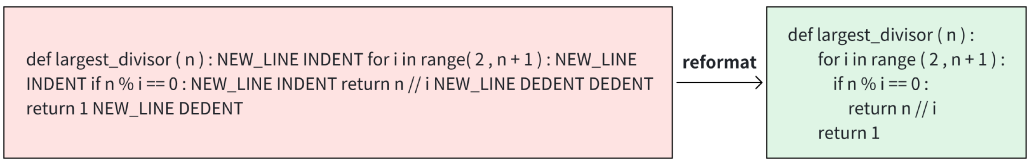


Fig. 3. An example of reformat process of *Python*.

extraction mechanisms that normalize structural differences and harmonize function signatures prior to execution. Furthermore, we mitigate the third issue through a rigorous test case augmentation pipeline that exposes latent semantic divergences, especially under boundary conditions. All identified inconsistencies are manually verified and corrected to ensure complete semantic equivalence across language pairs.

3.2 Dataset Reconstruction and Automated Testing

To ensure high-quality, executable, and semantically consistent code across multiple programming languages, we conducted a thorough reconstruction of the original G-TransEval dataset. First, we focused on the Python portion of the dataset, which initially existed in a structured but non-executable one-line format. To transform such text into standardized code, we designed a regular expression-based parsing and substitution pipeline. This automated process systematically restored standard Python syntax, reconstructed correct indentation hierarchies, and ensured full structural integrity of function bodies, as shown in Figure 3. Beyond syntax repair, we performed a comprehensive semantic alignment across all four programming languages (Python, C++, Java, and JavaScript). Some translation groups in the original dataset exhibited functional divergence caused by differences in control flow, operator usage, or built-in library behaviors. To address these inconsistencies, we built a semantic validation pipeline that executed all implementations within a translation group against a unified test suite. By comparing run-time outputs, we accurately identified functionally inconsistent cases, which were then manually reviewed and corrected. This iterative refinement ensured that the final dataset preserved strict one-to-one functional equivalence across all languages. After reconstruction and validation, the resulting dataset comprised 400 fully parallel translation groups, each containing aligned implementations in Python, C++, Java, and JavaScript.

To ensure consistent and reproducible evaluation, we developed a unified and fully automated multi-language testing framework that provides a complete post-processing and testing pipeline.

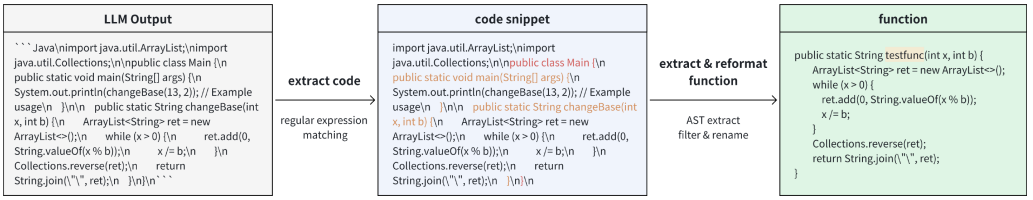


Fig. 4. An example of the post-processing flow.

The post-processing stage begins with regex-based code extraction, which isolates the translated code from the raw LLM output. Next, an AST-based function extraction step identifies and retrieves the target function from the extracted code. During this process, auxiliary components such as main functions (commonly present in *C++* and *Java*) are filtered out, and the target function is renamed to a standardized identifier $testfunc$ for unified testing. Figure 4 demonstrates an example of this post-processing flow. During the testing process, all previous manual steps, such as two-round Java testing, are fully automated. The testing framework further supports multi-threaded execution, enabling efficient code validation across large translation datasets. Finally, it incorporates a dual-level reporting mechanism that automatically generates fine-grained results for each translation pair and aggregated benchmark statistics, ensuring a consistent, efficient, and fully automated evaluation process.

3.3 Test Case Expansion and Semantic Validation

To enhance test coverage and robustness, we designed an LLM-based test augmentation pipeline that systematically expands each function's test suite from an average of 5 to 13 cases. The test case augmentation process begins by selecting Java as the reference language due to its static typing and strict syntax, which enable reliable test execution and precise output collection. Using Qwen2.5-Coder-7B-Instruct, we generate diverse input cases covering boundary conditions, edge cases, type variations, and exceptional scenarios. These generated inputs are first executed on the Java implementation to obtain ground truth input output pairs. The resulting pairs are then cross validated across all four language implementations to ensure consistency.

Importantly, this expanded testing, particularly the inclusion of boundary conditions and negative inputs, reveals latent semantic drift in the original corpus. We find that approximately 5% of the translation pairs exhibit subtle functional inconsistencies that were not detected by the original, limited test suites. Consider the *CountDigit* function as an example. The original C++ and Python ground truth implementations are shown in Figure 5(a) and (b). Although the Python version passes basic positive integer tests, expanded boundary testing with negative inputs exposes a critical semantic discrepancy. In C++, integer division using the operator / truncates toward zero (e.g., $-5/10$ evaluates to 0). While in Python, floor division using the operator // truncates toward negative infinity (e.g., $-5//10$ evaluates to -1), which leads to infinite recursion in this function. To resolve this issue, we manually revise the Python implementation to use an explicit type cast, thereby preserving the C++ truncation semantics, as shown in Figure 5(c).

Finally, all validated test samples are deduplicated, standardized, and revalidated across the corrected implementations to guarantee strict one to one functional equivalence. The finalized test cases are then integrated into the benchmark, substantially improving its ability to rigorously evaluate translation fidelity and robustness.

```cpp
int CountDigit(int n) {
    if (n == 0) { return 0; }
    return 1 + CountDigit(n / 10);
}
```

(a) original C++ ground truth

```python
def count_digit(n):
    if n == 0:
        return 0
    return 1 + count_digit(n // 10)
```

(b) original Python ground truth

```python
def count_digit(n):
    if n == 0:
        return 0
    return 1 + count_digit(int(n / 10))
```

(c) fix Python ground truth

Fig. 5. An example of Semantic Validation.

3.4 Metric Integration

To better capture translation fidelity, we replace static text-based metrics (e.g., CodeBLEU) with execution-driven indicators that evaluate models from multiple dimensions.

— *pass@k* [6]: Measures the probability that at least one of the top-k generated translations passes all test cases. This metric quantifies probabilistic translation success and reflects the model's ability to produce correct code within limited decoding attempts. Its formula is as follows:

$$pass@k = \frac{1}{N}\sum_{i=1}^{N}\left(1 - \frac{\binom{n_i - c_i}{k}}{\binom{n_i}{k}}\right), \tag{1}$$

where N is the total number of test samples, n_i is the number of generated codes for the ith sample, c_i is the number of codes that pass the test case, and k is the number of candidates considered.

— *ACA:* Represents the average proportion of test cases passed per translated function. It provides a fine-grained, continuous measure of functional correctness, complementing the binary nature of pass@k. Its formula is as follows:

$$ACA = \frac{1}{N}\sum_{i=1}^{N}\frac{p_i}{t_i}, \tag{2}$$

where N is the total number of test samples, p_i is the number of test cases that the ith sample passes, and t_i is the total number of test cases for the ith sample.

3.5 Dataset Summary

The FuncTransEval benchmark represents a significant advancement over existing code translation datasets by addressing long-standing limitations in structural consistency, semantic fidelity, test coverage, and evaluation automation. It comprises 400 fully parallel translation groups, each containing functionally equivalent implementations in four widely used programming languages: Python, C++, Java, and JavaScript. This yields a total of 1,600 high-quality, executable functions, meticulously aligned to ensure strict one-to-one semantic correspondence across all language pairs.

As summarized in Table 2, FuncTransEval addresses the fundamental evaluation gaps present in prior works. While large-scale corpora focus on general problem solving and previous translation benchmarks suffer from divergent behaviors or execution failures, our dataset guarantees manually verified semantic equivalence and introduces a unified AST-based execution pipeline to seamlessly

Table 2. Dataset Statistics

Property	CodeNet/xCodeEval	PolyHumanEval/HumanEval-X	G-TransEval	FuncTranEval (Ours)
Primary Task Focus	General Problem Solving	Code Generation	Code Translation	Code Translation
Testing Granularity	Program level (I/O)	Function level	Function level	Function level
Executable Code	✓	✓	✗ (e.g., Python one-liners)	✓
Functions Semantically Equivalent	✗ (Algorithm variations)	✗ (Divergent behaviors)	✗ (Edge cases)	✓ (Manually verified)
Unified Execution Pipeline	✗	✗ (Fails on naming/encapsulation)	✗ (Manual for Java)	✓ (AST-based extraction)
Tasks	-	164	400	400
Aligned Test Cases	✓	✗	✓	✓
Average Test Cases	Varies widely	~ 7	5	~ 13
Evaluation Metrics	Accuracy/Execution Time	pass@k	CA, BLEU, CodeBLEU	pass@k, ACA

normalize testing. Furthermore, it enables rigorous functional validation by expanding the average number of aligned test cases per function to 13 and adopts modern, execution-driven evaluation metrics (pass@k and ACA) in place of outdated static measurements. By providing a reproducible, fine-grained, and bias-resistant foundation, FuncTransEval establishes the fair testbed necessary for assessing the genuine capabilities of LLMs. In the following section, we leverage this unified pipeline to conduct a preliminary study, exposing the systematic failure patterns inherent in direct LLM translation and directly motivating the design of our implicit-context-aware framework.

4 Preliminary Study and Research Questions

Although LLMs have recently demonstrated remarkable progress in code generation tasks, they seems yet to be reliably used to automate code translation. As highlighted in [17], even state-of-the-art LLMs still struggle with function-level translation. However, this results is not reliable enough as the evaluation pipelines in prior works remain methodologically fragile. They rely on ambiguous instruct prompts and omit essential post-processing steps (e.g., code extraction, interface alignment). As a result, model performance is frequently underestimated or misrepresented, obscuring the true capabilities and limitations of LLMs in code translation tasks [15]. To overcome these issues, we introduce a standardized evaluation pipeline, referred to as Standard, built upon our proposed benchmark FuncTransEval. This pipeline is designed to ensure fair, stable, and reproducible evaluation by integrating two key improvements: (1) Prompt optimization, where translation instructions are carefully tailored for both base and instruct models to stabilize output formats and improve translation consistency. The prompts are illustrated in Appendix A. (2) Post-processing integration, which leverages FuncTransEval's robust mechanism to extract translated code, normalize function signatures, and conduct automated unit testing for functional validation. Together, these components establish Standard as a unified, reliable evaluation framework that enables consistent assessment of LLMs' true translation performance across multiple programming languages.

Using this improved evaluation pipeline, we conduct a comprehensive baseline study across nine representative LLMs spanning *Non-LLM Baselines* (TransCoder, TransCoder-ST), *Code-specialized LLMs* (CodeLlama-7B, CodeLlama-7B-Instruct, Qwen2.5-Coder-7B, and Qwen2.5-Coder-7B-Instruct), and *General-purpose LLMs* (Qwen2.5-7B-Instruct, Meta-Llama-3.1-8B, and Meta-Llama-3.1-8B-Instruct). In order to ensure a fair comparison, all models are tested under consistent settings (temperature 0, top_p 0.95) ensuring fair cross-model comparison. The results not only reveal the current state of LLM-based translation but also uncover systematic failure patterns that motivate the design of our proposed framework. We formulate a series of research questions to systematically investigate the nature of translation errors and the contextual aware gaps in direct LLM translation. These questions guide our error analysis and directly inform the architecture of ICAM-Trans.

Table 3. Comparison of Model Performance on FuncTransEval

Model	Python→C++		C++→Python		Python→Java		Java→Python		C++→Java		Java→C++	
	pass@1	ACA	pass@1	ACA	pass@1	ACA	pass@1	ACA	pass@1	ACA	pass@1	ACA
Non-LLM Baselines												
TransCoder	28.00	30.77	37.00	41.62	40.75	47.09	40.75	47.09	52.75	55.76	52.25	53.95
TransCoder-ST	36.00	40.01	56.00	60.23	51.75	55.32	68.00	72.70	56.00	58.11	58.25	61.12
Code-specialized LLMs												
CodeLlama	14.55	15.47	13.55	15.22	15.55	16.94	15.00	16.76	20.00	20.96	16.00	16.64
CodeLlama-Instruct	68.75	72.70	72.25	76.02	72.55	75.24	75.75	79.73	80.75	82.21	78.75	81.75
Qwen-Coder	**78.75**	**80.39**	**91.75**	**93.65**	83.00	85.53	90.75	93.63	**94.75**	**95.97**	79.75	81.31
Qwen-Coder-Instruct	69.25	71.87	91.00	92.68	**84.25**	**86.45**	**92.00**	**94.40**	91.67	93.78	78.75	80.71
General-purpose LLMs												
Qwen-Instruct	72.25	75.36	85.25	88.08	79.55	82.06	88.50	91.39	89.25	90.72	**84.25**	**86.64**
LLaMa3.1	53.25	57.57	73.00	77.44	61.55	64.87	74.25	78.11	76.00	78.19	74.55	75.75
LLaMa3.1-Instruct	64.75	68.01	84.50	87.36	76.75	80.67	84.50	88.38	84.25	87.84	77.50	79.70

Model	Python→JavaScript		JavaScript→Python		C++→JavaScript		JavaScript→C++		Java→JavaScript		JavaScript→Java	
	pass@1	ACA	pass@1	ACA	pass@1	ACA	pass@1	ACA	pass@1	ACA	pass@1	ACA
Code-specialized LLMs												
CodeLlama	12.25	15.96	12.50	14.22	16.50	19.08	17.55	19.33	7.25	9.00	17.55	18.48
CodeLlama-Instruct	68.50	77.07	77.75	81.21	73.00	81.59	63.75	67.00	65.00	73.13	62.75	64.78
Qwen-Coder	88.75	92.86	91.55	93.74	89.75	93.90	**73.00**	**75.77**	**92.00**	**95.52**	**81.75**	**83.27**
Qwen-Coder-Instruct	**89.75**	**92.87**	**93.00**	**94.83**	89.55	92.26	70.00	71.57	89.25	91.52	79.55	80.61
General-purpose LLMs												
Qwen-Instruct	86.00	89.88	89.75	92.53	**90.00**	**93.46**	72.75	74.79	84.25	86.98	77.55	78.91
LLaMa3.1	69.75	79.74	59.25	61.60	75.75	83.25	52.55	55.15	74.50	81.41	51.55	53.40
LLaMa3.1-Instruct	85.25	89.84	85.75	89.92	87.75	91.57	58.25	60.51	87.25	91.72	72.50	75.74

Values are represented in %. The best results achieved for each translation pair are highlighted in bold.

4.1 RQ1: How Effectively Can LLMs Perform Direct Code Translation under a Standardized Evaluation Process?

Under the standardized evaluation pipeline, LLMs demonstrate strong translation capabilities. As shown in Table 3, LLM-based models substantially outperform traditional non-LLM baselines across all language pairs. Among them, Qwen2.5-Coder-7B achieves the best overall performance, with pass@1 scores ranging from 73.00% to 94.75% and ACA scores from 75.77% to 95.97%. In contrast, CodeLlama consistently performs below 20%, which can be attributed to its lack of instruction fine-tuning—limiting its ability to interpret user prompts and execute guided translation tasks effectively. These results suggest that LLMs can perform direct function-level code translation with high fidelity and stability under a standardized evaluation framework, establishing them as the new performance baselines for future research in this area.

4.2 RQ2: How Do Instruction-Following Capabilities and Model Specialization Affect the Performance of LLMs in Function-Level Code Translation?

Our experiments show that *instruction-following capability is a key determinant of translation quality*. Models trained with instruction tuning—such as CodeLlama-Instruct and LLaMA3.1-Instruct—consistently outperform their base counterparts across all language pairs, demonstrating stronger comprehension of task intent and adherence to cross-language conventions. Interestingly, Qwen-Coder base models achieve relatively strong performance even without explicit instruction tuning, likely because their pretraining corpus already includes translation-related tasks, granting them implicit instruction-following ability. In contrast, the base CodeLlama model performs

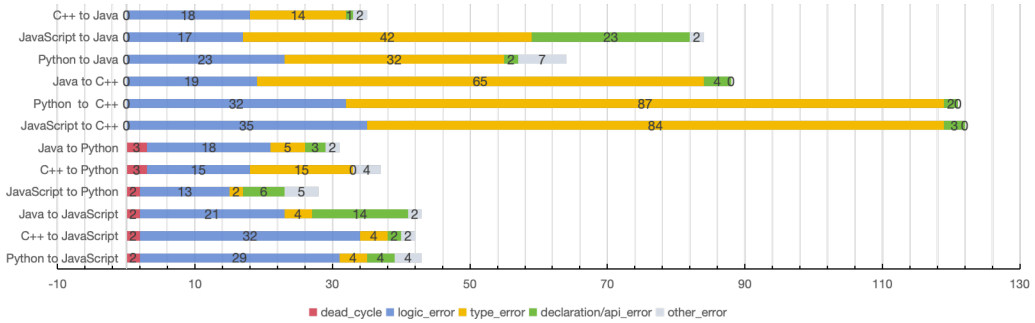


Fig. 6. Distribution of failed cases in each translation group.

substantially worse, highlighting the importance of aligning model behavior with task instructions. However, *code specialization alone does not guarantee superior performance*. For example, Qwen-Instruct (a general-purpose model) achieves comparable or even better results than Qwen-Coder in certain directions, such as Java → C++ and C++ → JavaScript. This indicates that effective function-level translation relies less on code-specific pretraining and more on general semantic reasoning and cross-lingual alignment. Overall, these findings suggest that high-quality code translation depends on both robust instruction alignment and contextual reasoning rather than narrow domain specialization.

4.3 RQ3: What Are the Dominant Error Categories in Direct LLM-Based Code Translation, and How Are They Distributed across Language Pairs?

We conducted a systematic error analysis on 738 translation failures produced by Qwen2.5-Coder-7B-Instruct over the enhanced benchmark. Figure 6 visualizes the distribution of these errors across multiple language pairs, grouped into five representative categories: Type errors, Logic errors, Declaration/API errors, Dead Cycles, and other errors. *Type Errors* is the most frequent failure mode across nearly all translation directions. These errors include incorrect data type mappings, mismatched function signatures (parameter or return types), and invalid type casting across language boundaries. It appears as a run-time error in Python and JavaScript, and as a compile-time error in C++ and Java. Their prevalence reflects the LLM's inability to enforce consistent type systems without explicit structural reasoning. This suggests that pattern-based translation alone cannot guarantee type correctness, which is inherently context-dependent. *Logic Errors* form the second largest category, where the translated code is syntactically correct but functionally incorrect. The LLM often misinterprets control flow, algorithmic operations, or conditional logic, leading to semantic drift from the source function. This indicates that LLMs lack an internal representation of execution semantics, resulting in failures to preserve functional equivalence even when surface-level syntax is correct. *Declaration/API Errors* arise from incorrect use of target-language standard libraries, system APIs, or framework-dependent declarations. The LLM frequently substitutes unavailable functions or misapplies semantically similar but incompatible constructs. Such cases reveal weak cross-language knowledge grounding, particularly when mappings between libraries or idioms are non-trivial. *Dead cycle* and *other_errors* occur infrequently but are still present. Dead cycle is particularly prone to appear in interpreted languages like Python and JavaScript, which lack compile-time checks. Such errors are critical in real-world code translation scenarios, as they can directly cause system crashes or run-time failures. Overall, logic and type errors collectively account for nearly 85% of all translation errors. *All these errors are not uniformly distributed.* Their prevalence varies significantly across translation directions, reflecting the inherent asymmetry in language

design (e.g., static vs. dynamic typing, class-based vs. script-based structures). This distribution implies that a one-size-fits-all translation strategy is insufficient; instead, translation systems must adapt their reasoning to the specific semantic and syntactic challenges posed by each language pair.

4.4 RQ4: What Insights Do Error Patterns Provide for Improving LLM-Based Code Translation?

The error distribution observed in RQ3 provides a direct empirical explanation for the limitations of standard prompting. Specifically, nearly 85% of all failures are attributed to logic and type errors. This concentration is not accidental. It indicates that the dominant failure mode of direct LLM-based translation is structural rather than incidental. These dominant error categories share a common root cause: missing implicit context.

Such implicit context includes execution semantics, type-system constraints, and ecosystem-level conventions that are not fully expressed in the surface token sequence of the source function. Instead, they reside in the language's run-time behavior, typing discipline, and library design principles. Because standard prompting treats translation as a surface-level sequence-to-sequence mapping task, it forces the model to simultaneously translate syntax and infer these unstated constraints within a single generation pass. Without explicit modeling of these constraints, semantic inconsistencies become unavoidable. This structural limitation manifests in the failure patterns illustrated in Figure 1:

— *Type Inconsistency (Unconstrained Type Inference)*: In dynamically typed languages (e.g., Python or JavaScript), the absence of explicit type declarations forces the model to guess variable and return types, leading to mismatched function signatures. This is the largest error category, indicating that type inference is the most critical implicit context missing from standard prompting.
— *Logic Inconsistency (Implicit Semantic Misinterpretation)*: The model fails to infer language-specific operational rules, such as implicit $float$-to-int truncation in C++, resulting in altered numerical behavior. It is the second largest error category, indicating that reasoning over implicit execution semantics is a critical contextual gap.
— *Declaration/API Misuse (Ecosystem Misalignment)*: Literal API translation without adapting to the target-language environment leads to nonexistent or incompatible calls (e.g., generating Java's *Math.trun*). While this category is less prevalent than logic and type errors, it still highlights the model's failure to reason about cross-language ecosystem differences, which is another form of implicit context.

The empirical provides concrete evidence that end-to-end prompting alone is insufficient. High-fidelity code translation requires explicit modeling of implicit context before generation. Rather than relying solely on holistic translation, future systems must first analyze and formalize execution semantics, type dependencies, and cross-language constraints, and then guide generation through structured reasoning. By decoupling contextual analysis from surface translation, the system reduces the risk of hallucinating hidden language rules or unstated semantic assumptions. This insight directly motivates the design of ICAM-Trans, which introduces specialized analysis agents to extract and bridge implicit contextual gaps prior to target code generation.

5 Methodology

5.1 Overview

The *I*mplicit-*C*ontext-*A*ware *M*ulti-agent framework for Function-Level Code Translation (*ICAM-Trans*) is a hierarchical and autonomous multi-agent architecture designed to explicitly uncover

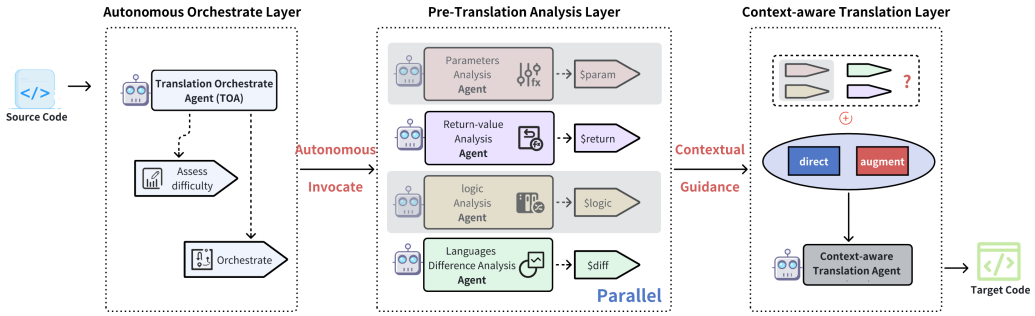


Fig. 7. The overview of ICAM-Trans.

and leverage implicit contextual cues that are often overlooked in conventional code translation. Unlike traditional end-to-end translation paradigms that rely solely on direct source-to-target mapping, ICAM-Trans introduces autonomous pre-translation analysis guided by an orchestration agent, enabling a more implicit-context aware translation process. As illustrated in Figure 7, the framework consists of three cooperative layers, each coordinated by one or multiple specialized agents:

— *Autonomous Orchestrate Layer*: The TOA dynamically evaluates the translation difficulty of each function and determines whether pre-translation analysis is necessary, as well as which specific analyses should be performed.
— *Pre-Translation Analysis Layer*: A set of specialized CAAs conduct multi-perspective analyses of the input function. The activation of these agents are guided by the TOA, enabling the system to focus on the most relevant implicit contextual aspects for each translation task.
— *Context-Aware Translation Layer*: The CTA integrates the outputs from the pre-translation analysis layer and dynamically constructs enriched prompts that incorporate the contextual insights to guide the translation process.

5.2 Autonomous Orchestrate Layer

In this layer, the *TOA* serves as the central coordinator and intelligent decision-maker within the ICAM-Trans framework. Unlike conventional fixed pipelines that proceed directly to translation, the TOA performs an autonomous task analysis on each incoming Source Code function to determine the optimal translation strategy. The TOA begins by assessing potential translation challenges, leveraging contextual clues derived from both the source function's complexity (e.g., control flow, data structures) and the known language differences between the source and target programming languages. Rather than following a fixed pipeline, the TOA dynamically decides whether and which auxiliary context analyses are necessary, selectively engaging the specialized CAAs. This approach ensures that computational resources are focused on critical aspects of the code while avoiding the introduction of unnecessary information that could interfere with model generation. The exact instruction templates for TOA are detailed in Appendix B.1. To maintain transparency, the TOA explicitly outputs its decision-making process in a structured JSON format, which includes:

— `Identified Translation Difficulties`: A summary of the potential risks or challenges detected in the source function.
— `Selected Analysis Agents`: A list of the specific CAAs (e.g., PAA, LAA) activated for the pre-translation analysis phase.

Table 4. Mapping of Identified Error Categories to CAAs

CAA	Targeted Error Category	Function Aspect	Justification and Role in ICAM-Trans
PAA	Type Inconsistency	Interface (Input)	Analyzes parameter types and semantics to prevent cross-lingual type-mapping failures.
RAA	Type Inconsistency	Interface (Output)	Analyzes expected return semantics to ensure the generated function signature strictly matches the target language rules.
LAA	Logic Inconsistency	Implementation (Algorithm)	Surfaces implicit algorithmic rules and operational nuances (e.g., implicit float-to-int truncation) to preserve strict semantic equivalence.
DAA	Declaration/API Misuse	Implementation (Ecosystem)	Identifies necessary library mappings and language-specific idioms to prevent hallucinated or incompatible API invocations.

By formalizing this orchestration, the TOA not only directs the subsequent pre-analysis layer but also provides a clear and interpretable rationale for its choices, facilitating debugging, evaluation, and future enhancements to the orchestration logic.

5.3 Pre-Translation Analysis Layer

The *CAAs* form the core of the pre-translation analyses layer of the ICAM-Trans framework. Rather than selecting agents arbitrarily, the design and selection of these four specialized agents are directly derived from the dominant error distributions identified in our empirical study (Section 4.4).

To comprehensively capture a function's implicit context and prove the sufficiency of our design, we define an isolated function by its two anatomical halves: its interface and its implementation. As summarized in Table 4, we map our specialized agents directly to the primary failure modes: Type Inconsistencies are mitigated by analyzing the interface via the PAA and RAA; Logic Inconsistencies are addressed by analyzing the core algorithmic implementation via the LAA; and Declaration/API Errors are prevented by adapting the implementation to the target ecosystem via the DAA. Because a standalone function relies solely on its interface and local implementation, these four dimensions are necessary and sufficient to capture the entirety of the pre-translation context.

The CAA module comprises the following four specialized agents. Rather than relying on grounded external analysis (e.g., deterministic static analysis tools), these agents are implemented entirely through structured LLM reasoning via specialized zero-shot prompting. The exact instruction templates for each agent are detailed in Appendix B.

— *PAA*: Performs semantic and structural inspection of function parameters. It leverages the LLM's in-context reasoning to analyze function signatures, naming conventions, and implicit typing cues. The PAA extracts the exact types (including complex nested structures such as `list[int]`) and generates a structured output to guide dynamic-to-static type mapping. The final analysis result is strictly formatted as:
`[parameter_name] - [parameter_type]: [one_sentence_description]`
(e.g., mat - list of list of int: A 2D list representing a matrix.)
— *RAA*: Utilizes targeted prompting to deduce the expected return semantics from the source code. Similar to the PAA, it captures both simple and nested return types, producing a

concise, structured summary to ensure that the generated target function signature is correctly declared. The final analysis result is strictly formatted as:
`[number] return value(s):`
`Value[index] - [type]: [one_sentence_description]`
(e.g., 1 return value(s): Value1 - string: The function concatenates the input string with its reverse.)

— *LAA*: Prompted to explicitly extract the core algorithmic steps and control-flow structures. Crucially, it highlights implicit language-specific operational nuances (such as implicit float-to-int truncation in C++ or floor division behaviors) that must be preserved. The output is a direct textual summary formatted as:
`[description]`
(e.g., The function FindMaxK calculates the largest power of 2 less than or equal to a given integer n.)

— *DAA*: Leverages the LLM's extensive pre-trained parametric knowledge. We explicitly prompt the DAA to focus only on elements that actually appear in the code across five predefined categories: (1) Syntax Differences, (2) Operator and Expression Differences, (3) Library and API Mapping, (4) Data Types and Structures, and (5) Language Specific Features. It dynamically filters out irrelevant categories and generates actionable mapping guidance formatted as:
`- [consideration_type]: [description]`
(e.g., - Operator and expression differences: JavaScript uses 'Math.trunc()' for integer division, while Java uses '(int)' casting.)

The activation of CAAs is dynamically controlled by the TOA, ensuring that only relevant agents are invoked based on the complexity and characteristics of the source function. This adaptive mechanism optimizes computational resources and focuses analysis on critical semantic dimensions. Besides, to enhance pre-translation efficiency, the selected CAAs execute concurrently, allowing multiple facets of the source function (parameters, return values, logic, and cross-language considerations) to be analyzed simultaneously. This parallelism minimizes latency in the implicity-context analysis, ensuring that the CTA receives comprehensive and timely contextual guidance to generate accurate, semantically aligned target code.

5.4 Context-Aware Translation Layer

The *CTA* constitutes the final layer of the ICAM-Trans framework and is responsible for generating the target function-level code. It is explicitly designed to leverage the Contextual Guidance produced by the Pre-translation Analysis Layer, thereby enhancing translation accuracy, semantic consistency, and language-specific compliance. The CTA employs a flexible prompt construction strategy that integrates the upstream analysis results into the translation process:

— `System Prompt Instruction`: The CTA is initialized with a comprehensive TranslatedSystem_prompt (illustrated in Appendix B.6) that establishes its role as an expert code translator and enforces critical constraints, particularly the preservation of parameter and return-value consistency between source and target code.

— `Dynamic Context Injection`: Prior to translation, the CTA checks for the presence of analysis results generated from the activated CAAs (parameters, return values, logic, and language differences). If any result is available, it constructs a rich user prompt that combines the original source code with all relevant contextual insights. This mechanism ensures that the underlying LLM has explicit information regarding type mappings, logical nuances, and cross-language considerations, directly addressing common translation errors.

—`Basic Translation Fallback`: When the TOA determines that the translation task is straightforward and no pre-translation analyses are required, the CTA executes a simplified prompt to maintain efficiency while still producing reliable code.

By consuming and acting upon the structured, agent-generated context, the CTA effectively transforms the LLM from a generic code translator into an informed, domain-specific adaptation engine. This design allows the generated target code to faithfully preserve semantics, maintain function signatures, and comply with target language conventions, significantly improving translation reliability and correctness.

6 Experimental Evaluation

6.1 Experimental Setup

All experiments were conducted on a single high-performance workstation equipped with an NVIDIA GeForce RTX 5090 D GPU (32 GB VRAM). Open source models were executed locally using vLLM with float16 precision to balance computational efficiency and numerical stability. For closed-source LLMs, we accessed models via their official APIs. All experiments were conducted on the enhanced FuncTransEval benchmark (Section 3). We selected two representative LLMs from distinct paradigms to evaluate ICAM-Trans:

—Qwen2.5-Coder-7B-Instruct: An open source, code-specialized LLM fine-tuned for multilingual programming tasks. It serves as a representative of open source LLM ecosystems, allowing us to test ICAM-Trans under fully controllable local inference conditions.
—Qwen-Flash: A close-source model with advanced reasoning and cross-language understanding capabilities, accessed via API-based inference. It represents the commercial LLM ecosystem, reflecting real-world performance constraints and deployment scenarios.

These models were chosen to provide a balanced perspective on the translation capabilities of current-generation LLMs across open source and commercial ecosystems. For all experiments, we maintained consistent hyperparameters (temperature, top-p, and decoding settings) across both models to ensure a fair comparison. Evaluation metrics include pass@1, pass@10, and ACA, providing quantitative measures of functional correctness, probabilistic translation success, and overall semantic preservation.

6.2 Quantitative Comparison with Baselines

To rigorously evaluate the effectiveness of ICAM-Trans, we compare it against four representative baseline approaches, selected to cover a spectrum of common code translation strategies:

—*Vanilla* [17]: The most basic LLM-based translation, which directly applies the raw translation prompt provided without any additional contextual guidance or prompt engineering. This baseline reflects the performance of out-of-the-box LLM translation.
—*Standard*: A prompt-enhanced baseline developed in our study (Section 4). It incorporates carefully designed instructions and minimal contextual cues to improve translation quality over Vanilla, while still remaining a direct, non-agent-based approach.
—*Oneshot (Yang et al.* [30]*)*: The single-example prompting strategy proposed by Yang et al., in which the model is provided with one source-to-target translation example to guide inference.
—*UniTrans (Yang et al.* [30]*)*: The test-case-augmented translation framework proposed by Yang et al. UniTrans extends the Oneshot strategy by augmenting each input function with automatically generated test inputs to improve translation robustness. Following the recommended configuration in the original paper, we augment each input function with five automatically

Table 5. Main Experiment of CAFM-Trans

Metric	Method	Python			C++			Java			JS			Average
		C++	Java	JS	Python	Java	JS	Python	C++	JS	Python	C++	Java	
Qwen-Flash														
pass@1	Vanilla	45.13	71.15	85.53	73.40	81.85	77.93	84.53	72.85	83.63	83.75	42.15	69.03	**72.58**
	Standard	77.30	89.65	91.35	95.57	96.05	94.88	94.58	83.98	94.00	97.08	77.58	87.15	**89.93**
	Oneshot	54.48	87.65	-	95.65	97.23	-	94.95	66.80	-	-	-	-	**82.79**
	UniTrans	61.97	46.30	-	95.02	89.30	-	93.98	84.50	-	-	-	-	**78.51**
	*ICAM**	*81.15*	*90.23*	*92.40*	*95.95*	*97.05*	*94.07*	*94.82*	*91.45*	*94.33*	*95.55*	*81.45*	*87.93*	***91.36***
pass@10	Vanilla	78.00	90.50	95.50	94.25	96.75	95.25	95.75	94.00	95.25	96.25	76.25	90.50	**91.52**
	Standard	84.00	93.00	93.25	96.50	98.25	96.75	95.75	91.50	95.75	97.75	83.00	92.00	**93.13**
	Oneshot	59.75	90.24	-	96.50	98.00	-	95.75	72.75	-	-	-	-	**85.50**
	UniTrans	77.00	63.75	-	96.50	98.25	-	96.75	95.25	-	-	-	-	**87.92**
	*ICAM**	*90.00*	*95.25*	*95.50*	*97.50*	*99.00*	*97.00*	*97.25*	*97.00*	*97.75*	*97.50*	*91.00*	*94.50*	***95.77***
ACA	Vanilla	46.90	72.77	87.95	74.51	82.93	79.40	86.25	75.31	85.37	84.82	43.59	69.94	**74.15**
	Standard	79.55	91.02	93.59	96.93	96.55	96.63	96.76	85.54	96.39	98.55	78.93	87.92	**91.53**
	Oneshot	54.94	89.63	-	97.26	98.04	-	96.87	67.48	-	-	-	-	**84.04**
	UniTrans	64.25	47.55	-	97.26	90.54	-	96.63	86.20	-	-	-	-	**80.41**
	*ICAM**	*83.18*	*91.76*	*94.45*	*97.16*	*97.86*	*95.87*	*96.89*	*93.42*	*96.13*	*97.44*	*83.27*	*89.10*	***93.04***
Qwen2.5-Coder-7B-Instruct														
pass@1	Vanilla	63.90	79.35	88.88	88.50	86.83	88.05	90.20	72.40	90.65	90.93	63.93	75.90	**81.63**
	Standard	70.50	82.58	89.13	90.15	90.28	88.65	91.05	77.28	89.30	91.83	68.63	77.15	**83.88**
	Oneshot	50.77	80.68	-	90.43	86.05	-	89.83	56.03	-	-	-	-	**75.63**
	UniTrans	26.83	19.35	-	88.25	47.55	-	68.77	37.40	-	-	-	-	**48.03**
	*ICAM**	*74.53*	*80.95*	*89.25*	*90.65*	*91.48*	*89.05*	*89.30*	*79.65*	89.33	*92.10*	*71.45*	*76.38*	***84.51***
pass@10	Vanilla	88.25	95.75	95.00	96.75	98.50	95.00	97.50	93.25	96.00	97.75	88.25	94.00	**94.67**
	Standard	87.25	92.00	93.75	96.00	97.50	94.25	95.50	92.50	94.25	96.00	86.00	91.00	**93.00**
	Oneshot	65.50	92.50	-	97.00	98.00	-	95.75	73.25	-	-	-	-	**87.00**
	UniTrans	77.75	74.00	-	96.50	91.25	-	95.75	87.50	-	-	-	-	**87.13**
	*ICAM**	*87.75*	*92.25*	*93.25*	*96.00*	*97.50*	*93.75*	*95.00*	*90.75*	*94.25*	*97.00*	*87.50*	*93.00*	***93.17***
ACA	Vanilla	65.83	82.12	92.42	90.89	89.06	91.66	93.18	74.64	93.31	93.14	66.00	78.12	**84.20**
	Standard	72.48	85.05	92.39	92.29	92.46	91.89	93.80	79.60	91.99	93.92	70.40	78.74	**86.25**
	Oneshot	52.02	83.25	-	92.57	88.22	-	92.48	57.66	-	-	-	-	**77.70**
	UniTrans	27.97	20.24	-	90.49	48.52	-	72.12	38.70	-	-	-	-	**49.67**
	*ICAM**	*77.21*	*83.61*	*92.58*	*92.97*	*93.50*	*92.50*	*92.70*	*82.19*	*92.29*	*94.01*	*73.50*	*77.82*	***87.07***

Values are represented in %. Yellow shading denotes the best performance for each metric among all compared baselines. Bold values in the average column represent the mean performance across all translation pairs for each baseline.

generated test inputs. To ensure a fair comparison focused on translation capability rather than post-hoc correction mechanisms, we disable the iterative repair module in UniTrans and evaluate only its prompt-based translation stage.

Experiments are conducted using both Qwen-Flash and Qwen2.5-Coder-7B-Instruct on the FuncTransEval benchmark. All models were evaluated under identical decoding conditions: temperature= 0.7, top_p = 0.95, and n = 10. The quantitative results are summarized in Table 5.

Overall. ICAM-Trans consistently achieves the best or near-best performance across nearly all metrics and settings, demonstrating its strong generality and ability to unlock the intrinsic translation potential of different LLMs. Notably, on the more powerful Qwen-flash, ICAM-Trans attains the highest scores on all three metrics (pass@1, pass@10, ACA). On Qwen2.5-Coder-7B-Instruct, it still leads in pass@1 and ACA, while remaining competitive in pass@10. These results indicate

that ICAM-Trans not only improves the translation accuracy but also enhances the robustness and executability of function-level cross-lingual code translation.

Under the pass@1 metric, which measures the accuracy of the top-1 generated translation, ICAM-Trans clearly outperforms all baselines on both models. On Qwen-Flash, ICAM-Trans achieves an average accuracy of 91.36%, exceeding the second-best result (Standard, 89.93%) by +1.43 percentage points, which confirms a notable gain in one-shot translation correctness. For instance, the accuracy of Python $\longrightarrow$ C++ translation sees a significant boost, reaching 81.15% from the second-best 77.30% (Standard), representing an improvement of +3.85 percentage points. On Qwen2.5-Coder-7B-Instruct, ICAM-Trans similarly delivers the best performance, reaching 84.51%, ahead of Standard (83.88%) and Vanilla (81.63%). These results demonstrate that ICAM-Trans substantially strengthens the deterministic language-to-language mapping capability of LLMs. The improvement is particularly evident in stronger base models, such as Qwen-Flash, where the implicit context-aware multi-agent fusion mechanism effectively activates latent translation reasoning, enabling more accurate and semantically grounded translation.

Under the pass@10 metric, which assesses the probabilistic robustness of translation by verifying whether any of the 10 sampled outputs is correct, ICAM-Trans again demonstrates strong performance. On Qwen-Flash, ICAM-Trans achieves 95.77%, the highest among all methods and +2.64 percentage points above the second-best Standard baseline (93.13%), indicating that its translation search space is both precise and sufficiently diverse. On Qwen2.5-Coder-7B-Instruct, although Vanilla attains the top score (94.67%), ICAM-Trans ranks closely second, showing only a marginal difference. This minor drop can be attributed to its analysis-driven reasoning mechanism, which concentrates probability mass around high-confidence candidates. Such focused generation behavior benefits stronger base models (e.g., Qwen-Flash) by ensuring more reliable and semantically grounded outputs, but may slightly reduce the diversity of candidates on weaker ones. Nevertheless, ICAM-Trans maintains consistent robustness across all translation directions and model types, remaining among the top performers under every evaluation configuration.

Under the ACA metric, which measures the proportion of passed test cases per translated function, offering a continuous and functionality-oriented evaluation of translation fidelity beyond discrete correctness assessed by pass@1. Once again, ICAM-Trans achieves the strongest performance across both models. On Qwen-Flash, ICAM-Trans attains an ACA of 93.04%, exceeding Standard (91.53%) and Oneshot (84.04%) by a substantial margin. Similarly, on Qwen2.5-Coder-7B-Instruct, it achieves 87.07%, maintaining a consistent lead of approximately +0.82 points over the next best baseline (Standard). These consistent improvements demonstrate that ICAM-Trans effectively preserves functional semantics between source and target languages. By incorporating implicit contextual reasoning before translation, the framework enables models to maintain logical coherence and type integrity, resulting in translated programs that are not only syntactically correct but also semantically equivalent and executable. This property is particularly critical for practical cross-lingual code translation systems, where correctness must extend beyond textual similarity to functional reliability.

In summary, ICAM-Trans consistently achieves superior overall performance across both models and all evaluation metrics. Although Vanilla attains a marginally higher pass@10 on the weaker Qwen2.5-Coder-7B-Instruct, this single outlier result does not undermine ICAM-Trans's overall advantage. The Vanilla baseline lacks consistent accuracy and robustness, as reflected by its notably lower scores on both pass@1 and ACA, indicating that its occasional success stems more from output randomness than true translation quality. By contrast, ICAM-Trans's implicit context-aware analysis mechanism systematically enhances the model's reasoning precision and semantic fidelity. This mechanism concentrates probability mass around high-confidence candidates—an effect that allows stronger backbone models (e.g., Qwen-Flash) to fully exploit their inherent

Table 6. Ablation Study on the Effectiveness of Each CAA

Metric	Method	Python			C++			Java			JS			Average
		C++	Java	JS	Python	Java	JS	Python	C++	JS	Python	C++	Java	
pass@1	Standard	69.75	84.00	89.25	90.75	91.25	89.50	92.25	78.00	89.25	93.00	69.50	79.00	**84.63**
	w/ PAA	73.75↑	84.50↑	91.50↑	93.00↑	91.50↑	90.00↑	92.25	81.25↑	90.00↑	91.50	73.50↑	79.75↑	**86.04**↑
	w/ RAA	71.25↑	83.00	90.50↑	92.00↑	92.75↑	89.25	92.25	81.50↑	89.25	91.75	69.50	76.25	**84.94**↑
	w/ LAA	76.00↑	84.75↑	89.50↑	88.75	91.00	89.00	90.00	79.25↑	90.00↑	90.50	74.25↑	83.75↑	**85.56**↑
	w/ DAA	75.50↑	85.50↑	89.00	89.75	92.25↑	89.25	90.50	80.50↑	90.25↑	92.25	73.00↑	83.50↑	**85.94**↑
ACA	Standard	72.37	86.20	92.52	92.48	93.48	92.16	94.63	80.31	91.32	94.83	71.07	80.27	**86.80**
	w/ PAA	75.51↑	86.71↑	93.96↑	95.09↑	93.38	92.74↑	94.57	82.95↑	92.81↑	93.34	75.18↑	80.62↑	**88.07**↑
	w/ RAA	73.81↑	85.55	93.34↑	94.06↑	94.43↑	92.27↑	94.95↑	83.65↑	92.10↑	93.38	71.39↑	77.67	**87.22**↑
	w/ LAA	78.78↑	87.87↑	92.93↑	90.92	92.91	91.87	93.17	82.08↑	93.18↑	92.85	76.51↑	85.31↑	**88.20**↑
	w/ DAA	77.61↑	87.83↑	92.40	91.49	93.33	92.87↑	92.85	82.15↑	93.89↑	93.84	74.63↑	84.70↑	**88.13**↑

Values are represented in %. Yellow shading denotes the best performance for each metric among all compared methods. Bold values in the average column represent the mean performance across all translation pairs for each method.

reasoning and translation capacity. Even when applied to relatively weaker models, ICAM-Trans maintains high robustness and semantic consistency, achieving the best overall balance between accuracy, reliability, and diversity across all experimental settings.

6.3 Ablation Study

To verify the rationality and effectiveness of the components designed in ICAM-Trans, we conducted an ablation study using Qwen2.5-Coder-7B-Instruct as the base model under a deterministic decoding setting (temperature = 0, top_p = 0.95, n = 1).

6.3.1 Effectiveness of the CAAs. We first designed ablation experiments to verify the effectiveness of each agent in CAAs. We selected **Standard**, the second-best performing method on average, as the baseline to ensure a fair and competitive comparison. Each ablation configuration activates only one analysis agent in CAAs (including PAA, RAA, LAA, and DAA, to provide pre-analysis context for the CTA. This setup allows us to systematically assess how each agent independently influences translation performance. The results, summarized in Table 6, show that integrating any single agent consistently improves average performance across all metrics compared to the Standard baseline.

Overall Effectiveness of CAAs. Across all translation pairs, each CAA demonstrates clear effectiveness. Compared to the baseline, all agents achieve higher average scores in both pass@1 and ACA, indicating consistent performance gains. Even in cases where certain agents yield marginal or no improvement, the impact is not negative, suggesting that the introduction of CAAs generally stabilizes and enhances translation quality rather than harming it.

Translation-Direction Dependency. Crucially, the empirical data reveal that no single agent is universally optimal across all translation scenarios. Instead, performance gains are strictly translation-direction dependent. For instance, the best-performing results do not cluster around one specific agent but are distributed based on the unique syntactic and semantic gaps between the source and target languages. This pair-specific variance highlights a fundamental limitation of static analysis pipelines: applying all agents uniformly is computationally inefficient and can introduce redundant noise. Therefore, this finding directly validates the necessity of an adaptive mechanism, specifically our TOA, to dynamically assess the translation context and orchestrate only the most effective subset of agents for each unique task.

6.3.2 Effectiveness of the TOA. Building directly on the key takeaway that agent effectiveness is translation-direction dependent, we conducted an ablation study on the TOA. The TOA is designed to dynamically select the most relevant subset of CAAs for each translation instance. To verify that this adaptive orchestration is not just theoretically sound but empirically necessary, we compared

Table 7. Ablation Study on the Effectiveness of TOA

Metric	Method	Python			C++			Java			JS			Average
		C++	Java	JS	Python	Java	JS	Python	C++	JS	Python	C++	Java	
pass@1 (%)	*ICAM**	76.25	83.75	89.75	91.25	91.75	89.50	90.50	80.50	90.00	91.00	74.25	78.25	85.56
	w/o TOA	74.50	82.75	89.25	91.25	91.00	90.50	91.75	81.00	90.50	91.75	74.00	79.25	85.63
ACA (%)	*ICAM**	79.24	86.34	93.38	93.59	93.66	92.42	93.83	82.44	92.53	93.77	76.59	79.56	88.11
	w/o TOA	77.69	85.43	92.47	93.19	92.80	93.81	94.54	82.83	93.89	93.63	76.50	80.69	88.12
time_cost (s)	*ICAM**	4.80	5.97	4.35	4.96	5.77	4.24	4.18	4.51	3.95	5.25	4.78	6.88	4.97
	w/o TOA	6.47	6.13	5.26	5.54	6.59	6.90	6.11	6.89	6.45	7.87	6.49	9.20	6.66
	time	+34.81%	+2.74%	+20.81%	+11.54%	+14.29%	+62.81%	+46.16%	+52.55%	+63.14%	+49.94%	+35.82%	+33.75%	+33.95%
activate_agents	*ICAM**	2.84	2.72	2.77	2.57	1.30	2.20	2.64	1.41	2.02	2.82	2.74	2.65	2.39

the full ICAM-Trans framework. In this experiment, we used the full ICAM-Trans as the baseline and compared it with a variant *w/o TOA*, where all four CAAs are always activated simultaneously, thus removing the agent selection mechanism. The results are summarized in Table 7.

We observed that disabling TOA resulted in only marginal average improvements in pass@1 and ACA, since it introduced more analysis information, much of it was redundant rather than truly beneficial. For example, for the translation direction Python⟶C++, performance even declined by 1.75%. More critically, disabling TOA caused a substantial rise in computational cost, an average increase of 33.95% and a maximum of 68.81%, since all agents were engaged indiscriminately, regardless of their relevance to the given translation pair. These results confirm that TOA effectively balances translation performance and computational efficiency by dynamically controlling the activation of analysis agents.

On average, only 2.39 CAAs were activated per translation in ICAM-Trans, with notable variation across translation directions, from as few as 1.30 for C++⟶Java to as many as 2.82 for JavaScript⟶Python. This variation indicates that the TOA module is actively utilized in practice and adapts its agent selection according to the complexity and contextual demands of different translation pairs.

6.4 Case Study

To further explain the effectiveness of the ICAM-Trans framework in overcoming limitations of direct code translation, we present a detailed case study involving four representative examples. These cases, summarized in Table 8, illustrate scenarios where direct translation methods produced erroneous code, while ICAM-Trans successfully generated functionally correct equivalents. Each case highlights specific challenges in cross-language translation, such as logic inconsistency, type mismatch and API misuse, and showcases how ICAM-Trans addresses these issues through its multi-agent analysis system.

In CASE1, the source C++ code defines a function to compute the largest power of 2 less than or equal to a given integer n. However, the direct Python translation incorrectly omits the flooring operation when calculating the logarithm (which is implicit in the C++ code), leading to a *Logic error*. By leveraging the ICAM-Trans framework, it dynamically invokes the *LAA* to analyze the function's logic and identify the missing flooring step. This analysis is provided to the **Context-Aware Translator (CAT)**, enabling it to generate the correct Python code that includes the flooring operation using $math.floor()$.

In CASE2, the source JavaScript code defines a function that concatenates a string with its reverse. However, in the direct C++ translation, the return type was incorrectly defined as *int* instead of *string*, leading to a *Type error*. By leveraging the ICAM-Trans framework, it dynamically invokes both the PAA and RAA to analyze the function's implicit parameter and return value.

Table 8. Cases That Direct Code Translation Fail, but ICAM-Trans Framework Success

CASE	Source Code	Error Code	CAAs	Correct Code
1	C++ Source Code	Error Python Code	LAA's Analysis	Correct Python Code

	int FindMaxK (int n) { int p = log2 (n) ; return pow (2 , p) ; }	def find_max_k(n):
		  p = math.log2(n)
		  return 2 ** p
			The function FindMaxK calculates the largest power of 2 less than or equal to a given integer n.
			...
				def FindMaxK(n):
				  p = math.log2(n)
				  return 2 ** math.floor(p)

2	JavaScript Source Code	Error C++ Code	RAA's Analysis	Correct C++ Code

	function evenLength (n) { let res = n ; for (let j = n . length - 1 ; j >= 0 ; – j) { res += n [j] ; } return res ; }	int evenLength(const std::string& n) {
		  int res = 0;
		  for (int j = n.length()-1; j >= 0; –j) {res += n[j] - '0';}
		  return res;
		}
			1 return value(s):
			Value1 - string: The function concatenates the input string with its reverse.
				std::string evenLength (const std::string& n) {
				  std::string res = n;
				  for (int j = n.length()-1; j >= 0; –j) {res += n[j];}
				  return res;
				}

3	Python Source Code	Error Java Code	PAA's Analysis	Correct Java Code

	def min_flips(mat, s):
	  n = len(mat)
	  m = len(mat[0])
	  count = 0
	  ...
	  return count
		public static int minFlips(
		char[][] mat, String s) {
		  int n = mat.length;
		  ...
		  return count;
		}
			mat - list of list of int: A 2D list representing a matrix.
			s - string: A string containing digits.
				public static int minFlips(
				int[][] mat, String s) {
				  int n = mat.length;
				  ...
				  return count;
				}

4	JavaScript Source Code	Error Java Code	DAA's Analysis	Correct Java Code

	function getNextGap (gap) { gap = Math.trunc (gap * 10 / 13) ; if (gap < 1) { return 1 ; } return gap ; }	public static int getNextGap(int gap) {
		  gap = Math.trunc(gap * 10 / 13);
		  if (gap<1) {return 1;}
		  return gap;
		}
			- Operator and expression differences: JavaScript uses 'Math.trunc()' for integer division, while Java uses '(int)' casting.	public static int getNextGap(int gap) {
				  gap = (int)(gap * 10 / 13);
				  if (gap<1) {return 1;}
				  return gap;
				}

The RAA identify the return type correctly, and the analysis is provided to the CAT, enabling it to generate the correct C++ code with the appropriate *string* return type.

In CASE3, the source Python code implicitly defines the parameter *mat* as a 2D list (list of lists) of integers. However, in the direct Java translation, *mat* was incorrectly defined as a 2D array of characters *char*[][], also leading to a *Type error*. By leveraging the ICAM-Trans, it also invokes both PAA and RAA to analyze the function's information. The PAA inspects the source code and successfully extracts the definition of "mat" as a 2D list of integers. With the help of the parameters' information, ICAM-Trans translates the type of "mat" correctly into Java's integer array type *int*[][].

In CASE4, the source JavaScript code utilizes the "*Math.trunc*()" function to perform integer division. However, in Java, there is no direct equivalent function. The direct translation incorrectly retained *Math.trunc*(), leading to a *Declaration/API error*. By leveraging the ICAM-Trans, it assesses the translation difficulty and decides to activate only the DAA in CAAs. The *DAA* then identifies the operator and expression differences between JavaScript and Java, specifically noting that Java uses (*int*) casting for integer division. This analysis is provided to the CTA, enabling it to generate the correct Java code using (*int*) casting instead of *Math.trunc*().

Overall, the four representative cases collectively demonstrate the practical effectiveness and interpretability of ICAM-Trans. Each case addresses a distinct failure type commonly observed in direct LLM-based translation and highlights how CAAs mitigate these issues. By dynamically

Table 9. Comparison of Average Inference Time (in Seconds) across Standard, UniTrans, and ICAM-Trans

	Standard	UniTrans	ICAM-Trans
Python → Java	1.77	15.98	5.06
Python → C++	1.48	20.68	4.28
Python → JavaScript	1.11	-	3.86
Java → Python	0.92	14.45	3.52
Java → C++	1.40	17.88	3.59
Java → JavaScript	1.13	-	3.50
C++ → Python	0.92	9.14	3.70
C++ → Java	1.79	16.67	3.92
C++ → JavaScript	1.14	-	3.78
JavaScript → Python	0.91	-	3.65
JavaScript → Java	1.84	-	5.45
JavaScript → C++	1.66	-	4.23
Overall Average	**1.34**	**15.80**	**4.05**

Bold values in the 'Overall Average' row represent the mean performance across all translation pairs for each method.

invoking only the relevant agents under the guidance of the TOA, ICAM-Trans ensures that the translation process remains both context-aware and computationally efficient. These case studies not only verify the correctness improvements observed in the quantitative evaluation but also reveal that the framework's advantage lies in its explicit reasoning capability, which transforms implicit LLM intuition into structured, verifiable translation guidance. This interpretability is critical for building trustworthy code translation systems that can be audited, debugged, and extended to more complex programming scenarios.

6.5 Computational Cost and Tradeoff Analysis

While ICAM-Trans demonstrates significant improvements in translation accuracy and semantic consistency, the multi-agent orchestration inherently introduces additional computational complexity. To provide a complete picture of this tradeoff, we quantified the average inference time (in seconds) across all 12 translation directions using Qwen2.5-Coder-7B-Instruct, comparing our framework against both the basic Standard baseline and the advanced, test-case-augmented UniTrans framework.

As shown in Table 9, the Standard baseline requires an average of 1.34 seconds per function. In contrast, ICAM-Trans requires an average of 4.05 seconds. This increase in latency is directly attributable to the pre-translation reasoning phase. The system consumes extra inference time as the TOA evaluates the difficulty, and the CAAs generate explicit textual representations of the implicit parameters, return values, and logical nuances. However, it is noteworthy that ICAM-Trans is nearly four times faster than the advanced UniTrans baseline, which requires an average of 15.80 seconds per function due to its reliance on iterative test case generation and execution. This demonstrates that our multi-agent orchestration architecture achieves a highly favorable balance: it significantly improves translation accuracy and robustness while maintaining much higher computational efficiency compared to other complex translation pipelines.

Despite the overhead compared to the basic baseline, we argue that the tradeoff is highly advantageous in practical software engineering contexts. In real-world migration tasks, the most expensive bottleneck is human labor. Manually inspecting, diagnosing, and debugging subtle semantic drifts,

such as a misunderstood cross-lingual type conversion or an implicit algorithmic truncation, can take a developer minutes to hours per function. Exchanging an additional ∼2.7 seconds of inference compute time to guarantee a highly reliable, semantically faithful, and executable translation significantly reduces the downstream human-in-the-loop debugging effort, validating the practical utility and efficiency of the ICAM-Trans framework.

7 Limitations and Threats to Validity

While our ICAM-Trans demonstrates consistent improvements across multiple baselines and language pairs, several limitations and potential threats to validity must be acknowledged.

Dataset Limitations. Although FuncTransEval significantly enhances G-TransEval by providing better semantic alignment, expanded test coverage, and automated evaluation, it only focuses on four mainstream languages (Python, C++, Java, JavaScript). This coverage may not fully represent the diversity of programming paradigms, especially for some rare or domain-specific languages (e.g., Rust, SQL). Future work should extend the benchmark to include additional languages and translation directions to further validate the generalizability of our findings.

Inherent LLM Limitations and Hallucination Risk. ICAM-Trans builds on the reasoning and generation capabilities of LLMs. Consequently, its performance is partly bounded by the underlying LLM's intrinsic ability to understand the user instructions, perform consistent reasoning, and handle long prompts. Crucially, while our multi-agent task decomposition significantly reduces the risk of model hallucination by narrowing the focus of each prompt, it fundamentally relies on structured LLM reasoning rather than grounded external analysis. Because all intermediate contextual insights are generated by the LLMs themselves, the framework inevitably inherits the same uncertainty it aims to mitigate, which means the hallucination risk is reduced, not eliminated. Improvements in model architectures or alignment methods may alter the observed results, implying that ICAM-Trans's relative advantages could vary across future model generations.

Generalization and Scalability to Class and Repository Levels. While this work deliberately focuses on function-level translation, we recognize that practical software migration typically occurs at the class, module, or repository scale (as evaluated by recent structural benchmarks like ClassEval-T and RepoTransBench). However, scaling to these higher levels fundamentally relies on high-fidelity, semantically equivalent translations at the atomic function level as a prerequisite baseline. If a model cannot reliably resolve intra-function implicit contexts (e.g., operator semantics or type coercions), these errors will compound exponentially at the repository level.

Fortunately, the insights and multi-agent architecture of ICAM-Trans are inherently scalable to address these broader structural complexities. To extend from isolated functions to class- or project-level translation, the TOA can remain the core coordinator, while the suite of CAAs can be expanded to include specialized agents for structural context, such as a *Call Graph Analysis Agent* or a *Member State Analysis Agent*. Furthermore, the dynamic orchestration mechanism currently used to prevent prompt bloat is perfectly suited to manage the sparsity of class-level dependencies—extracting and injecting only the necessary inter-method context rather than overwhelming the LLM with the entire repository. Future work will explore these hierarchical orchestration avenues to enhance ICAM-Trans's applicability to complex, real-world software modernization scenarios.

8 Conclusion

In this work, we systematically investigated the capabilities and limitations of LLMs in function-level code translation. We identified critical shortcomings in existing evaluations, including the lack of specialized benchmarks and fragile pipelines that often underestimate LLMs' true potential. To address these gaps, we constructed *FuncTransEval*, a reliable multilingual benchmark covering Python, C++, Java, and JavaScript, with executable unit tests and standardized evaluation metrics.

This benchmark provides a rigorous foundation for assessing LLM translation performance. Through detailed analysis on FuncTransEval, we revealed that while LLM-based translators outperform traditional non-LLM baselines, they still encounter frequent logic inconsistencies, type mismatches, and library/API misuse. These errors highlight a fundamental insight: effective code translation requires not only syntactic correctness but also a deep understanding of implicit semantics and cross-language contextual dependencies. Motivated by this observation, we proposed *ICAM-Trans*, an implicit-context-aware multi-agent framework that explicitly uncovers and leverages hidden contextual information during translation. By integrating pre-translation reasoning through specialized CAAs and guiding a CTA, ICAM-Trans achieves more semantically faithful and robust translations. Extensive experiments demonstrate that ICAM-Trans consistently outperforms strong baselines in execution-based metrics, and ablation studies confirm the complementary contributions of each agent in enhancing translation reliability and interpretability.

Overall, our work establishes a more trustworthy benchmark for evaluating function-level code translation and introduces a novel framework that effectively addresses semantic inconsistencies in cross-language translation. We believe that these contributions provide valuable insights for both future research in LLM-based code translation.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. arXiv:2303.08774v6. Retrieved from https://arxiv.org/abs/2303.08774v6

[2] Karan Aggarwal, Mohammad Salameh, and Abram Hindle. 2015. *Using Machine Translation for Converting Python 2 to Python 3 Code*. DOI: https://doi.org/10.7287/peerj.preprints.1459v1

[3] Paul F. Albrecht, Philip E. Garrison, Susan L. Graham, Robert H. Hyerle, Patricia Ip, and Bernd Krieg-Brückner. 1980. Source-to-source translation: Ada to pascal and pascal to ada. *ACM SIGPLAN Notices* 15, 11 (1980), 183–193. DOI: https://doi.org/10.1145/947783.948658

[4] Gerardo Canfora and Massimiliano Di Penta. 2007. New frontiers of reverse engineering. In *Future of Software Engineering (FOSE '07)*, 326–341. DOI: https://doi.org/10.1109/FOSE.2007.15

[5] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q. Feldman, et al. 2022. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. arXiv:2208.08227v4. Retrieved from https://arxiv.org/abs/2208.08227v4

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. arXiv:2107.03374v2. Retrieved from https://arxiv.org/abs/2107.03374v2

[7] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, 1536–1547. DOI: https://doi.org/10.18653/v1/2020.findings-emnlp.139

[8] Linyuan Gong, Jiayi Wang, and Alvin Cheung. 2024. ADELT: Transpilation between deep learning frameworks. arXiv:2303.03593. Retrieved from http://arxiv.org/abs/2303.03593

[9] Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2024. AgentCoder: Multi-agent-based code generation with iterative testing and optimisation. arXiv:2312.13010. Retrieved from https://arxiv.org/abs/2312.13010

[10] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. arXiv:2409.12186v3. Retrieved from https://arxiv.org/abs/2409.12186v3

[11] Mingsheng Jiao, Tingrui Yu, Xuan Li, Guanjie Qiu, Xiaodong Gu, and Beijun Shen. 2023. On the evaluation of neural code translation: Taxonomy and benchmark. arXiv:2308.08961. Retrieved from https://arxiv.org/abs/2308.08961

[12] Anup K. Kalia, Jin Xiao, Rahul Krishna, Saurabh Sinha, Maja Vukovic, and Debasish Banerjee. 2021. Mono2Micro: A practical and effective tool for decomposing monolithic java applications to microservices. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21)*. ACM, 1214–1224. DOI: https://doi.org/10.1145/3468264.3473915

[13] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. 2014. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '14)*. ACM, 173–184. DOI: https://doi.org/10.1145/2661136.2661148

[14] Mohammad Abdullah Matin Khan, M. Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2023. xCodeEval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. arXiv:2303.03004. Retrieved from https://arxiv.org/abs/2303.03004

[15] Marcos Macedo, Yuan Tian, Filipe R. Cogo, and Bram Adams. 2024. Exploring the impact of the output format on the evaluation of large language models for code translation. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, 57–68. DOI: https://doi.org/10.1145/3650105.3652301

[16] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2013. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '13)*. ACM, 651–654. DOI: https://doi.org/10.1145/2491411.2494584

[17] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ACM, 1–13. DOI: https://doi.org/10.1145/3597503.3639226

[18] Xue Pengyu, Wu Linhao, Yang Zhen, Wang Chengyi, Li Xiang, Li Jia, Jin Ruikai, Pei Yefei, Shen Zhaoyan, Lyu Xiran, et al. 2025. ClassEval-T: Evaluating large language models in class-level code translation. In *34the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '25)*. ACM, 1421–1444. DOI: https://doi.org/10.1145/3728940

[19] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. arXiv:2105.12655. Retrieved from https://arxiv.org/abs/2105.12655

[20] Ricardo Pérez-Castillo, Manuel A. Serrano, and Mario Piattini. 2021. Software modernization to embrace quantum technology. *Advances in Engineering Software* 151 (2021), 102933. DOI: https://doi.org/10.1016/j.advengsoft.2020.102933

[21] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. arXiv:2308.12950v3. Retrieved from https://arxiv.org/abs/2308.12950v3

[22] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS '20)*. Curran Associates Inc., 20601–20611. Retrieved from https://proceedings.neurips.cc/paper_files/paper/2020/file/ed23fbf18c2cd35f8c7f8de44f85c08d-Paper.pdf

[23] Baptiste Roziere, Jie M. Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2022. Leveraging automated unit tests for unsupervised code translation. arXiv:2110.06773. Retrieved from https://arxiv.org/abs/2110.06773

[24] Rajaraajeswari Settu and Pethuru Raj. 2013. Cloud application modernization and migration methodology. In *Cloud Computing: Methods and Practical Approaches*. Zaigham Mahmood (Ed.), Springer, 243–271. DOI: https://doi.org/10.1007/978-1-4471-5107-4_12

[25] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems* 36 (2023), 8634–8652. Retrieved from https://proceedings.neurips.cc/paper_files/paper/2023/file/1b44b878bb782e6954cd888628510e90-Paper-Conference.pdf

[26] Qingxiao Tao, Tingrui Yu, Xiaodong Gu, and Beijun Shen. 2024. Unraveling the potential of large language models in code translation: how far are we? arXiv:2410.09812. Retrieved from https://arxiv.org/abs/2410.09812

[27] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, et al. 2023. LLaMA: Open and efficient foundation language models. arXiv:2302.13971. Retrieved from https://arxiv.org/abs/2302.13971

[28] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv:2109.00859. Retrieved from https://arxiv.org/abs/2109.00859

[29] Yanli Wang, Yanlin Wang, Suiquan Wang, Daya Guo, Jiachi Chen, John Grundy, Xilin Liu, Yuchi Ma, Mingzhi Mao, Hongyu Zhang, et al. 2026. RepoTransBench: A real-world multilingual benchmark for repository-level code translation. *IEEE Transactions on Software Engineering* 52, 2 (2026), 675–690. DOI: https://doi.org/10.1109/TSE.2025.3645056

[30] Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024. Exploring and unleashing the power of large language models in automated code translation. *Proceedings of the ACM on Software Engineering* 1, FSE, Article 71 (2024), 1585–1608. DOI: https://doi.org/10.1145/3660778

[31] Zhiqiang Yuan, Weitong Chen, Hanlin Wang, Kai Yu, Xin Peng, and Yiling Lou. 2024. Transagent: An llm-based multi-agent system for code translation. arXiv:2409.19894. Retrieved from https://arxiv.org/abs/2409.19894