applied sciences

MDPI

Article

ODEL: An Experience-Augmented Self-Evolving Framework for Efficient Python-to-C++ Code Translation

Kaiyuan Feng [1], Furong Peng [1,*] and Jiayue Wu [2]

[1] Institute of Big Data Science and Industry, Shanxi University, Taiyuan 030006, China
[2] Chumin School, Shanxi University, Taiyuan 030006, China
* Correspondence: pengfr@sxu.edu.cn

Abstract

Automated code translation plays an important role in improving software reusability and supporting system migration, particularly in scenarios where Python implementations need to be converted into efficient C++ programs. However, existing approaches often rely heavily on large external models or static inference pipelines, which limits their ability to improve translation quality over time.To address these challenges, this paper proposes ODEL, an On-Demand Experience-enhanced Learning framework for Python-to-C++ code translation. ODEL adopts a hybrid inference architecture in which a lightweight internal model performs routine translation, while a more capable external model is selectively invoked upon verification failure to conduct error analysis and generate structured experience records. These experience records are accumulated and reused across subsequent translation phases, enabling progressive improvement through a closed-loop workflow that integrates generation, verification, consideration, and experience refinement. Experiments on the HumanEval-X benchmark demonstrate that ODEL significantly improves translation accuracy compared with competitive baselines. Specifically, the framework increases Pass@1 from 71.82% to 81.10% and Pass@10 from 74.30% to 89.02%, and exhibits a consistent performance improvement across multiple translation phases. These results indicate that experience reuse within a continuous task stream can effectively enhance automated code translation without modifying model parameters.

Keywords: Python-to-C++ code translation; experience-augmented learning; teacher–student framework; large language models (LLMs)

Check for updates

Academic Editors: Ilsun You, Hsing-Chung Chen, Won-Yong Shin and Mert Nakip

Received: 23 December 2025
Revised: 12 January 2026
Accepted: 18 January 2026
Published: 2 February 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

1. Introduction

The increasing diversity of programming environments and the demand for cross-platform compatibility have made automated code translation a critical technology for improving software reusability and accelerating project migration and integration [1]. This need arises from the complementary strengths of programming languages across different application domains. In practice, converting code between languages is often essential to meet specific performance targets, platform constraints, or system integration requirements [2]. For instance, in high-performance computing and embedded systems, Python (e.g., version 3.9) is favored for rapid prototyping and algorithm validation due to its expressive syntax and rich libraries, while C++ (compiled with GCC version 11.4.0) is preferred for deployment because of its execution efficiency and fine-grained memory control [3]. Automatically translating Python into optimized C++ thus bridges the gap

between development agility and production performance, offering significant engineering benefits [4].

Early approaches to code translation largely relied on rule-based systems, which performed conversions through manually defined syntactic mappings. Although such methods offer transparency, they suffer from several limitations: crafting comprehensive rule sets is labor-intensive, difficult to scale to diverse language constructs and edge cases, and often yields output that lacks optimizations characteristic of the target language [5]. In recent years, large language models (LLMs) have transformed the landscape by learning cross-lingual syntactic and semantic patterns from large-scale code corpora, enabling more fluent and context-aware translation [6]. However, deploying these models in real-world settings remains challenging. Large closed-source models impose practical limitations related to deployment and scalability, while lightweight open-source models often struggle with complex translation tasks due to limited reasoning capability [7].

More fundamentally, existing LLM-based translation pipelines face persistent challenges in adapting to long-term usage scenarios, while larger models can achieve strong performance, most current systems remain static; they do not systematically learn from translation errors encountered during application, leading to repeated mistakes and limited adaptability over time [8–10]. Recent work on multi-agent collaboration attempts to decompose translation tasks across specialized agents [11], but these approaches primarily focus on within-task coordination and lack mechanisms for accumulating experience across successive translation instances.

To address these challenges, this paper proposes an ODEL framework, a system-level experience-driven architecture for Python-to-C++ code translation that targets progressive performance improvement under practical deployment constraints. Unlike prompt engineering, self-reflection, or multi-agent orchestration methods that focus on improving within-task reasoning, ODEL emphasizes experience reuse across successive translation iterations within a continuous task stream. The core design of ODEL adopts a hybrid inference strategy. A lightweight internal model is responsible for routine translation, while a more capable external model is selectively invoked only when verification fails to perform error diagnosis and experience distillation. Importantly, the external model does not directly generate final code outputs; instead, it produces structured experience records that are accumulated and reused in subsequent translation phases. By integrating experience retrieval, verification-driven feedback, and experience refinement into a closed-loop workflow, ODEL enables incremental performance improvement without modifying model parameters or requiring fine-tuning. Compared with existing prompt engineering, multi-agent collaboration, and iterative refinement approaches, ODEL emphasizes cross-instance experience reuse rather than solely focusing on within-task reasoning enhancement. By decoupling experience accumulation from model parameter updates, the proposed framework offers a flexible system-level solution for improving code translation quality across successive translation phases under practical deployment constraints [12,13].

The main contributions of this work are summarized as follows:

1. An experience-driven, on-demand enhancement translation framework, ODEL, significantly outperforms existing baseline methods. By on-demand invocation of a high-performance external model for deep error analysis and structured experience distillation, the framework systematically improves the translation accuracy of the lightweight internal model. On the HumanEval-X benchmark, ODEL increases Pass@1 from 71.82% to 81.10% and Pass@10 from 74.30% to 89.02%, clearly demonstrating the effectiveness and superiority of its experience accumulation and reuse mechanism in enhancing code translation quality.

2. A sustainable experience accumulation and self-evolution mechanism enables long-term performance improvement across successive tasks. Through a multi-phase translation experiment, we demonstrate that ODEL can achieve sustained self-optimization without external intervention, across successive translation phases within a continuous task stream.
3. A teacher–student collaborative experience distillation mechanism effectively enhances both the quality of accumulated experience and the performance ceiling of the system. By introducing a high-performance external model (DeepSeek 685B) as the "teacher" for deep error diagnosis and structured experience generation, ODEL produces higher-quality and more generalizable experience units. Experiments show that, compared to using only the internal model for experience generation, the external experience mechanism further improves Pass@1 by 6.35 percentage points, underscoring the critical role of high-quality external experience in pushing the system's performance boundaries.

2. Related Work

This section reviews prior studies related to automated code translation and experience-enhanced code generation. We first summarize approaches based on foundation models, followed by system-level frameworks that improve translation performance through architectural design. Finally, we discuss multi-agent collaboration methods and analyze their limitations in long-term experience accumulation.

2.1. Methods Based on Foundation Models

Recent advances in large language models (LLMs) have significantly advanced research in code intelligence. Pioneering efforts such as CodeX (OpenAI Codex, codex-1 release, accessed on 19 January 2025 via https://openai.com/blog/openai-codex/) [14] and AlphaCode [15] demonstrated the capability of pre-trained models to perform complex code generation tasks. Subsequent open-source models, including CodeLlama (v1.0) [16] and Qwen-Coder (v1.0) [17], further improved performance across a wide range of programming benchmarks.

Static Code Translation. Notably, TransCoder [2] introduced an unsupervised framework for translating between C++, Java, and Python using masked language modeling and back-translation, without requiring parallel data. More recently, CodeT5+ [18] and StarCoder [19] leverage encoder–decoder architectures trained on massive multilingual code corpora to support diverse translation scenarios. Despite their success, these models treat translation as a fixed mapping: once deployed, they cannot incorporate new feedback or correct recurring errors without costly retraining or fine-tuning.

Prompt Engineering–Based Methods. Prompt engineering–based methods focus on designing structured prompts or in-context examples to guide LLMs toward more accurate translations. Prior studies have shown that incorporating example-based guidance or error-aware prompts can improve output correctness [20,21]. However, these methods rely heavily on manually designed prompts and lack a systematic mechanism for learning from recurring translation errors over time.

Fine-Tuning–Based Methods. Fine-tuning–based methods adapt pre-trained models to specific language pairs using parallel code corpora, while fine-tuning can significantly improve translation accuracy [22,23], it depends on large volumes of high-quality aligned data and requires retraining when adapting to new tasks or domains, which limits flexibility in dynamic development environments.

Positioning of Our Approach. In contrast to the aforementioned methods, the framework proposed in this work does not require expensive fine-tuning or extensive prompt

engineering. Instead, it functions as a lightweight, modular enhancement that can be integrated into existing translation pipelines. By equipping the base model with precise, contextually relevant experiential guidance—derived from structured error analysis and accumulated correction knowledge—our system significantly improves translation accuracy while maintaining deployment feasibility under practical constraints. This design effectively bridges the gap between static model capabilities and the dynamic, experience-driven demands of practical code translation.

2.2. Methods Based on System-Level Frameworks

Unlike methods that modify model parameters, system-level approaches aim to enhance translation performance through external architectural innovations and post-processing mechanisms. These can be classified into two principal categories, each with distinct design philosophies and operational characteristics.

Multi-Round Verification Methods. This class of methods employs iterative refinement cycles, typically driven by test execution or compiler feedback, to progressively correct translation outputs. For instance, Chen et al. [24], who proposed a compiler-guided evaluation and repair loop, and Cassano et al. [25], who developed a multilingual benchmark incorporating verification phases, while these techniques can improve functional correctness through runtime feedback, they often entail significant practical deployment constraints due to repeated execution and analysis. Moreover, they generally lack a structured mechanism to capture, generalize, and reuse correction patterns across tasks, limiting long-term efficiency gains.

External Knowledge–Enhanced Methods. Another line of research seeks to augment LLMs by retrieving relevant information from external knowledge bases or historical corpora. For instance, Parvez et al. [26] explored retrieval-augmented generation for code repair and synthesis. Recent efforts like ReACC [27] apply retrieval-augmented generation (RAG) to code completion, yet their application to cross-language translation remains limited. Crucially, most retrieval systems store raw code snippets rather than structured error-fix knowledge, making them less effective for targeted correction in translation contexts.

Positioning of Our Framework. In contrast to the above approaches, our proposed framework advances system-level design by introducing a experience generation and reuse mechanism. Rather than relying solely on post hoc verification or generic retrieval, our system triggers targeted error analysis and structured knowledge extraction only when needed, converting translation failures into reusable corrective experiences. This design not only reduces unnecessary computational overhead but also enables the system to accumulate and apply translation-specific expertise in a context-aware manner. This strategy addresses a key gap in current system-level methods: the transition from passive correction to active, experience-driven learning.

2.3. Multi-Agent Collaboration Methods

A more recent research direction explores multi-agent collaboration, where multiple LLM-based agents with specialized roles work in concert to tackle complex translation tasks. For instance, Karanjai et al. [28] proposed the UniTranslator framework, in which a central DirectorLLM coordinates concept agents, language specialists, and test generators to decompose translation into structured subtasks. Such architectures can be viewed as a systematic extension of advanced prompt engineering, effectively distributing complexity across specialized modules, while multi-agent collaboration can enhance performance through task decomposition and role specialization, its capabilities remain largely confined to within-task coordination. These systems generally lack mechanisms for accumulating knowledge across tasks or consolidating experience over time, as also observed by Athi-

waratkun et al. [29], who highlighted the limited capacity of existing frameworks for cross-task learning and experience transfer.

Recent agent-based systems like Reflexion [11] and Self-Refine [12] introduce reflective loops, but they operate within a single agent's internal memory and do not externalize knowledge for system-wide reuse. Similarly, experience-augmented program synthesis frameworks such as DreamCoder [30] store successful programs but not the diagnostic reasoning behind corrections—limiting their applicability to error-driven translation repair.

Synthesis and Research Gap. Collectively, existing work falls into three categories: (1) static translators (e.g., TransCoder [2], CodeT5+ [18]) that cannot adapt post-deployment; (2) iterative debuggers (e.g., Reflexion [11], Self-Refine [12]) that improve within-task but forget across tasks; and (3) retrieval-based systems (e.g., Parvez et al. [26]) that lack structured, actionable error-fix representations. None provide a unified mechanism for parameter-free, cross-task, and experience-grounded adaptation—precisely the gap our framework addresses.

Positioning of Our Approach. Our framework extends the multi-agent paradigm by introducing a temporal learning dimension. Rather than treating each translation task in isolation, we enable agents to accumulate, structure, and reuse corrective experiences across tasks, thereby supporting adaptation and performance evolution. This design transforms multi-agent collaboration from a static, task-specific orchestration into a dynamic, experience-enriched learning system, addressing a fundamental limitation in current collaborative translation methods.

Beyond code translation, our work also relates to broader research on continual learning and memory-augmented systems in artificial intelligence. Continual learning aims to enable models to accumulate knowledge over time while avoiding catastrophic forgetting. Prior approaches often rely on model fine-tuning or replay buffers [10,23], which can be computationally expensive or infeasible in deployment settings where model weights are frozen. In contrast, ODEL achieves incremental improvement through its experience repository—storing verified error-fix records from past tasks and reusing them to guide future code generation without updating the underlying model parameters.

Similarly, memory-augmented architectures enhance model capabilities by retrieving external knowledge during inference [13], while existing methods such as retrieval-augmented generation typically retrieve raw text or unstructured examples, ODEL's experience library stores structured, validated debugging experiences that capture not only what went wrong but also how a fix applies. This design aligns with the goal of building systems that learn from mistakes over time—a principle explored in reflective agents [11] and lifelong learning frameworks—but instantiated here in the context of cross-language code translation

3. ODEL Framework

To address the static limitations of existing code translation systems—particularly their inability to learn experience continuously from translation errors—this paper introduces a ODEL framework. It establishes a learning cycle that transforms isolated error instances into reusable experiential knowledge. By integrating a hybrid inference architecture with a systematic experience accumulation mechanism, our approach enables lightweight models to progressively improve translation accuracy in Python-to-C++ conversion tasks.

3.1. Architecture and Workflow of the Framework

The proposed framework adopts a three-module collaborative design, as illustrated in Figure 1. The ODEL is embedded throughout the translation pipeline, forming a closed-loop workflow that seamlessly integrates task analysis, code generation, verification, and

experience refinement. This architecture enables continuous interaction between inference, validation, and knowledge accumulation, ensuring that translation errors are systematically captured, analyzed, and transformed into reusable corrective guidance.Notably, experience generation and external model invocation are conditionally triggered only upon verification failure.

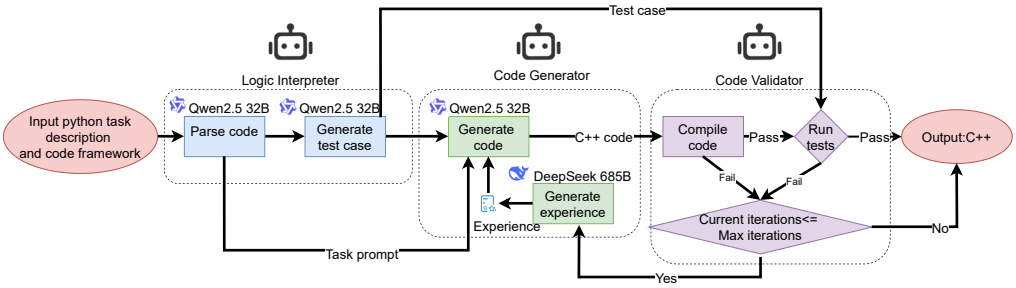


Figure 1. ODEL framework.

3.2. Logic Interpreter

As the initial stage of the translation workflow, the Logic Interpreter module performs dual parsing functions upon receiving the code task description and relevant contextual information (Figure 2). First, it transforms high-level task requirements into a structured programming logic representation, producing a detailed blueprint to guide subsequent code generation. Second, based on its functional understanding, it automatically synthesizes corresponding verification test cases (The generated test cases aim to capture core functional behaviors rather than exhaustively covering all edge cases), establishing a standardized validation baseline for downstream modules. This dual-output design ensures the system operates from a semantically grounded, verifiable foundation, thereby enhancing both the interpretability and reliability of the translation pipeline from its earliest phase.

System Role:
Autonomous code analysis agent
[STEP1]: Parse the source code ({source_lang}) and extract its abstract syntax and semantics. Include:
Program units: functions, classes, modules (name, parameters, return type)
Symbols: variables/constants (name, type, mutability, scope, initialization)
Control flow: if/for/while/return/break/continue (conditions)
Data flow: assignments, updates, dependencies
External interactions: imports, library calls
[STEP2]: Generate exactly THREE representative test cases for the code. Each test case includes:
"input": mapping of parameters to concrete values
"expected": exact expected output

Figure 2. Logic interpreter prompts.

3.3. Code Generator

The Code Generator (Code Generation and Experience Retrieval Module) operates as the central generation and learning component, employing an experience-guided generation strategy (Figure 3). Upon receiving the structured logic representation from the Logic Interpreter, it first retrieves relevant historical experiences from the experience knowledge base. Guided by this contextual knowledge, it generates an initial version of the target C++ code. If subsequent verification fails, a deep error reflection mechanism is triggered: the experience retrieval module analyzes the root cause of the failure, extracts structured corrective experience records associated with the observed failure patterns, and updates the experience repository. The module then regenerates the code, now informed by the refined experience. This process establishes a closed-loop iterative cycle of generation, verification,

reflection, and knowledge refinement, enabling continuous improvement in translation quality through systematic experience accumulation.

```
System Role:
Autonomous code translation agent.
Direct, semantics-preserving translation from {source_lang} to {target_lang}.
No fixes, optimizations, or refactoring.
Constraints:
- Output ONLY valid {target_lang} code
- No text, explanations, or comments outside code block
Rules:
- Exact given C++ signature
- Only allowed to add headers
- No extra functions/includes
Semantics:
- Strictly follow source logic
- Preserve flow, edges, flaws
Style:
- const-correct
- Idiomatic {target_lang}
- No comments
Experience-Guided Translation
The following lessons are distilled from prior translation failures.
They are provided as guidance to avoid repeating known mistakes,
but MUST NOT alter the intended semantics or structure of the source logic:
{lessons1}
{lessons2}
{lessons3}
.....
Input:
Source: {analysis}
Signature: {declaration}
Output:
c++
<translated C++ code>
```

Figure 3. Code generator prompts.

3.4. Code Validator

The Code Validator (Verification Feedback and Process Control Module) serves as the quality assurance and iterative control unit (Figure 4). It receives both the generated C++ code and the corresponding test cases, executes compilation and functional testing, and dynamically determines the subsequent workflow based on the verification outcome. If the code passes all tests, the module outputs the final translated result. If verification fails and the max iteration has not been reached, it returns detailed error information to the Code Generator to initiate a new refinement cycle. When the maximum iteration count is exceeded, the process terminates and returns the current best-effort solution. This resource-aware control mechanism ensures that the system always produces a feasible translation within bounded computational resources (Algorithm 1).

Algorithm 1 Code Validator

Require: `code` ← output from Code Generator
`test_cases` ← output from Logic Interpreter
Ensure: Validated code (success) or failure after max retries
Set `max_attempts` ← 3
Set `attempt` ← 0
while `attempt` ≤ `max_attempts` **do**
  Step 1: Perform quick syntactic checks
  if syntactic check fails **then**
    Record error information
    Send error feedback to Code Generator
    `attempt` ← `attempt` + 1
    continue
  end if
  Step 2: Invoke compiler for syntax validation
  if compilation fails **then**
    Record error information
    Send error feedback to Code Generator
    `attempt` ← `attempt` + 1
    continue
  end if
  Step 3: Construct complete executable test program
  Step 4: Write to temporary file and compile
  if compilation fails **then**
    Record error information
    Send error feedback to Code Generator
    `attempt` ← `attempt` + 1
    continue
  else
    return `code` {All steps passed}
  end if
end while
return `code` {Max retries exhausted}

System Role:
Autonomous verification agent (senior C++ QA engineer).
Validate translated C++ code with minimal executable test assertions.
Hard Constraint:
Maximum THREE (3) assertions only. One assertion = one test case.
Rules:
- Preserve original implementation verbatim
- Add tests only in main()
- Mark test section with comments (e.g., // Test code)
- Use <cassert>, <iostream>, using namespace std;
- Use standard assert() only
Output:
- ONLY one complete, valid C++ source file
- No explanations, markdown, or comments outside code
- No modifications to original code
Input:
Implementation: {target_code}
Test Cases: {test_cases}
Output:
test
// combined C++ implementation and test code

Figure 4. Code validator prompts.

3.5. Sustainable Experience Integration Mechanism

At the core of the proposed framework lies a sustainable experience-driven optimization cycle, designed to enable performance evolution through continuous knowledge application, validation, and refinement. This mechanism operates through an iterative two-phase process: (1) experience application—where retrieved knowledge guides initial code generation—and (2) experience evolution—where verification failures trigger structured knowledge extraction and experience updating. Together, these phases form a self-reinforcing learning loop that progressively enhances the system's translation capability without requiring model retraining, as illustrated in Figure 5.

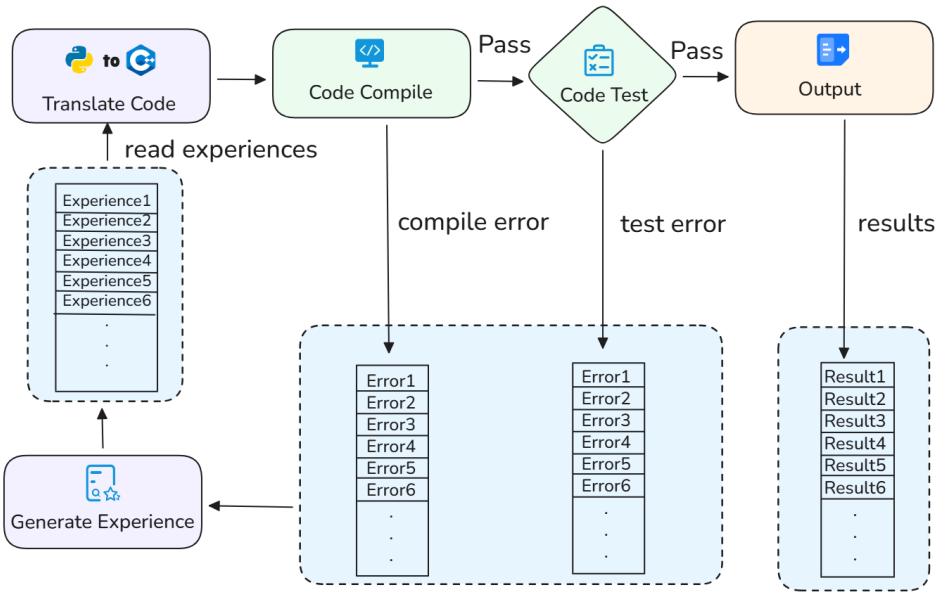


Figure 5. Experience integration mechanism diagram.

Read Experience Phase. In this stage, accumulated structured knowledge is transformed into executable logic or contextual constraints that directly inform the code generation process. This conversion allows abstract correction patterns and semantic rules to be dynamically integrated into translation, ensuring that historical experience is not only retrieved but also effectively operationalized to guide new generation tasks.

Generate Experience Phase. Translation errors serve as the catalyst for systematic knowledge base updates. When a failure occurs, the system performs in-depth contextual and causal analysis, extracting new, generalizable experience units from the error trace. This process expands and refines both the content and the relational structure of the knowledge repository, enabling the system to learn autonomously from usage and achieve sustained performance improvement over time.In this work, sustainability refers to performance improvement across successive translation phases within a continuous task stream.

3.6. Capability Alignment

To facilitate progressive performance improvement without modifying model parameters, the proposed framework incorporates a teacher–student collaboration mechanism for capability alignment. In this setting, a powerful external model (e.g., DeepSeek 685B) serves as a teacher for error diagnosis and experience distillation, while a lightweight internal model (e.g., Qwen 32B) acts as the student responsible for routine code generation, as illustrated in Figure 6.

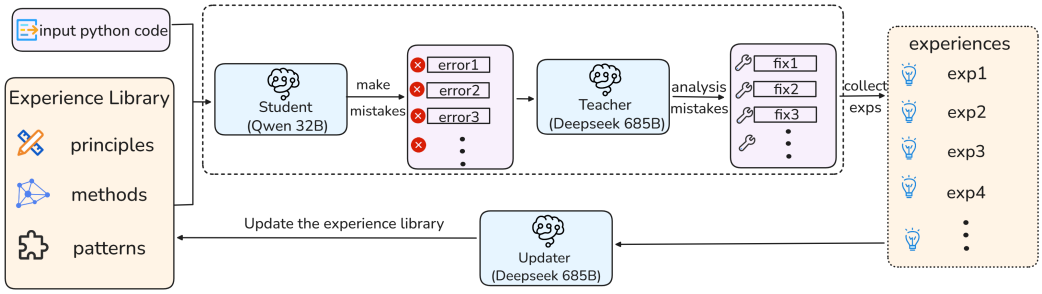


Figure 6. Model alignment diagram.

When verification fails for code generated by the internal model, the corresponding error context is submitted to the external model for diagnostic analysis. Leveraging its advanced reasoning and code understanding capabilities, the external model identifies the root causes of translation failures and distills structured corrective experience records, including error patterns, corrective strategies, and applicability conditions. These experience records are then stored in the experience repository.

In subsequent translation iterations, the internal model retrieves relevant experience records and incorporates them as contextual guidance during code generation. Through repeated exposure to and application of such structured guidance, the internal model is progressively better aligned with the target translation requirements, enabling improved generation quality without parameter updates or fine-tuning. Importantly, this alignment is achieved through experience-conditioned inference rather than direct modification of the model's internal representations.

This design allows the framework to benefit from the analytical strength of large models while maintaining the efficiency and deployability of a lightweight internal model, providing a practical pathway for sustained performance improvement under real-world constraints.

4. Experiments and Results Analysis

To evaluate the effectiveness of the proposed ODEL in Python-to-C++ code translation, we conducted a comprehensive experimental evaluation based on the HumanEval-X benchmark. Our experimental design is guided by the following research questions:

1. **RQ1 (Accuracy Improvement):** To what extent does the ODEL framework improve the functional correctness and overall accuracy of automated code translation?
2. **RQ2 (Performance Evolution Across Translation Phases):** Can the ODEL system achieve stable performance improvement over successive translation trials through its experience accumulation and reuse mechanism, thereby demonstrating self-evolution and sustained learning ability?
3. **RQ3 (Impact of External Model):** How does leveraging a more capable external model for error analysis and experience distillation contribute to further performance gains?

4.1. Experimental Setup

We evaluate our framework on the Python-to-C++ translation tasks from the HumanEval-X benchmark [29], which consists of 164 programming problems spanning a range of algorithmic and conceptual difficulties. Each task includes a natural language description, a Python reference implementation, a canonical C++ solution, and a set of functional test cases.

To ensure objective and reproducible assessment of translation quality, we adopt the unbiased Pass@k metric [14], which is widely used in code generation and translation re-

search. This metric evaluates whether a generated C++ implementation passes all provided test cases, providing a reliable measure of functional correctness.

$$\text{Pass@}k = \mathbb{E}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right] \quad (1)$$

where $n = 10$ is the number of generated candidates, c is the count of passing samples, and $k \in \{1, 10\}$.

4.2. Baselines

To systematically analyze the contribution of each component, we designed five comparative experimental configurations.

- **Qwen2.5-Coder (Single-Agent)**: Direct translation using Qwen2.5-Coder (32B) [31].
- **UniTrans Method [32]**: UniTrans is a multi-agent framework built on large language models, where multiple specialized agents cooperate through test generation, translation, execution feedback, and iterative refinement to improve automated code translation accuracy.
- **Multi-Agent (No Experience)**: The framework's main body uses Qwen2.5-Coder (32B) for translation and iteration.
- **ODEL (Internal Experience)**: The framework's main body uses Qwen2.5-Coder (32B) for translation and iteration, with its own "reflection" module responsible for error analysis and internal experience generation.
- **ODEL (External Experience)**: The framework uses Qwen2.5-Coder (32B) for translation and iteration, but calls DeepSeek 685B as an external analyzer for in-depth error diagnosis and high-quality experience generation.
- **Deepseek-Coder (Single-Agent)**: Direct translation using Deepseek-Coder (33B) [33].
- **Deepseek-Coder ODEL (External Experience)**: The framework uses Deepseek-Coder (33B) for translation and iteration, but calls DeepSeek 685B as an external analyzer for in-depth error diagnosis and high-quality experience generation.

4.3. Effectiveness of the ODEL

To address RQ1 (Does the ODEL framework significantly improve translation functional correctness?), we design a controlled experiment to validate the contribution of ODEL by comparing the performance of different learning mechanisms. The results are presented in Table 1.

Table 1. Performance comparison of different learning mechanisms on HumanEval-X.

Method	Pass@1	Pass@10
Qwen2.5-Coder (Single-Agent)	71.82%	74.30%
UniTrans Method	72.16%	76.64%
ODEL (External Experience)	81.10%	89.02%

As shown in Table 1, the ODEL framework with external experience integration achieves significant performance improvements over both the baseline and UniTrans methods. Specifically, it enhances Pass@1 by 9.28 percentage points (from 71.82% to 81.10%) and Pass@10 by 14.72 percentage points (from 74.30% to 89.02%). These results demonstrate empirically that ODEL's mechanism for systematic error analysis, experience extraction, and knowledge reuse effectively enhances translation accuracy through continuous learning adaptation.

4.4. Performance Evolution Across Translation Phases

To address RQ2 (Can the ODEL system achieve stable performance improvement over successive translation trials through its experience accumulation and reuse mechanism, thereby demonstrating self-evolution and sustained learning ability?), we designed a longitudinal experiment that simulates performance evolution in successive translation trials. This setup mimics a realistic deployment scenario, where the system processes a sequence of tasks and incrementally refines its knowledge base. Figure 7 illustrates the progression of the Pass@1 metric in five consecutive trials for the ODEL (external experience) configuration.

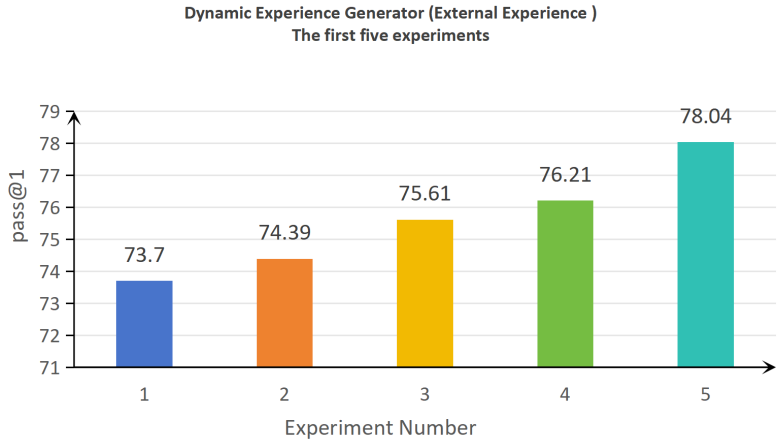


Figure 7. Pass@1 ODEL (external experience) the first five experiments.

As shown in Figure 7, the Pass@1 performance on the full test set exhibits a monotonically increasing trend as the system processes successive task phases and accumulates experience. This progression directly confirms that experience generated in earlier stages is effectively retrieved and applied to subsequent translation tasks.

The performance gain observed after the first phase largely stems from the framework's trial-and-error learning and initial experience generation. Improvements in each subsequent phase reflect the generalization and transfer of previously acquired experience to a wider variety of tasks. These results validate that ODEL not only learns within individual translation instances but also consolidates and transfers that knowledge across the task stream, enabling sustained, incremental improvement over time.

It is worth noting that the multi-phase evaluation in this work is conducted on a fixed task set from the HumanEval-X benchmark, following a consistent task ordering across phases. The observed performance improvements therefore reflect sustained learning behavior under a stable task stream rather than strict order-invariant generalization. Importantly, the experience accumulated by ODEL is stored in an abstract, pattern-based form derived from error analysis, rather than as task-specific solutions or test cases, which mitigates the risk of direct data leakage across phases.

4.5. Impact of Experience Source on Translation Performance (Ablation Study)

To address RQ3 (Does leveraging a stronger external model for error analysis and experience distillation yield additional performance gains?), we systematically examined how the source of experience generation influences overall translation quality. Within the same ODEL framework, we compared three distinct configurations: (1) no experience learning, (2) internal lightweight model for experience generation, and (3) powerful external model for experience distillation. The experimental outcomes are presented in Table 2.

Table 2. Performance comparison on the HumanEval-X dataset.

Method	Pass@1	Pass@10
Multi-Agent (No Experience)	72.56%	81.65%
ODEL (Internal Experience)	74.75%	84.70%
ODEL (External Experience)	81.10%	89.02%

Table 2 reveals several important findings:

(1) Even in the absence of experience learning, the multi-agent architecture itself yields measurable improvements over the baseline (Pass@10: 81.65% vs. 74.30%).
(2) Introducing experience learning with the internal model (Qwen 32B) further elevates Pass@10 to 84.70%, confirming the intrinsic value of the experience-driven paradigm.
(3) Most notably, when the powerful external model (DeepSeek 685B) is employed for deep error diagnosis and structured experience distillation, performance reaches its peak: Pass@1: 81.10% and Pass@10: 89.02%. This corresponds to a 6.35 percentage-point gain in Pass@1 and a 4.32 percentage-point gain in Pass@10 over the internal-model experience configuration.

These results underscore that a stronger analytical model can produce higher-quality, more generalizable experiences, effectively raising the system's upper performance bound under the evaluated setting while maintaining the efficiency of a lightweight internal model.

4.6. Deepseek-Coder (33B)

As shown in Table 3, Deepseek-Coder's Pass@1 score was 60.37%; after integrating with ODEL, Pass@1 increased to 69.09%, and Pass@10 rose even more significantly to 86.59%. This indicates that even models with relatively weaker initial capabilities can effectively enhance generation quality through external experience in ODEL, further demonstrating the framework's generalizability and effectiveness.Our model still remains among the best.

Table 3. Deepseek-Coder (33B) Performance comparison on the HumanEval-X dataset.

Method	Pass@1	Pass@10
Deepseek-Coder (Single-Agent)	60.37%	60.37%
Deepseek-Coder ODEL (External Experience)	69.09%	86.59%
Qwen2.5-Coder (Single-Agent)	71.82%	74.30%
UniTrans Method	72.16%	76.64%
ODEL (External Experience)	81.10%	89.02%

4.7. Discussion

Beyond its technical contribution, ODEL offers economic and societal benefits. By enabling accurate, deployment-efficient code translation without repeated model retraining, it can reduce the cost associated with software development. The parameter-free adaptation mechanism also lowers reliance on expensive large language models and computational resources, improving accessibility for smaller teams and promoting sustainable AI practices.

5. Conclusions

This paper introduces ODEL,an on-demand experience-enhanced learning framework designed for Python-to-C++ code translation. ODEL implements a hybrid collaborative architecture: a lightweight internal model (Qwen2.5-Coder 32B) handles routine translation tasks, while a powerful external model (DeepSeek 685B) is engaged only when verification fails, performing deep error diagnosis and distilling structured experiential

knowledge. This approach enhances the system's analytical depth without requiring continuous external model involvement. At its core, ODEL establishes a closed-loop learning cycle—integrating experience retrieval, code generation, verification feedback, and knowledge refinement—that transforms translation failures into reusable corrective experiences, enabling sustained performance evolution. Evaluated on the HumanEval-X benchmark, ODEL significantly improves functional correctness, increasing Pass@1 from 71.82% to 81.10% and Pass@10 from 74.30% to 89.02%. Ablation studies confirm the critical contribution of externally generated experience to performance gains and demonstrate the framework's ability to achieve autonomous, incremental improvement through accumulated experiential learning. In summary, ODEL advances code translation by combining selective external analysis with an internal experience-driven learning loop, fostering both high translation accuracy and long-term adaptive capability without dependence on persistent external model engagement.

Author Contributions: Conceptualization, J.W.; Methodology, K.F.; Software, J.W.; Validation, F.P.; Formal analysis, K.F.; Investigation, F.P.; Resources, F.P.; Data curation, J.W.; Writing—original draft, K.F.; Writing—review & editing, K.F.; Visualization, J.W.; Supervision, F.P.; Project administration, F.P.; Funding acquisition, F.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Key R&D Program of Shanxi Province (grant number 202402020101004), and National Natural Science Foundation of China (grant number 62276162).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Harman, M.; Jia, Y.; Zhang, Y. Search-Based Software Engineering: Trends, Techniques, and Applications. *ACM Comput. Surv.* **2012**, *45*, 11. [CrossRef]
2. Roziere, B.; Lachaux, M.-A.; Chanussot, L.; Lample, G. Unsupervised Translation of Programming Languages. In Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS '20), Vancouver, BC, Canada, 6–12 December 2020; pp. 20601–20611.
3. Nguyen, A.T.; Nguyen, T.T.; Nguyen, N.T. Lexical Statistical Machine Translation for Language Migration. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013), Saint Petersburg, Russia, 18–26 August 2013; pp. 651–654. [CrossRef]
4. Chen, X.; Liu, C.; Song, D. Tree-to-Tree Neural Networks for Program Translation. In Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS '18), Montreal, QC, Canada, 3–8 December 2018; pp. 2552–2562.
5. Karaivanov, S.; Raychev, V.; Vechev, M. Phrase-Based Statistical Translation of Programming Languages. In Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014), Portland, OR, USA, 20–24 October 2014; pp. 173–184. [CrossRef]
6. Nijkamp, E.; Pang, B.; Hayashi, H.; Tu, L.; Wang, H.; Zhou, Y.; Savarese, S.; Xiong, C. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. *arXiv* **2022**, arXiv:2203.13474. [CrossRef]
7. Zheng, Z.; Ning, K.; Wang, Y.; Zhang, J.; Zheng, D.; Ye, M.; Chen, J. A Survey of Large Language Models for Code: Evolution, Benchmarking, and Future Trends. *arXiv* **2023**, arXiv:2311.10372v1. [CrossRef]
8. Fedus, W.; Zoph, B.; Shazeer, N. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *J. Mach. Learn. Res.* **2022**, *23*, 5232–5270.
9. Ahmad, W.; Chakraborty, S.; Ray, B.; Chang, K.W. Unified Pre-training for Program Understanding and Generation. In Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Online, 6–11 June 2021; Association for Computational Linguistics: Stroudsburg, PA, USA, 2021. [CrossRef]

10. de Masson d'Autume, C.; Ruder, S.; Kong, L.; Yogatama, D. Episodic Memory in Lifelong Language Learning. In Proceedings of the 33rd International Conference on Neural Information Processing Systems, Vancouver, BC, Canada, 8–14 December 2019; pp. 13132–13141.
11. Shinn, N.; Cassano, F.; Berman, E.; Gopinath, A.; Narasimhan, K.; Yao, S. Reflexion: Language Agents with Verbal Reinforcement Learning. *arXiv* **2023**, arXiv:2303.11366. [CrossRef]
12. Madaan, A.; Tandon, N.; Gupta, P.; Hallinan, S.; Gao, L.; Wiegreffe, S.; Alon, U.; Dziri, N.; Prabhumoye, S.; Yang, Y.; et al. Self-Refine: Iterative Refinement with Self-Feedback. *arXiv* **2023**, arXiv:2303.17651. [CrossRef]
13. Lewis, P.; Perez, E.; Piktus, A.; Petroni, F.; Karpukhin, V.; Goyal, N.; Küttler, H.; Lewis, M.; Yih, W.; Rocktäschel, T.; et al. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS '20), Vancouver, BC, Canada, 6–12 December 2020; pp. 9459–9474.
14. Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; de Oliveira Pinto, H.P.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. Evaluating Large Language Models Trained on Code. *arXiv* **2021**, arXiv:2107.03374. [CrossRef]
15. Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Lago, A.D.; et al. Competition-Level Code Generation with AlphaCode. *Science* **2022**, *378*, 1092–1097. [CrossRef] [PubMed]
16. Rozière, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X.E.; Adi, Y.; Liu, J.; Sauvestre, R.; Remez, T.; et al. Code Llama: Open Foundation Models for Code. *arXiv* **2023**, arXiv:2308.12950. [CrossRef]
17. Bai, J.; Bai, S.; Chu, Y.; Cui, Z.; Dang, K.; Deng, X.; Fan, Y.; Ge, W.; Han, Y.; Huang, F.; et al. Qwen Technical Report. *arXiv* **2023**, arXiv:2309.16609. [CrossRef]
18. Wang, Y.; Le, H.; Gotmare, A.D.; Bui, N.D.Q.; Li, J.; Hoi, S.C.H. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. *arXiv* **2023**, arXiv:2305.07922. [CrossRef]
19. Li, R.; Allal, L.B.; Zi, Y.; Muennighoff, N.; Kocetkov, D.; Mou, C.; Marone, M.; Akiki, C.; Li, J.; Chim, J.; et al. StarCoder: May the Source Be with You! *arXiv* **2023**, arXiv:2305.06161. [CrossRef]
20. Austin, J.; Odena, A.; Nye, M.; Bosma, M.; Michalewski, H.; Dohan, D.; Jiang, E.; Cai, C.; Terry, M.; Le, Q.; et al. Program Synthesis with Large Language Models. *arXiv* **2021**, arXiv:2108.07732. [CrossRef]
21. Wang, D.; Li, L. Learning from Mistakes via Cooperative Study Assistant for Large Language Models. In Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP 2023), Singapore, 6–10 December 2023; Association for Computational Linguistics: Stroudsburg, PA, USA, 2023; pp. 10667–10685.
22. Zheng, Z.; Yin, P.; Lu, C.T.; Huang, X. CodeTrans: Towards Cracking the Language Barrier in Code Translation. In Proceedings of the 30th ACM International Conference on Information & Knowledge Management, Virtual Event, 1–5 November 2021; pp. 2940–2944.
23. Wang, Z.; Zhou, S.; Li, X.; Zhang, Y. Continual Learning for Code Generation. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, Lisbon, Portugal, 14–20 April 2024.
24. Chen, Z.; Kommrusch, S.J.; Monperrus, M. CERT: Continual Pre-training on Sketches for Library-Oriented Code Generation. In Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, Pittsburgh, PA, USA, 16–17 May 2022; pp. 73–84.
25. Cassano, F.; Gouwar, J.; Nguyen, D.; Nguyen, S.; Phipps-Costin, L.; Pinckney, D.; Yee, Mi.; Zi, Y.; Anderson, C.J.; Feldman, M.Q.; et al. MultiPL-E: A Scalable and Extensible Approach to Benchmarking Neural Code Generation. *arXiv* **2022**, arXiv:2208.08227. [CrossRef]
26. Parvez, M. R.; Chakraborty, S.; Ray, B.; Chang, K.-W. Retrieval Augmented Code Generation and Repair. In Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, Virtual Event, 7–11 November 2021; pp. 2119–2134.
27. Lu, S.; Xu, D.; Alon, U.; Neubig, G.; Hellendoorn, V. ReACC: Retrieval-Augmented Code Completion. In Proceedings of the IEEE/ACM 45th International Conference on Software Engineering (ICSE), Melbourne, Australia, 14–20 May 2023.
28. Karanjai, R.; Blackshear, S.; Xu, L.; Shi, W. Collaboration is all you need: LLM Assisted Safe Code Translation. In Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering, Trondheim, Norway, 23–28 June 2025; pp. 671–675. [CrossRef]
29. Athiwaratkun, B.; Gouda, S.K.; Wang, Z.; Li, X.; Tian, Y.; Tan, M.; Ahmad, W.; Wang, S.; Sun, Q.; Shang, M.; et al. Multi-lingual Evaluation of Code Generation Models. *arXiv* **2022**, arXiv:2210.14868. [CrossRef]
30. Ellis, K.; Wong, C.; Nye, M.I.; Sablé-Meyer, M.; Morales, L.; Hewitt, L.B.; Cary, L.; Solar-Lezama, A.; Tenenbaum, J.B. DreamCoder: Bootstrapping Inductive Program Synthesis with Wake-Sleep Library Learning. In Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), Online, 20–25 June 2021; pp. 835–850. [CrossRef]
31. Hui, B.; Yang, J.; Cui, Z.; Yang, J.; Liu, D.; Zhang, L.; Liu, T.; Zhang, J.; Yu, B.; Lu, K.; et al. Qwen2.5-Coder Technical Report. *arXiv* **2024**, arXiv:2409.12186. [CrossRef]

32. Yang, Z.; Liu, F.; Yu, Z.; Keung, J.W.; Li, J.; Liu, S.; Hong, Y.; Ma, X.; Jin, Z.; Li, G. Exploring and unleashing the power of large language models in automated code translation. *Proc. ACM Softw. Eng.* **2024**, *1*, 1585–1608. [CrossRef]
33. Guo, D.; Zhu, Q.; Yang, D.; Xie, Z.; Dong, K.; Zhang, W.; Chen, G.; Bi, X.; Wu, Y.; Li, Y.K.; et al. DeepSeek-Coder: When the Large Language Model Meets Programming—The Rise of Code Intelligence. *arXiv* **2024**, arXiv:2401.14196. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.