CL-ICE: A White-Box Framework for Cross-Language Code Translation Evaluation

Qi Han, Guoyu Huo, Yan Guang, Fei Kang*
Key Laboratory of Cyberspace Security, Ministry of Education, China
Information Engineering University, Zhengzhou 450001, China
qihan712@163.com, huoguoyuxi@163.com, gyinarmy@126.com, kfminnie@163.com

***Abstract*—Evaluating cross-language code translation remains a fundamental challenge due to the inherent diversity of functionally equivalent implementations. Traditional reference-based metrics fail to account for valid solutions that diverge in syntax, while execution-based evaluation is often difficult to scale because it depends on runnable environments, test cases, and external dependencies. We propose CL-ICE (Cross-Language Internal Concept Evaluation), a white-box framework that probes the internal representations of Code Large Language Models (Code LLMs) to assess translation quality without execution or reference comparison at inference time. Our approach leverages two key insights from recent interpretability research: (1) intermediate layers of Code LLMs form language-agnostic concept layers—where abstract, universal representations of algorithmic logic are encoded regardless of surface syntax, and (2) LLMs encode an internal "correctness" signal in their activation space that can be extracted through Linear Artificial Tomography (LAT). CL-ICE includes two complementary components: concept-layer profiling for semantic invariance and language-specific LAT for latent correctness probing. Experiments demonstrate that CL-ICE achieves correlation with functional correctness ($r = 0.582$) that approaches that of GPT-4-based oracle evaluation ($r = 0.615$), while maintaining deterministic operation and low computational cost.**

***Index Terms*—Large Language Models, Code Translation, Representation Engineering, Cross-Language Semantic Alignment, Language-Specific Probing**

I. Introduction

With the rapid progress of Code Large Language Models (Code LLMs), automated code translation has become increasingly important for software migration and legacy system modernization [1]–[3]. However, reliably evaluating cross-language translation quality remains an open challenge.

Existing evaluation methods are largely reference-based, typically measuring similarity between model-generated code and a human-written target solution. Metrics such as BLEU [4], CrystalBLEU [5], and CodeBLEU [6] have shown usefulness in some code-generation settings, but they remain fundamentally limited for cross-language translation. This task is inherently a one-to-many mapping: functionally equivalent target programs may differ substantially in control flow, variable naming, or API usage. As a result, valid translations can be unfairly penalized when they diverge from a single canonical reference. Embedding-based metrics [7] reduce reliance on literal matching to some extent, but often remain insensitive to subtle yet critical logical errors.

*Corresponding Author

Execution-based evaluation is generally the most reliable approach for measuring functional correctness, but it is difficult to scale because it requires runnable environments, dependencies, and test cases. LLM-as-a-Judge provides a flexible alternative, yet it introduces substantial inference cost and may suffer from non-trivial biases [8]. These limitations motivate a complementary evaluation paradigm that is less dependent on references, execution infrastructure, or black-box evaluators.

To address this problem, we propose CL-ICE (Cross-Language Internal Concept Evaluation), a white-box framework for cross-language code translation evaluation. Instead of comparing outputs to references or executing them at inference time, CL-ICE probes the internal activation space of Code LLMs to extract evaluation signals directly from latent representations. Our approach is motivated by two observations from recent interpretability research: intermediate layers of Code LLMs can encode language-agnostic representations of algorithmic logic [9], and model activations can also contain a latent correctness signal [10] that is more informative than surface similarity alone.

Based on these observations, CL-ICE combines two complementary components: concept-layer semantic alignment, which measures whether the translated code preserves source-program semantics, and language-specific latent correctness probing, which estimates whether the translation lies closer to the correct region of the target-language representation space. By integrating these signals, CL-ICE provides a structured and interpretable estimate of translation quality without requiring execution or reference comparison at inference time.

The main contributions of this paper are as follows:

- We propose CL-ICE, a white-box evaluation framework for cross-language code translation that operates without reference code or execution at inference time.
- We introduce a design that combines empirically localized concept-layer profiling with language-specific LAT probing to capture both semantic preservation and target-language correctness.
- Experiments show that CL-ICE substantially outperforms traditional automatic metrics and approaches GPT-4-based evaluation, while remaining deterministic and computationally efficient.

II. Related Works

This section reviews prior work most relevant to CL-ICE, including Code LLMs, traditional evaluation metrics for code translation, LLM-as-a-Judge, and interpretability studies on Code LLM representations.

A. Code LLMs

LLMs have substantially advanced code generation and translation. Early work such as Codex [11] demonstrated the effectiveness of large-scale code pretraining, while subsequent open models including StarCoder [12], CodeLlama [13], DeepSeek-Coder [14], and Qwen2.5-Coder [15] further improved multilingual coding ability. Despite these advances, reliably evaluating whether generated translations preserve source-program semantics remains a major challenge. In this work, Code LLMs serve both as translation models and as the representational backbone of our evaluator.

B. Traditional Code Evaluation Metrics and Limitations

Automatic evaluation of code has largely followed a reference-based paradigm. BLEU [4] and CrystalBLEU [5] measure lexical overlap, CodeBLEU [6] incorporates structural signals such as AST and data-flow information, and CodeBERTScore [7] uses contextual embeddings to capture semantic similarity. While effective in some settings, these metrics remain tied to reference solutions and are therefore limited for cross-language translation, where functionally equivalent implementations may differ substantially in structure or style. In contrast, CL-ICE evaluates consistency between source intent and translated code through internal representations, reducing reliance on a single canonical target.

C. LLM-as-a-Judge and Biases

Using strong LLMs as evaluators has emerged as an alternative to traditional automatic metrics. Prior work shows that LLM-based judges can align well with human preferences [16], and this paradigm has been extended to code evaluation through systems and benchmarks such as GPT-4-based judging [17], CODEJUDGE [18], and CODEJUDGE-BENCH [19]. However, recent evidence suggests that LLM judges can suffer from substantial bias, including preferences for superficial complexity or authority cues, while also incurring high inference cost [8]. CL-ICE instead adopts a white-box and deterministic evaluation strategy based on internal model representations.

D. Interpretability and Internal Representations of Code LLMs

CL-ICE is primarily motivated by recent interpretability work on Code LLMs. Yin et al. [9] show that intermediate layers can encode language-agnostic representations of algorithmic logic, providing a basis for measuring semantic alignment beyond surface syntax. Complementarily, Ribeiro et al. [10], building on Representation Engineering (RepE) [20], show that model activations contain correctness-related signals that can be extracted from internal representations. These findings suggest that Code LLM hidden states encode evaluation-relevant information about both semantics and correctness. Our framework builds directly on this line of work by combining concept-layer semantic alignment with latent correctness probing for cross-language translation evaluation.

III. METHODOLOGY

As shown in Figure 1, we propose CL-ICE, a white-box assessment framework that estimates code translation quality by probing the latent manifold of Code LLMs. The resulting score is intended as a correlation-based assessment signal rather than a direct replacement for execution-based verification.

The framework comprises two stages: (1) extracting maximally aligned semantic representations via empirically localized concept-layer profiling, and (2) quantifying functional validity using language-specific LAT.

A. Empirically Localized Concept-layer Profiling

1) Understanding Concept Layers: Prior work has established that Code LLMs exhibit hierarchical internal organization, with different layers encoding distinct linguistic and functional properties. Yin et al. [9] identify concept layers as regions encoding abstract algorithmic logic. These layers represent language-agnostic representations, in which universal and logic-centered abstractions of functionally equivalent code are formed independently of surface syntax, while the decodability of language-specific syntactic features is simultaneously minimized.

2) Empirical Localization Protocol for Qwen2.5-Coder-32B: Rather than assuming or directly transferring layer indices from prior studies, we ground our analysis in the empirical localization protocol proposed by Yin et al. Concept layers are characterized through their behavior under two complementary diagnostic probes:

Semantic Invariance Probe: controlled transformations that preserve semantics: (a) intra-language perturbations (identifier renaming), and (b) cross-language parallel implementations. For each layer l, we compute the average cosine similarity between perturbed pairs, yielding a semantic invariance score $I_{\text{sem}}^{(l)}$.

Syntactic Decodability Probe: We train linear classifiers to predict AST node types from layer-wise representations. Layers where syntactic information is least recoverable (low probe accuracy $A_{\text{syn}}^{(l)}$) indicate abstraction beyond explicit grammar.

Concept layers are identified as the contiguous region $\mathcal{L}_{\text{concept}}$ satisfying:

$$\mathcal{L}_{\text{concept}} = \arg \max_{\mathcal{L} \subset \{1,\dots,L\}} \left(\frac{1}{|\mathcal{L}|} \sum_{l \in \mathcal{L}} I_{\text{sem}}^{(l)} \right) \times \left(1 - \frac{1}{|\mathcal{L}|} \sum_{l \in \mathcal{L}} A_{\text{syn}}^{(l)} \right) \tag{1}$$

For our backbone Qwen2.5-Coder-32B, we conducted this localization on 500 problem-solution pairs from HumanEval covering Python, Java, C++, and JavaScript, revealing that layers 31–43 constitute the optimal concept-layer region—exhibiting peak semantic invariance ($I_{\text{sem}} > 0.82$) while minimizing AST decodability ($A_{\text{syn}} < 0.35$).

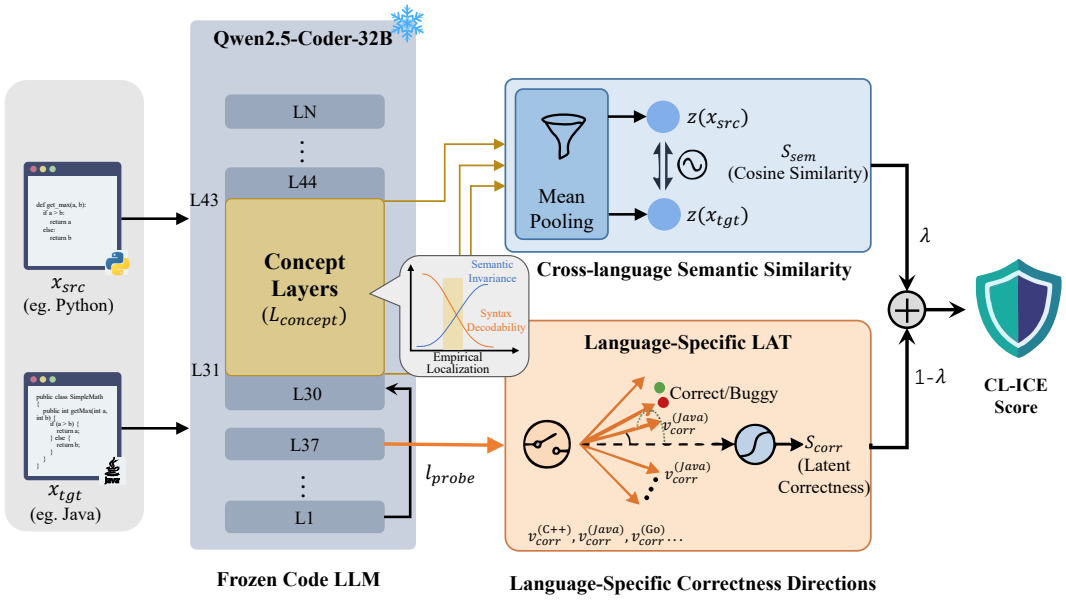

Fig. 1. Overview of the CL-ICE framework. The methodology evaluates code translation quality through two complementary internal signals: (1) Semantic Alignment (S_{sem}) derived from language-agnostic concept layers to ensure algorithmic intent preservation, and (2) Latent Correctness (S_{corr}) obtained via language-specific LAT probes to assess functional validity in the target language.

3) Semantic Alignment Extraction: Given a source program x_{src} and its translated counterpart x_{tgt}, we measure semantic preservation by operating exclusively on representations from the identified concept-layer region $\mathcal{L}_{\text{concept}}$. Let M denote our Code LLM with layer-wise hidden states $h^{(l)}(x)$. For each layer $l \in \mathcal{L}_{\text{concept}}$, we apply mean pooling over tokens to obtain:

$$v^{(l)}(x) = \frac{1}{|x|} \sum_{t=1}^{|x|} h_t^{(l)}(x) \tag{2}$$

These layer-level vectors are aggregated to form a concept-level embedding:

$$z(x) = \frac{1}{|\mathcal{L}_{\text{concept}}|} \sum_{l \in \mathcal{L}_{\text{concept}}} v^{(l)}(x) \tag{3}$$

The semantic alignment is then defined as:

$$S_{\text{sem}}(x_{\text{src}}, x_{\text{tgt}}) = \frac{z(x_{\text{src}}) \cdot z(x_{\text{tgt}})}{\|z(x_{\text{src}})\| \, \|z(x_{\text{tgt}})\|} \tag{4}$$

By restricting comparison to empirically validated concept layers, this score emphasizes preservation of algorithmic intent while reducing sensitivity to language-specific syntax.

B. Language-Specific Latent Correctness via LAT

Semantic alignment alone is insufficient for translation evaluation, as code may preserve high-level intent while introducing subtle functional errors that are often best revealed through execution. To address this at evaluation time without requiring test cases or runtime execution, we employ LAT from RepE to extract a model-internal correctness-related signal.

1) Motivation for Language-Specific Probing: In cross-language code translation evaluation, the target language plays a critical role in determining translation quality. Each programming language possesses unique syntactic structures, semantic conventions, and paradigmatic characteristics that influence how correctness manifests in the model's internal representations. For instance, correctness patterns in statically-typed languages like C++ (requiring explicit type declarations and memory management) differ fundamentally from those in dynamically-typed scripting languages like Python (emphasizing runtime flexibility and duck typing).

Training a universal correctness probe across all languages may yield a direction vector that captures only generic correctness patterns while missing language-specific nuances. Furthermore, a universal probe conflates language-specific error distributions, potentially encoding spurious correlations between syntactic surface form and execution status rather than semantic correctness. To ensure robust and precise correctness detection for each target language, we adopt a language-specific training strategy that tailors LAT probes to the unique characteristics of individual programming languages.

2) Language-Specific Direction Extraction: For each target language $L \in \{\text{C++}, \text{Java}, \text{JavaScript}, \text{Go}\}$ in our evaluation scope, we construct a dedicated training dataset and extract a language-specific correctness direction:

Language-Specific Contrastive Dataset: For each target language L, we construct paired datasets $\{(x_{\text{corr}}^{(L)}, x_{\text{bug}}^{(L)})_i\}_{i=1}^{N_L}$ containing correct and buggy implementations in that language. Correctness labels are determined via test execution during probe training only.

Language-Specific Direction Vectors: For each target lan-

guage L, we extract layer-l_{probe} representations and compute difference vectors:

$$\delta_i^{(L)} = h^{(l_{\text{probe}})}(x_{\text{corr}}^{(L)})_i - h^{(l_{\text{probe}})}(x_{\text{bug}}^{(L)})_i \tag{5}$$

Principal Component Analysis (PCA) [21] on centered $\{\delta_i^{(L)}\}$ yields the first principal component as the language-specific correctness direction $v_{\text{corr}}^{(L)}$. This direction captures the dominant pattern distinguishing correct from incorrect code in the target language L's representation space.

Benefits of Language-Specific Probing: By training separate probes for each target language, we ensure that:

- Each probe captures the unique correctness patterns specific to its target language's syntactic and semantic conventions.
- The evaluation framework adapts to language-specific characteristics without requiring complex post-hoc normalization.
- Correctness detection remains robust even when translating to languages with fundamentally different paradigms.

3) Correctness Score Computation: For a translated snippet x_{tgt} in target language L_{tgt}, the latent correctness score is computed using the corresponding language-specific direction:

$$S_{\text{corr}}(x_{\text{tgt}}) = \sigma\left(\alpha \cdot \left(h^{(l_{\text{probe}})}(x_{\text{tgt}}) \cdot v_{\text{corr}}^{(L_{\text{tgt}})}\right)\right) \tag{6}$$

where σ is the sigmoid function and α is a scaling factor. This score reflects the extent to which x_{tgt}'s representation aligns with activation patterns associated with correct implementations in its target language.

Importantly, S_{corr} is not a guarantee of functional correctness but a relative plausibility signal. It captures whether a translation resides closer to the correct region of the representation space, as learned from the training distribution of that specific target language. By computing language-specific correctness directions, this component naturally addresses the diversity of programming languages while maintaining a simple and interpretable formulation.

C. Final Score Aggregation

The overall CL-ICE score combines semantic alignment with latent correctness:

$$\text{CL-ICE}(x_{\text{src}}, x_{\text{tgt}}) = \lambda \cdot S_{\text{sem}}(x_{\text{src}}, x_{\text{tgt}}) + (1-\lambda) \cdot S_{\text{corr}}(x_{\text{tgt}}) \tag{7}$$

where $\lambda \in [0, 1]$ balances the two components. This formulation ensures that even semantically aligned translations are penalized if they exhibit low latent correctness (e.g., off-by-one errors), while also preventing structurally dissimilar yet functionally valid translations from being unfairly downgraded.

This results in a clean, interpretable evaluation metric that naturally accounts for both semantic intent and functional validity.

IV. Experimental Setup

To empirically validate the efficacy of the proposed CL-ICE framework, we designed a comprehensive experimental study aimed at addressing six primary research questions (RQs):

RQ1 (Correlation with Functional Correctness): How effectively does CL-ICE correlate with ground-truth functional correctness compared to existing surface-level, structural, and generative evaluation metrics?

RQ2 (Ablation Study): What are the individual contributions of the semantic alignment and latent correctness components to overall performance?

RQ3 (Bias Analysis): Is CL-ICE susceptible to common LLM-judge biases, such as favoring longer code or being misled by irrelevant variable renaming?

RQ4 (Generalizability across Backbone Models): Is the CL-ICE framework effective across different Code LLM architectures and parameter scales, or is it specific to the Qwen backbone?

RQ5 (Efficiency Analysis): How does CL-ICE compare to LLM-based judges in computational overhead and inference cost?

RQ6 (Hyperparameter Sensitivity): How sensitive is CL-ICE to hyperparameter choices, and are the selected settings stable across different language pairs?

A. Dataset Construction

We use HumanEval-X [14] as the main benchmark, which contains 164 programming problems with parallel solutions and test cases across Python, C++, Java, JavaScript, and Go. Python is treated as the source language, and the other four languages are used as translation targets. We use CodeLlama-7B-Instruct [13] to generate 10 candidate translations for each source-target pair, yielding 6,560 translation instances in total. Ground-truth labels are obtained by executing the generated candidates against the provided unit tests, where passing all tests is labeled as correct and failing any test as incorrect.

B. Baselines

We benchmarked CL-ICE against three distinct categories of evaluation metrics:

- N-gram Metrics: BLEU-4 [4] and CrystalBLEU [5], specialized for code.
- Structural and Semantic Metrics: CodeBLEU [6] (AST + data-flow matching) and CodeBERTScore [7] (embedding-based similarity).
- Generative Evaluation: GPT-4 as an LLM-as-a-Judge, scoring translations 0–100.

For reference-based metrics, we use the canonical target-language solutions provided by HumanEval-X as references.

C. Implementation Details

The CL-ICE framework was implemented using Qwen2.5-Coder-32B [15] as the backbone model. We utilize only the base versions of the model, avoiding confounding effects from post-training alignment techniques such as RLHF. All experiments were conducted on a workstation equipped with

8 NVIDIA A800 (80GB) GPUs using PyTorch [22] (v2.3.0) and HuggingFace Transformers [23] (v4.43.0).

Following the empirical protocol described in Section III-A and adopted by Yin et al. [9], we localize the concept-layer region on Qwen2.5-Coder-32B using HumanEval-X and identify layers 31–43 as the final $\mathcal{L}_{\text{concept}}$.

Language-specific correctness directions are trained using Project CodeNet [24]. For each target language $L \in \{C++, Java, JavaScript, Go\}$, we leveraged the dataset's submission metadata to construct contrastive pairs $(x_{corr}^{(L)}, x_{bug}^{(L)})$. Specifically, we paired submissions marked as "Accepted" (functionally correct) with those marked as "Wrong Answer" (logically incorrect) for the same problem. This approach helps ensure that the training data better reflects real-world coding distributions rather than synthetic translation artifacts. We curated a balanced training set containing 500 distinct correct-incorrect pairs for each language. Layer 37 was selected as l_{probe} based on a systematic separability analysis across layers 31–43: for each candidate layer, we measured the linear separability of correct and incorrect representations using Fisher's discriminant ratio on a 20% held-out split of the CodeNet training pairs. Layer 37 consistently achieved the highest mean discriminant ratio across all four target languages (C++: 2.41, Java: 2.18, JavaScript: 2.35, Go: 2.09), justifying its selection as the probe layer.

We report 95% confidence interval (CI) using 1,000-sample bootstrap and use paired significance tests where appropriate, with two-sided $p < 0.05$. Hyperparameters are tuned on a held-out validation set, yielding $\lambda = 0.6$ and $\alpha = 10$.

V. Results and Analysis

A. Correlation with Functional Correctness (RQ1)

As shown in Figure 2, CL-ICE achieves substantial correlation ($r = 0.582$), consistently outperforming traditional reference-based metrics across Pearson, Spearman, and Kendall measures. Compared with the strongest automatic baseline, CodeBERTScore, the improvement ($\Delta r = +0.130$) is statistically significant ($p < 0.01$).

In contrast, although GPT-4 judge yields slightly higher point estimates, the gap is small ($\Delta r = -0.033$) and not statistically significant ($p > 0.05$), indicating comparable performance.

Traditional n-gram metrics (BLEU-4, CrystalBLEU) exhibited the weakest correlation. This is primarily attributable to the inherent limitation of the single-reference evaluation paradigm. Code translation is a one-to-many problem where a source program can map to multiple functionally equivalent target implementations. Since these metrics measure similarity against a single canonical reference, they fail to recognize valid translations that diverge in implementation details or coding style. CodeBLEU improved via syntactic structure but remains constrained by this bottleneck; it penalizes correct translations that employ valid but differing control flows (e.g., recursion vs. iteration) simply because their ASTs do not align with the specific reference provided.

This supports our hypothesis that probing the internal manifold can extract richer evaluation-relevant signals than surface embeddings alone. Notably, CL-ICE operates deterministically yet remains competitive with the stochastic GPT-4 judge, suggesting that internal activations contain useful evaluation signals that can complement external judging schemes.

As shown in Figure 3, CL-ICE consistently achieves the highest Pearson correlation across all four language pairs, ranging from $r = 0.554$ to $r = 0.621$. Compared with the strongest baseline in each setting, these improvements are statistically significant under paired bootstrap testing ($p < 0.05$ for all four language pairs). This result indicates that the advantage of CL-ICE is stable rather than being driven by a single favorable target language.

By contrast, reference-based metrics exhibit substantially greater variation across targets. For example, although CodeBLEU attains moderate correlation on Java ($r = 0.428$), its performance degrades on C++ ($r = 0.297$) and Go ($r = 0.341$), suggesting that surface-level structural matching is sensitive to language-specific syntactic divergence rather than consistently capturing algorithmic correctness.

The consistently strong and statistically reliable performance of CL-ICE across diverse target languages supports the effectiveness of our language-specific LAT design. By tailoring correctness probes to the characteristics of each target language, CL-ICE can capture correctness patterns more robustly without requiring complex cross-language normalization mechanisms.

B. Ablation Study (RQ2)

We quantified each module's contribution by sequentially removing components. Table I presents degradation in Kendall-Tau correlation for each variant.

TABLE I
Ablation study on CL-ICE measured by Kendall's τ.

Method	τ	Δ
CL-ICE (full)	**0.465**	–
w/o Latent Correctness (S_{corr})	0.336	-0.129
w/o Semantic Alignment (S_{sem})	0.389	-0.076
w/o Language-Specific Probing	0.401	-0.064

The most substantial performance decline occurred when eliminating the Latent Correctness component (-0.129). This result suggests that semantic similarity measures alone are insufficient for identifying subtle functional defects. The S_{corr} term effectively serves as a proxy for execution-sensitive correctness signals that remain undetectable through semantic embeddings alone.

Furthermore, the exclusion of language-specific probing resulted in a performance drop of 0.064. This finding underscores the necessity of adapting correctness detection to the specific constraints of the target language. A generic correctness probe trained on a mixed-language corpus often fails to encode the nuanced correctness patterns inherent to different programming paradigms, thereby reducing evaluation

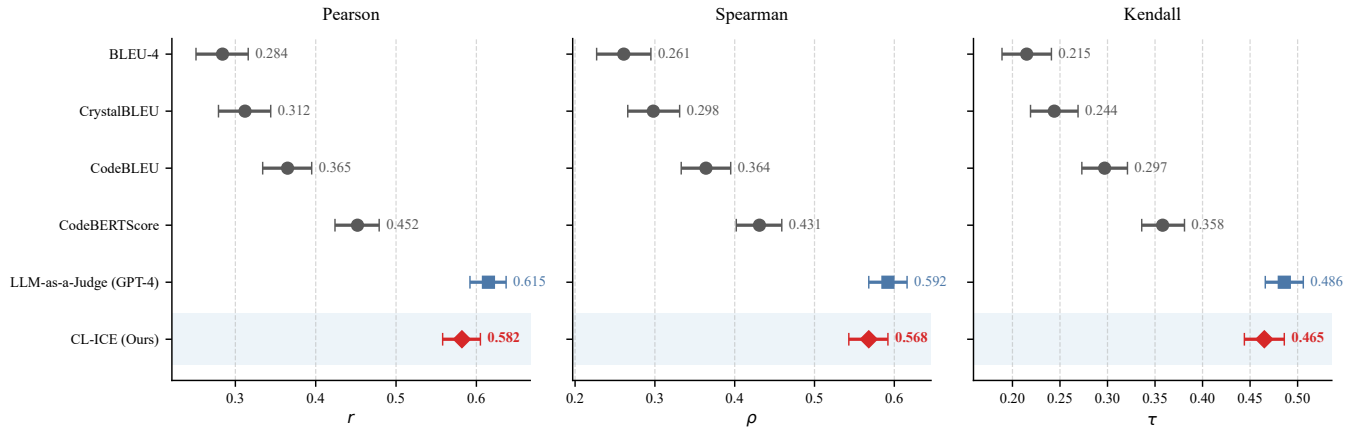

Fig. 2. Correlation with functional correctness across evaluation metrics. CL-ICE achieves strong correlation across Pearson, Spearman, and Kendall measures, outperforming traditional reference-based metrics and approaching GPT-4-based evaluation.

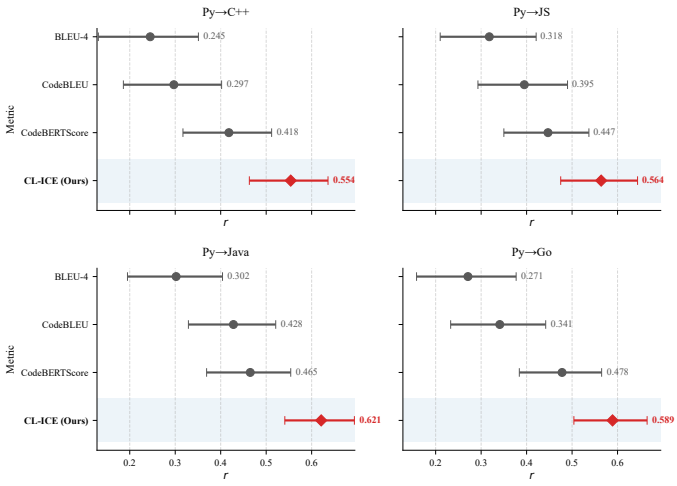

Fig. 3. Per-language performance comparison across different evaluation metrics. CL-ICE consistently achieves the highest Pearson correlation across all four translation directions, demonstrating robust cross-language generalization.

accuracy for languages with distinct syntactic and semantic rules.

Collectively, the ablation results demonstrate that semantic alignment and language-specific latent correctness are both essential and complementary components of the framework. Semantic alignment ensures the preservation of algorithmic intent, whereas language-specific LAT provides an assessment of functional validity that inherently respects the unique characteristics of each target language.

C. Bias Analysis (RQ3)

To assess whether CL-ICE inherits biases from its LLM backbone, we constructed a diagnostic set of 200 translation pairs with deliberate perturbations: variable renaming, artificial lengthening, and misleading comments.

TABLE II
BIAS SUSCEPTIBILITY ANALYSIS MEASURED BY CORRELATION DROP (%). LOWER IS BETTER.

Bias Type	LLM Judge	CL-ICE
Variable Renaming	-15.7%	**-2.3%**
Artificial Lengthening	-8.4%	**-1.8%**
Misleading Comments	-12.1%	**-3.6%**
Average Drop	-12.1%	**-2.6%**

Table II demonstrates that CL-ICE exhibits minimal susceptibility. Variable renaming caused only 2.3% correlation drop (vs. GPT-4's 15.7%), suggesting concept-layer representations effectively abstract over lexical forms. Artificial lengthening yielded negligible degradation (1.8%), whereas GPT-4 incorrectly ranked longer incorrect code higher in 34% of cases. This robustness stems from LAT's orthogonality to stylistic features.

D. Generalizability across Backbone Models (RQ4)

To address the concern that the efficacy of CL-ICE might be an artifact of the specific architecture of Qwen2.5-Coder-32B, we extended our evaluation to two additional Code LLMs with distinct architectures and parameter scales: DeepSeek-Coder-33B [14] and CodeLlama-13B.

For each model, we replicated the full pipeline: (1) empirically localizing their specific concept-layer regions ($L_{concept}$) using the protocol described in Section III-A, and (2) training language-specific LAT probes ($v_{corr}^{(L)}$) as detailed in Section III-B. We then evaluated their performance on the data constructed in Section IV-A.

As shown in Figure 4, the CL-ICE framework demonstrates robust transferability across different backbones:

- Universality of Concept Layers: All tested models exhibited distinct language-agnostic regions. While the specific layer indices shifted (e.g., deeper in Qwen/DeepSeek than

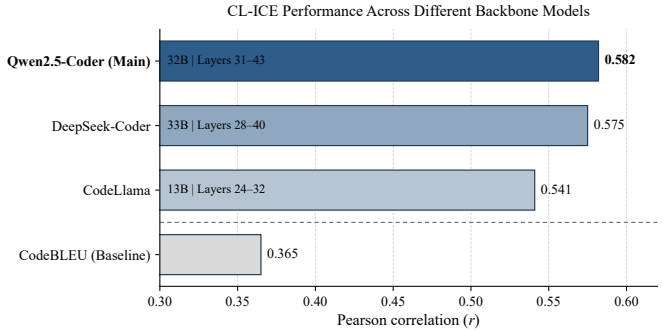

Fig. 4. CL-ICE performance across different backbone Code LLMs. The framework shows strong transferability, achieving consistently high correlation with functional correctness across Qwen2.5-Coder, DeepSeek-Coder, and CodeLlama, and significantly outperforming traditional baselines.

in CodeLlama), the phenomenon of emerging language-agnostic representations remains consistent.

- Performance Scaling: We observe a positive correlation between the backbone model's capabilities and the evaluation accuracy. DeepSeek-Coder-33B achieves performance nearly identical to Qwen2.5 ($r = 0.575$), while the smaller CodeLlama-13B yields a lower but still competitive correlation ($r = 0.541$). Importantly, even with a smaller backbone, CL-ICE significantly outperforms the best traditional metric, CodeBLEU (+0.176).

This ablation suggests that CL-ICE is a model-agnostic framework. The internal signals for semantic alignment and latent correctness are intrinsic properties of well-trained Code LLMs, suggesting that the evaluator's performance may improve as future backbone models become more capable.

E. Efficiency Analysis (RQ5)

CL-ICE requires only a single forward pass through Qwen2.5-Coder-32B (180 ms/sample on NVIDIA A800), achieving 83× speedup over GPT-4-based evaluation (avg. 15 s/sample). The deterministic nature also eliminates variance from sampling, ensuring reproducible evaluation.

The computational breakdown shows that concept-layer embedding extraction accounts for 120 ms, while LAT projection requires only 60 ms, making CL-ICE practical for large-scale deployment. Furthermore, the language-specific correctness probes can be pre-computed and cached, enabling amortization of training costs across multiple evaluation sessions.

F. Hyperparameter Sensitivity (RQ6)

We swept $\lambda \in [0.0, 1.0]$ (step 0.1) and $\alpha \in \{1, 2, 5, 8, 10, 12, 15, 20, 30\}$ independently, with all other settings fixed. The sensitivity results are shown separately in Figure 5. We report Pearson r averaged over the four language pairs; shaded bands denote the cross-language standard deviation, indicating per-language stability.

The performance curves exhibit flat plateaus around the selected values: r varies by at most $\Delta r = 0.016$ within $\lambda \in [0.5, 0.7]$ and by $\Delta r = 0.006$ within $\alpha \in [8, 12]$.

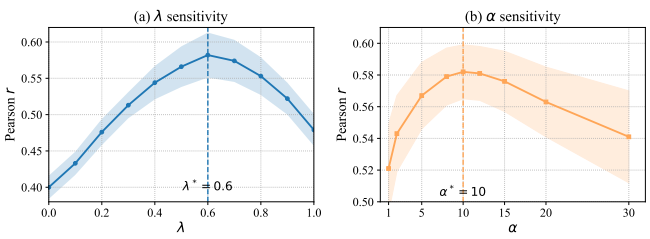

Fig. 5. Hyperparameter sensitivity of CL-ICE. Curves show the mean Pearson r across four language pairs; shaded bands indicate cross-language standard deviation. Both optima ($\lambda^* = 0.6$, $\alpha^* = 10$) lie at the center of flat plateaus, and the narrow bands confirm cross-language stability without per-language tuning.

Narrow shaded bands across all language pairs confirm that the globally selected pair $(\lambda, \alpha) = (0.6, 10)$ is stable, requiring no per-language tuning.

VI. Conclusion

This paper presents CL-ICE, a white-box framework for evaluating cross-language code translation through the internal representations of Code LLMs. By combining concept-layer semantic alignment with language-specific latent correctness probing, CL-ICE provides a reference-free and execution-free evaluation signal at inference time. Experimental results show that it correlates more strongly with functional correctness than conventional reference-based metrics, while remaining deterministic, efficient, and relatively robust to common judge biases. Rather than serving as a replacement for execution-based verification or LLM-as-a-Judge, CL-ICE is better understood as a scalable and complementary alternative for scenarios where execution is costly, references are limited, or large-scale screening is required. Future work may extend this framework to more diverse programming languages and integrate it with selective execution or external judging to form more flexible hybrid evaluation pipelines.

References

[1] J. D. Weisz, M. Muller, S. Houde, J. Richards, S. I. Ross, F. Martinez, M. Agarwal, and K. Talamadupula, "Perfection not required? human-ai partnerships in code translation," in *Proceedings of the 26th International Conference on Intelligent User Interfaces*, 2021, pp. 402–412.

[2] C. Ziftci, S. Nikolov, A. Sjövall, B. Kim, D. Codecasa, and M. Kim, "Migrating code at scale with llms at google," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, 2025, pp. 162–173.

[3] H. Zhang, C. David, M. Wang, B. Paulsen, and D. Kroening, "Scalable, validated code translation of entire projects using large language models," *Proceedings of the ACM on Programming Languages*, vol. 9, no. PLDI, pp. 1616–1641, 2025.

[4] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, P. Isabelle, E. Charniak, and D. Lin, Eds. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, Jul. 2002, pp. 311–318. [Online]. Available: https://aclanthology.org/P02-1040/

[5] A. Eghbali and M. Pradel, "Crystalbleu: precisely and efficiently measuring the similarity of code," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[6] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020.

[7] S. Zhou, U. Alon, S. Agarwal, and G. Neubig, "CodeBERTScore: Evaluating code generation with pretrained models of code," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, H. Bouamor, J. Pino, and K. Bali, Eds. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 13 921–13 937. [Online]. Available: https://aclanthology.org/2023.emnlp-main.859/

[8] J. Moon, Y. Hwang, D. Lee, T. Kang, Y. Kim, and K. Jung, "Don't judge code by its cover: Exploring biases in llm judges for code evaluation," *arXiv preprint arXiv:2505.16222*, 2025.

[9] Z. Yin, X. Gu, and B. Shen, "Neuron-guided interpretation of code llms: Where, why, and how?" *arXiv preprint arXiv:2512.19980*, 2025.

[10] F. Ribeiro, C. Spiess, P. Devanbu, and S. Nadi, "On llms' internal representation of code correctness," *arXiv preprint arXiv:2512.07404*, 2025.

[11] M. Chen, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[12] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[13] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[14] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[15] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu *et al.*, "Qwen2. 5-coder technical report," *arXiv preprint arXiv:2409.12186*, 2024.

[16] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. Xing *et al.*, "Judging llm-as-a-judge with mt-bench and chatbot arena," *Advances in neural information processing systems*, vol. 36, pp. 46 595–46 623, 2023.

[17] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[18] W. Tong and T. Zhang, "Codejudge: Evaluating code generation with large language models," *arXiv preprint arXiv:2410.02184*, 2024.

[19] H. Jiang, Y. Chen, Y. Cao, H.-y. Lee, and R. T. Tan, "Codejudgebench: Benchmarking llm-as-a-judge for coding tasks," *arXiv preprint arXiv:2507.10535*, 2025.

[20] A. Zou, L. Phan, S. Chen, J. Campbell, P. Guo, R. Ren, A. Pan, X. Yin, M. Mazeika, A.-K. Dombrowski *et al.*, "Representation engineering: A top-down approach to ai transparency," *arXiv preprint arXiv:2310.01405*, 2023.

[21] H. Abdi and L. J. Williams, "Principal component analysis," *Wiley interdisciplinary reviews: computational statistics*, vol. 2, no. 4, pp. 433–459, 2010.

[22] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[23] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz *et al.*, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, 2020, pp. 38–45.

[24] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker *et al.*, "Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks," *arXiv preprint arXiv:2105.12655*, 2021.