MATCH: Task-Driven Code Evaluation through Contrastive Learning

Marah Ghoummaid, Vladimir Tchuiev, Ofek Glick,
Michal Moschkovitz, and Dotan Di Castro
Bosch Research
marahghoummaid@gmail.com, vladimir.tchuiev@il.bosch.com,
ofek.gluck@gmail.com,michal.moshkovitz@il.bosch.com,
dotan.dicastro@il.bosch.com

Abstract

AI-based code generation is increasingly prevalent, with GitHub Copilot estimated to generate 46% of the code on GitHub. Accurately evaluating how well generated code aligns with developer intent remains a critical challenge. Traditional evaluation methods, such as unit tests, are often unscalable and costly. Syntactic similarity metrics (e.g., BLEU, ROUGE) fail to capture code functionality, and metrics like CodeBERTScore require reference code, which is not always available. To address the gap in reference-free evaluation, with few alternatives such as ICE-Score, this paper introduces MATCH, a novel reference-free metric. MATCH uses Contrastive Learning to generate meaningful embeddings for code and natural language task descriptions, enabling similarity scoring that reflects how well generated code implements the task. We show that MATCH achieves stronger correlations with functional correctness and human preference than existing metrics across multiple programming languages.

1 Introduction

The integration of large language models (LLMs) into modern software development has revolutionized the field through automated code generation. This field, often called neural natural-language-to-code (NL2Code), has advanced rapidly with a wave of models, from specialized open-source tools to powerful proprietary systems (Allal et al., 2023; Zhou et al., 2022; Fried et al., 2022; Rozière et al., 2024; Hui et al., 2024; Luo et al., 2023; Pavlichenko et al., 2025; Stallone et al., 2024; OpenAI, 2025; Google, 2025). These LLMs are revolutionizing software development, with code generation tools like GitHub Copilot now responsible for approximately 46% of code on GitHub (Gao and Research, 2024), highlighting the critical need for effective evaluation methods. Although executing generated code against unit tests is considered the gold standard for assessing functional correctness, it often proves impractical. Comprehensive test creation requires significant manual effort, especially in niche languages like Verilog and COBOL, where community-driven testing is limited and automated tools may be unavailable. Proprietary codebases further complicate evaluation when documentation remains locked within companies.

Execution-based methods for code validation evaluate the functional correctness of generated code using testing tools such as unit tests. However, these methods might overlook valuable code snippets that contain minor syntactic errors yet still capture essential logic. For software engineers who can easily fix minor bugs, these functionally sound snippets are highly valuable. This highlights the need for evaluation metrics that prioritize functional correctness over strict syntactic adherence, focusing on a code's ability to perform intended tasks despite imperfections.

Syntactic similarity metrics such as BLEU (Papineni et al., 2002) and ROUGE (Chin-Yew, 2004) offer computationally efficient alternatives but fail to capture the semantic meaning of code, being easily influenced by superficial variations in formatting or identifier names. Some natural language metrics have been enhanced to incorporate code-specific elements like Abstract Syntax Trees (ASTs) and Program Dependency Graphs (PDGs), including RUBY (Tran et al., 2019) and CodeBLEU (Ren et al., 2020). Despite these improvements, such metrics often lack correlation with human evaluations of code quality (Evtikhiev et al., 2023).

Recent approaches, such as CodeBERTScore (Zhou et al., 2023), aim to improve traditional metrics by leveraging pre-trained language models to evaluate the similarity of generated code against a reference implementation—a ground-truth solution known to successfully solve the task. While effective, these approaches fundamentally depend on

Findings of the Association for Computational Linguistics: EMNLP 2025, pages 11399–11414
November 4-9, 2025

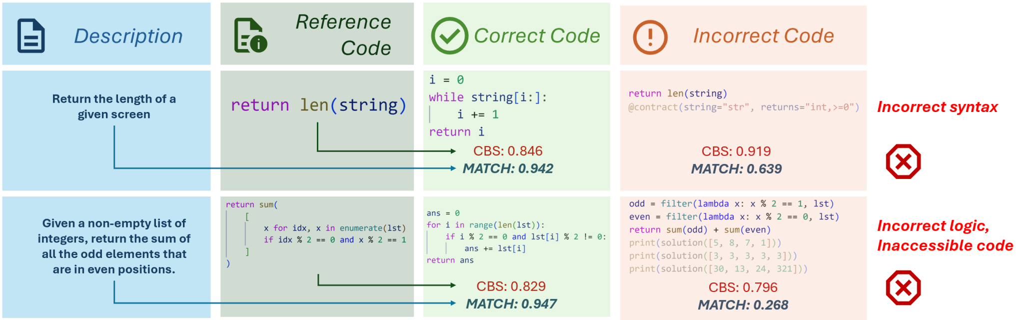


Figure 1: Two examples from the HumanEval dataset that illustrate MATCH's evaluation ability for correct and incorrect code snippets given a description (for MATCH) and a reference code (for CodeBERTScore, denoted as *CBS*). The type of incorrectness for the incorrect code is described as well. We show the score for both approaches for both correct and incorrect Python code, displaying MATCH's ability to differentiate between them compared to CodeBERTScore.

the availability of reference code, which prevents their use in novel code generation tasks where such references are unavailable or impractical to obtain. To address this limitation, the ICE-Score metric proposed by Zhuo (2023) prompts a large language model (LLM) like GPT to evaluate generated code quality directly, without relying on test cases or reference solutions. The LLM assesses various aspects such as correctness and usefulness, making ICE-Score a flexible, reference-free metric, though this advantage comes at the cost of increased latency and computational resources. A different approach is taken by CodeScore (Dong et al., 2025), which trains an LLM to predict functional correctness by estimating how many test cases a snippet passes and whether it runs without errors. CodeScore supports three variants: natural language only, reference code only, or both. The two variants involving reference code require the reference implementation, which can be restrictive and costly. Moreover, all three variants require precomputed percentages of passing test cases, further limiting applicability.

To overcome these challenges, we introduce MATCH: **M**etric for **A**ssessing **T**ask and **C**ode **H**armony, a novel *Contrastive Learning*-based approach to code evaluation that does not require reference code or execution. By learning meaningful embeddings for both code and natural language task descriptions, MATCH provides a nuanced assessment of code that captures both functional and semantic correctness. We show that MATCH correlates strongly with human judgments of code quality and significantly outperforms existing metrics, particularly in scenarios where traditional evaluation methods are impractical. In Figure 1, we present examples comparing MATCH with CodeBERTScore, illustrating how similarity to task descriptions helps MATCH avoid inherent biases of reference-based methods while yielding scores that reflect syntactic and logical correctness. Additional examples are provided in appendix B.

Contributions: In summary, this paper makes the following contributions:

- We introduce MATCH, a new metric for code evaluation based on *Contrastive Learning*.
- Our metric is designed to be practical and easy to use, requiring only a natural language description of the desired task, without the need for unit tests, existing APIs, or reference code.
- We demonstrate that our metric correlates well with human evaluations and functional correctness, outperforming existing metric over several popular programming languages.

The remainder of this paper is structured as follows: Section 2 provides a detailed overview of related work in code evaluation. Section 3 formulates the specific problem addressed in this paper. Section 4 describes the new method of MATCH. Section 5 and Section 6 present the experimental setup and benchmark results respectively. Section 7 presents performance comparisons for different architecture choices. Finally, Section 8 concludes the paper and discusses future directions, including a limitations discussion and potential areas for future research.

2 Related Work

Natural Language Metrics: Early code evaluation approaches have relied on syntactic similarity metrics adapted from natural language processing, such as BLEU (Papineni et al., 2002), which measures the overlap of n-grams between generated and reference text. CrystalBLEU (Eghbali and Pradel, 2022) modifies BLEU by excluding references that significantly overlap with the input, aiming to provide a more accurate assessment of diverse text generation quality. Other metrics include ROUGE (Chin-Yew, 2004), which evaluates recall by comparing n-gram overlap; METEOR (Banerjee and Lavie, 2005), which considers synonyms and stemming; and ChrF (Popović, 2015), which focuses on character n-grams. While these metrics compare generated code to reference implementations known to fulfill the intended tasks, they primarily assess syntactic similarity and fail to capture the functional aspects of the code, such as functional correctness.

Adapted Natural Language Metrics for Code: To address these shortcomings, some metrics have been enhanced to include code-specific elements, such as Abstract Syntax Trees (ASTs), which represent the hierarchical structure of code; Program Dependency Graphs (PDGs), which illustrate the dependencies between program components; and Data Flow Graphs, which depict the flow of data within a program. Examples of such enhanced metrics include RUBY (Tran et al., 2019) and CodeBLEU (Ren et al., 2020). CodeBLEU (Ren et al., 2020) enhances traditional BLEU by incorporating program-specific features, evaluating similarity through n-gram matches, weighted n-gram matches, AST matches, and Data Flow matches. Similarly, RUBY, proposed by (Tran et al., 2019) is a similarity metric that compares the PGDs of the generated and reference codes; if a PDG cannot be constructed, it falls back to comparing ASTs, and if an AST cannot be constructed, it uses weighted string edit distance between the tokenized reference and generated code. According to Evtikhiev et al. (2023) 's study, despite these adaptations, however, these metrics often lack correlation with human evaluations of code quality.

Advanced Evaluation Metrics for Code Generation: CodeBERTScore (Zhou et al., 2023) adapts BERTScore for code by leveraging embeddings from pre-trained code language models such as CodeBERT to measure similarity between generated and reference code. While it captures semantic overlap more effectively than surface-level metrics, it fundamentally requires reference implementations, and its scores are relative, making them difficult to interpret as an absolute measure of correctness for a single code snippet. A different approach is CodeScore (Dong et al., 2025), an LLM-based metric trained to predict functional correctness by estimating how many test cases a snippet passes and whether it runs without errors. It supports multiple input types—natural language only, reference code only, or both, but all formats require precomputed test-case labels, and some require reference code, which can be restrictive and costly. Another alternative is ICE-Score (Zhuo, 2023), a reference-free metric that prompts an LLM to directly evaluate code quality; however, this approach suffers from high latency and computational costs. These lines of work motivate alternative strategies, such as *Contrastive Learning*, which we leverage in our method to learn meaningful embeddings for code and task descriptions.

Contrastive Learning is a self-supervised paradigm that learns representations by contrasting positive and negative examples. The core idea is to bring similar instances closer in embedding space while pushing dissimilar ones apart, making it effective in scenarios with limited labeled data. Key works include (Oord et al., 2018), which introduced a self-supervised framework using InfoNCE loss, and (Radford et al., 2021), which demonstrated CLIP's effectiveness in a multi-modal setting with image-caption pairs. In the context of code, (Jain et al., 2020) emphasized semantic functionality through compiler-based transformations to create embeddings capturing code behavior. These works highlight Contrastive Learning's potential to model underlying code semantics, which we leverage in our evaluation task.

3 Problem Formulation

The input consists of: a natural language task description or instruction $t \in \mathcal{T}$, where $\mathcal{T}$ represents the set of all possible task descriptions, a code generation model generates a code snippet $c \in \mathcal{C}$, where $\mathcal{C}$ is the set of all possible code snippets, and a ground truth label y that is assigned to evaluate the quality of the code. This label can take one of the following forms:

- **Binary label** $y_{\text{bin}} \in \{0, 1\}$: Functional correctness is an example of a binary label, where 1 indicates the code is functionally correct for the given task and 0 indicates an incorrect implementation.

- **Continuous label** $y_{\text{cont}} \in [0, S]$: This label provides a continuous score between 0 and S representing the code's quality. For example, it can indicate how preferable a code snippet is to human developers. As the code becomes more preferable, the score increases towards S, the maximum possible value.

We use y to refer to either type of label, depending on the context.

Goal Our goal is to define a metric function $f : \mathcal{T} \times \mathcal{C} \to [-1, 1]$, where for a given task description t and a code snippet c, the function $f(t, c)$ produces a metric score that reflects the quality of the code snippet c with respect to the task described in t. Ideally, the metric function f should correlate with human evaluations and functional correctness.

In particular, for two candidate code snippets $c_1, c_2 \in \mathcal{C}$ implementing the same task $t \in \mathcal{T}$, we want the metric to indicate that if c_1 performs the task better and is preferred by human evaluators more than c_2, then c_1 should receive a higher metric score. Specifically, we seek a function f such that $f(t, c_1) > f(t, c_2)$ when c_1 is considered better than c_2 based on its functionality and human preference.

4 MATCH

This section introduces MATCH, our method for code evaluation. We detail its neural architecture in Section 4.1 and the contrastive loss objectives used for optimization in Section 4.2

4.1 Architecture

MATCH generates a similarity score between task descriptions and code snippets by learning embeddings. This process is illustrated in Figure 2.

A key aspect of MATCH is its ability to model both functional correctness (successful implementation) and human preferences (developer-favored code) in the similarity space. A close (positive) pair (t, c) represents a task description t and a corresponding code snippet c that either successfully implements the task or is highly preferred by developers.

Initial Embeddings The initial embedding layer employs a text encoder for the task description t and a code encoder for the code snippet c. These encoders can be either trained or kept frozen, an aspect we analyze in Section 7. These initial embeddings are then forwarded to the next layer aimed at enhancing and aligning both embeddings within a shared space.

Enhanced Embeddings Layer This layer enhances the initial embeddings by integrating cross-modal information between the task description and code snippet. It is always trained to effectively combine information from both inputs.

Formally, given a task description $t \in \mathcal{T}$ (in natural language) and its corresponding code snippet $c \in \mathcal{C}$, we denote the enhanced embeddings for the task description and code snippet as $\mathbf{e}_\text{t}$ and $\mathbf{e}_\text{c}$, respectively. Both embeddings reside in a d-dimensional embedding space, where $d \in \mathbb{Z}^+$ represents the dimensionality. In general, both $\mathbf{e}_t$ and $\mathbf{e}_c$ depend on both t and c, such that:

$$\mathbf{e}_t = f_t(t, c) \quad , \quad \mathbf{e}_c = f_c(t, c) \tag{1}$$

where f_t and f_c are functions that creates enriched embeddings from (t, c) pairs. These functions aggregate the initial embedding and embedding enhancement blocks as can be seen in Figure 2.

We propose two alternatives for the enhanced embeddings layer:

- **Cross-Attention:** This alternative employs two cross-attention components, one for each input, based on (Vaswani et al., 2017). For instance, the code component uses code as query and task description as key/value, and vice-versa for the text component. Each component is followed by a linear layer projecting embeddings into a shared space. A detailed illustration can be found in Figure 3a.

- **Linear: Linear:** This simpler approach applies separate linear transformations to each initial embedding, projecting them into a shared space. Here, $\mathbf{e}_t = f_t(t)$ and $\mathbf{e}_c = f_c(c)$. The shared space alignment is driven by the *Contrastive Learning* objective (Section 4.2). This approach is illustrated in Figure 3b.

We explore these alternatives further in Section 7.

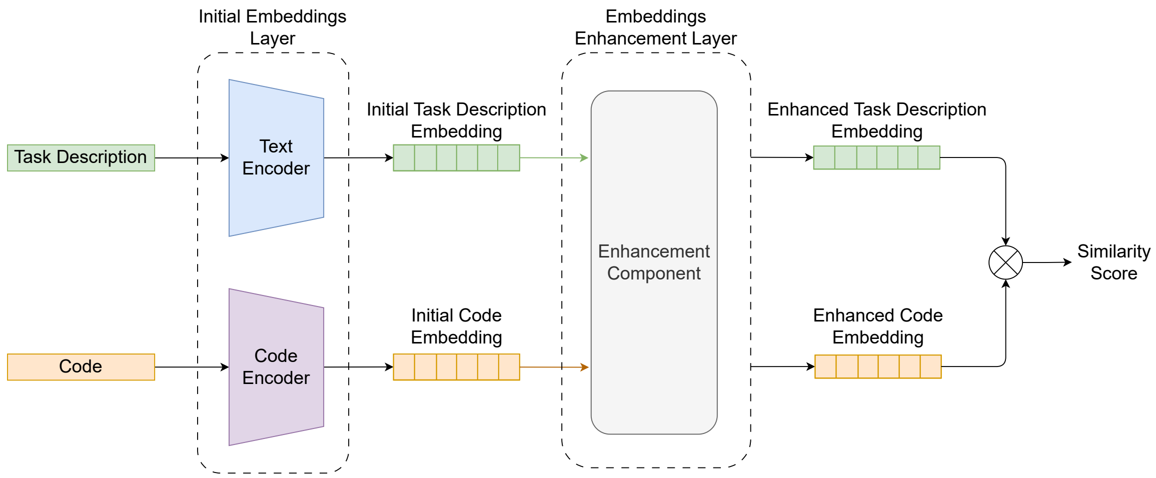


Figure 2: MATCH Architecture: This figure illustrates the general architecture of MATCH, featuring an Enhanced Embeddings Layer that integrates contextual information from task descriptions and code snippets. Specific implementations, including Linear and Cross-Attention enhancements, are shown in Figure 3. The architecture computes similarity between learned embeddings using an appropriate similarity function. such as Cosine Similarity.

Similarity Score Calculation: We compute the similarity between the enhanced embeddings $\mathbf{e}_t$ and $\mathbf{e}_c$ using *Cosine Similarity*:

$$\cos(\mathbf{e}_\mathrm{t}, \mathbf{e}_\mathrm{c}) = \frac{\mathbf{e}_\mathrm{t} \cdot \mathbf{e}_\mathrm{c}}{\|\mathbf{e}_\mathrm{t}\| \, \|\mathbf{e}_\mathrm{c}\|} \tag{2}$$

The final MATCH score, $f(t, c)$, is then given by:

$$f(t, c) = \cos\left(f_t(t, c), f_c(t, c)\right) \tag{3}$$

4.2 Contrastive Learning Objective

We optimize our architecture using contrastive loss objectives tailored for both binary and continuous label scenarios (as detailed in Section 3).

Given a task description $t \in \mathcal{T}$ (in natural language) and a code snippet $c \in \mathcal{C}$, The following sections 4.2.1 and 4.2.2 outline the loss functions for both binary and continuous labels.

4.2.1 Binary Labels

For a binary label $y_{bin} \in \{0, 1\}$ (e.g. indicating whether the code snippet successfully implements the task), we optimize the following loss function for binary loss $\mathcal{L}_{bin}$:

$$\mathcal{L}_{bin}(t, c, y) = \begin{cases} 1 - f(t, c) & , y = 1 \\ \max(0, f(t, c) - m) & , y = 0 \end{cases} \tag{4}$$

Here, m is a margin that is constrained within the range of $[-1, 1]$, defining the threshold distinguishing between similar and dissimilar pairs, $f(t, c)$ is computed via Equation (3), and $y \equiv y_{bin}$ for brevity.

4.2.2 Continuous Labels

For a continuous label on a scale $y_{cont} \in [0, S]$, with $S \in \mathbb{R}^+$ for example, a score reflecting human preference regarding the usefulness of the code snippet for the given task, we optimize the following loss function for continuous loss $\mathcal{L}_{cont}$:

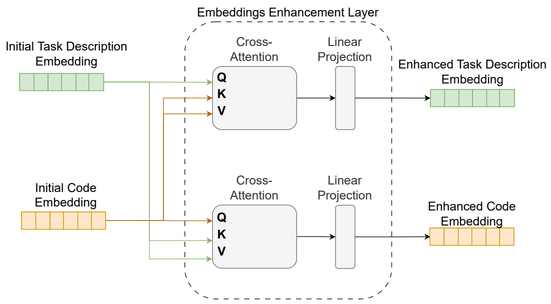
$$\mathcal{L}_\mathrm{cont}(t, c, y) = \mathrm{MSE}\left(sim(t, c) - \frac{y}{S}\right) \tag{5}$$

Here, $y \equiv y_{cont}$ for brevity, $f(t, c)$ is computed via Equation (3), and $sim(t, c)$ is defined as follows:

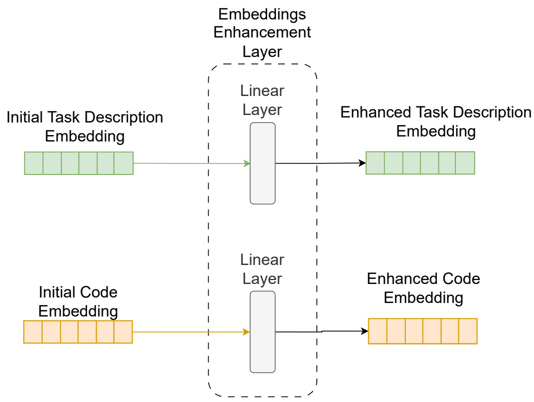
$$sim(t, c) \triangleq \frac{1 + f(t, c)}{2} \tag{6}$$

5 Experimental Setup

Baselines Our main comparisons are with **CodeBERTScore** (Zhou et al., 2023), which enhances code evaluation by incorporating natural language input alongside the generated code and using pre-trained models to assess consistency between them, and **ICE-Score** (Zhuo, 2023), which offers a reference-free alternative by employing a large language model (LLM) to evaluate code quality. We further consider **CodeScore** (Dong et al., 2025), an LLM-based metric trained to predict the pass rate of generated code. CodeScore has three reported variants: $(g{+}n)$, $(g{+}r)$, and $(g{+}r{+}n)$, which differ in the combination of generated code, reference



(a) Cross-Attention Layer: which integrates contextual information by using each input as both query and key-value pairs.



(b) Linear Layer: which applies separate linear transformations to refine features from both the task description and the code snippet.

Figure 3: An illustration of the specific implementations of the Enhanced Embeddings Layer: the Cross-Attention and Linear layers.

Metric	Java			Python			JavaScript			C++		
	τ	r_s	r_p	τ	r_s	r_p	τ	r_s	r_p	τ	r_s	r_p
BLEU	.460	.301	.291	.361	.334	.274	.219	.255	.242	.140	.215	.131
CodeBLEU	.492	.308	.318	.388	.315	.323	.238	.267	.296	.202	.158	.157
ROUGE-1	.481	.341	.356	.390	.334	.343	.238	.276	.309	.244	.319	.333
ROUGE-2	.436	.278	.308	.365	.307	.303	.199	.233	.281	.221	.270	.293
ROUGE-L	.464	.343	.360	.382	.352	.356	.207	.264	.303	.245	.323	.336
METEOR	.511	.343	.356	.426	.400	.408	.257	.314	.337	.204	.210	.219
chrF	.527	.346	.369	.439	.385	.393	.316	.344	.376	.319	.331	.348
CodeBERTScore	.547	.403	.406	.464	.418	.388	.331	.389	.327	.331	.390	.379
ICE-Score	.616	.504	.499	.341	.310	.315	**.540**	.437	.435	**.509**	.485	.486
MATCH (Base(T), CA)	.576	.684	.675	.613	.673	**.688**	.438	.602	.593	.395	.639	**.680**
MATCH (LS(F), Linear)	**.673**	**.701**	**.700**	**.668**	**.701**	.672	.439	**.630**	**.626**	.494	**.688**	.674

Table 1: Correlation of various metrics to functional correctness (τ = Kendall, r_s = Spearman, and r_p = Pearson) across programming languages using the HumanEval dataset. The best values in each column are bolded. Standard deviation are reported in Table 6

code, and natural language descriptions used as inputs; note two of the three variants require reference code. We adopt this notation when presenting results. Additionally, we compare our metric with established evaluation methods such as **BLEU**, **ROUGE**, **METEOR**, **chrF**, and **CodeBLEU**, all of which serve as baselines in our experiments.

MATCH Variants In our experiments, we explore several versions of MATCH, defined as MATCH (CodeEncoder(X), E). Here, **CodeEncoder** can be either a base encoder (**Base**) or a language-specific pre-trained encoder (**LS**), X indicates whether the encoder is frozen (F) or trainable (T), and E denotes the enhancement layer type, either cross-attention (CA) or linear ($Linear$). These variants allow us to assess the impact of encoder choice and enhancement architecture on performance.

Correlation metrics To evaluate our metric, we use three correlation metrics: Kendall's Tau (τ) (Kendall, 1938), Pearson's correlation coefficient (r_p) (Cohen et al., 2009), and Spearman's rank correlation coefficient (r_s) (Spearman, 1904). These metrics follow best practices in natural language evaluation and previous code evaluation studies (Zhou et al., 2023; Zhuo, 2023; Dong et al., 2025), allowing us to quantify how well each metric's scores align with reference values.

6 Experiments

In this section, we evaluate the utility of MATCH by examining its alignment with different types of labels. The first two experiments assess the correlation of MATCH with functional correctness: one using binary correctness labels (Section 6.1) and the other using pass-rate labels derived from execution tests (Section 6.2). The third experiment evaluates the correlation with human preferences for generated code (Section 6.3).

6.1 Functional Correctness

Dataset To evaluate alignment with functional correctness, we use the HumanEval dataset (Chen et al., 2021), which provides natural language descriptions of programming tasks, hand-crafted input-output test cases, and human-written reference solutions. Originally developed for Python, HumanEval has been translated into 18 programming languages by (Cassano et al., 2022), and includes predictions generated by the Codex model (code-davinci-002) along with their associated binary functional correctness labels, where each label indicates whether the generated code passes all test cases (1) or not (0). In our experiments, we focus on four widely-used languages: Java, C++, Python, and JavaScript, as highlighted in (Zhou et al., 2023).

Training The task-code pairs for training our model were directly sourced from HumanEval, leveraging its binary functional correctness labels as our optimization target ($\mathcal{L}_{\text{bin}}$, see Section 4.2.1). To ensure robust evaluation, we conducted five independent experiments, each using a different random split of the complete dataset into training, validation, and test sets. For each experiment, a separate model was trained using only the training and validation data, then evaluated on the held-out test set to prevent data leakage. Final performance is reported as the mean and standard deviation across all five experiments. Correlation metrics reported in Table 1 are averaged across five splits, with standard deviations provided in Table 6. Additional information about the dataset and it attributes, as well as further details on our model's architecture, training hyperparameters, and implementation, are provided in Appendix A.

Results For correlation with functional correctness, we report the average correlation scores (Pearson, Spearman, and Kendall's Tau) across the five test splits, as shown in Table 1. The corresponding standard deviations are provided in Table 6.

We evaluate two variations of MATCH: **MATCH (Base(T), CA)**, which uses a trained CodeBERT-base encoder with a cross-attention enhancement layer, and **MATCH (LS(F), Linear)**, which employs frozen language-specific encoders from Zhou et al. (2023) with a linear projection layer. Our results show that MATCH achieves the highest or comparable Kendall's Tau correlation with functional correctness across all four programming languages, with ICE-Score slightly outperforming for JavaScript. Moreover, MATCH consistently surpasses all baseline methods in both Spearman and Pearson correlation. Overall, MATCH demonstrates stronger alignment with functional correctness than existing evaluation metrics.

6.2 Functional Correctness with Pass Rate Labels

Dataset To complement our binary functional correctness experiments, we use MBPP-Eval (Dong et al., 2025), a large-scale dataset for functional correctness evaluation. Each task consists of a natural language description, a reference solution, and approximately 24 generated code candidates from various LLMs. Each task also includes around 100 automatically constructed test cases, enabling computation of a continuous pass-rate label (PassRatio) for each code snippet. We replicate the experimental setup from Dong et al. (2025), training on the provided training and development splits and reporting results on the test set. We additionally include **CodeScore** (Dong et al., 2025) in this experiment, since MBPP-Eval provides the pass-rate labels required for its training. The dataset contains 15,679 training examples and 3,000 examples each for development and test sets. The average PassRatio across splits is approximately 0.28, underscoring the difficulty of this benchmark.

Training For this experiment, we optimize our models using $\mathcal{L}_{\text{cont}}$, as defined in Section 4.2.2, which leverages the continuous PassRatio labels provided by MBPP-Eval. To ensure robustness, we train and evaluate across five random seeds and report the average results. The training and validation sets are used exclusively for optimization and model selection, while the test set is reserved for final evaluation to prevent data leakage. Further details on the model architecture, hyperparameters, and implementation are provided in Appendix A.

Results Table 2 reports results on the MBPP-Eval benchmark. CodeScore supports three variants: $g{+}n$ (generated code + natural language), $g{+}r$ (generated code + reference code), and $g{+}r{+}n$ (all three inputs). Since our method relies only on the natural language description of the task, we report the *g+n* variant in the table as the most directly comparable baseline.

Focusing on this fair comparison, MATCH achieves higher Kendall–Tau and Spearman correlations, and a comparable Pearson correlation, relative to CodeScore (g+n). This shows that MATCH aligns more closely with functional correctness without relying on reference code. CodeScore (g+r) and (g+r+n) attain slightly higher correlations, but both require reference implementations, making them less applicable in scenarios where such references are unavailable.

Across all other baselines, MATCH consistently attains the highest correlations for all metrics.

Metric	Python		
	τ	r_s	r_p
BLEU	.132	.188	.128
CodeBLEU	.150	.213	.189
ROUGE-1	.268	.375	.298
ROUGE-2	.226	.317	.261
ROUGE-L	.250	.351	.289
METEOR	.183	.259	.223
chrF	.180	.254	.256
CodeBERTScore	.240	.338	.302
ICE-Score	.275	.330	.322
CodeScore (g+n)	.341	.488	**.577**
MATCH (LS(F), Linear)	**.374**	**.516**	.574

Table 2: Correlation of various metrics to functional correctness (τ = Kendall, r_s = Spearman, and r_p = Pearson) across on MBPP-Eval dataset. The best values in each column are bolded.

6.3 Human Preference

Dataset To evaluate the correlation between MATCH and human preferences, we use human annotations from Evtikhiev et al. (2023) for the CoNaLa benchmark (Yin et al., 2018). CoNaLa focuses on Python code generation from natural language descriptions, with data sourced from StackOverflow. For this benchmark, experienced software developers graded code snippets produced by five different models on a scale from 0 to 4, where 0 indicates irrelevance and 4 signifies that the code effectively addresses the problem. These continuous scores serve as human preference labels.

Training For this experiment, MATCH is trained and evaluated on CoNaLa using the human preference labels. Similar to the functional correctness experiment (Section 6.1), we conducted five independent experiments using different random splits of the dataset into training, validation, and test sets. In each experiment, the model was trained on the training and validation data and evaluated on the held-out test set to prevent data leakage. Given the continuous nature of the labels, we employ $\mathcal{L}_{\text{cont}}$, as defined in Section 4.2.2, as our optimization objective. Additional details on the dataset, its attributes, and our model's architecture, training hyperparameters, and implementation are provided in Appendix A.

Results For correlation with human preferences, we report the average metrics (Pearson, Spearman, and Kendall-Tau) across the five test splits, as shown in Table 3. The standard deviation across splits is provided in Table 7. We find that **MATCH (Base(T), CA)** achieves comparable Kendall-Tau correlation to CodeBERTScore, while outperforming all baselines in both Spearman and Pearson correlations. These results indicate that MATCH aligns more strongly with human judgments overall.

Metric	Python		
	τ	r_s	r_p
BLEU	.148	.272	.286
CodeBLEU	.256	.374	.424
ROUGE-1	.505	.633	.638
ROUGE-2	.357	.525	.549
ROUGE-L	.488	.617	.627
METEOR	.168	.272	.330
chrF	.507	.622	.626
CodeBERTScore	**.577**	.662	.660
ICE-Score	.311	.561	.636
MATCH (Base(T),CA)	.568	**.721**	**.741**

Table 3: Correlation to functional correctness of various metrics (τ = Kendall, r_s = Spearman, and r_p = Pearson) on CoNaLa Dataset. Best values per column are bolded. Standard deviation are reported in Table 7

7 Analysis

This section presents an analysis of the different components within our architecture and examines the trade-offs associated with various design choices, as shown in Table 4. We address the following aspects:

Metric	Java			Python			JavaScript			C++			CoNala		
	τ	r_s	r_p	τ	r_s	r_p	τ	r_s	r_p	τ	r_s	r_p	τ	r_s	r_p
MATCH (LS(F),Linear)	**.673**	**.701**	**.700**	**.668**	**.701**	.672	.439	**.630**	**.626**	**.494**	**.688**	.674	.503	.679	.712
MATCH (LS(F),CA)	.635	.692	.663	.604	.687	.645	.385	.603	.566	.450	.678	.665	.442	.642	.671
MATCH (Base(T),CA)	.576	.684	.675	.613	.673	**.688**	.438	.602	.593	.395	.639	**.680**	**.568**	**.721**	**.741**
MATCH (Base(T),Linear)	.510	.581	.539	.566	.657	.661	**.464**	.624	.613	.305	.636	.647	**.560**	**.726**	**.744**

Table 4: Correlation to functional correctness of different variations of MATCH (τ = Kendall, r_s = Spearman, and r_p = Pearson) across programming languages from HumanEval and correlation to human preferences on CoNaLa Datasets. Best values per column are bolded. Standard devaitions are presented in Table 8.

Impact of Language-Specific Code Encoders vs. a Base Encoder Table 4 shows that using a language-specific code encoder, when available, yields improved correlations with functional correctness. However, a base code encoder tends to achieve better correlations with human preference.

Impact of Training the Code Encoder vs. Freezing It Table 4 indicates that training a base code encoder achieves reasonable results when a language-specific encoder is not available. When language-specific encoders are available, MATCH generally achieves better results. For the CoNaLa dataset, training the base code encoder consistently results in higher correlations with human preference. However, for HumanEval, freezing a language-specific encoder generally yields better performance.

Impact of a Cross-Attention Enhancement Layer vs. a Linear Layer Table 4 suggests that when a language-specific code encoder is available, a linear enhancement layer is sufficient and achieves strong correlations with functional correctness. Conversely, in the absence of a language-specific encoder, a cross-attention layer can compensate and yield good correlations with functional correctness. Furthermore, a linear enhancement layer consistently achieves the best correlations with human preference.

Architecture Choice - Summary MATCH using language-specific encoders tends to correlate better with functional correctness. When these encoders are available, a linear enhancement layer (Lang(F), Linear) is often sufficient and resource-efficient. Alternatively, MATCH with a base code encoder and cross-attention (Base, CA) provides a good trade-off: it can perform reasonably well while requiring less data and computational resources than training or using language-specific encoders. Notably, when using the same code encoder, MATCH consistently shows a stronger correlation with human preference when a linear enhancement layer is used compared to cross-attention. Overall, MATCH provides a flexible framework for code evaluation, allowing users to select architectures that balance performance, data needs, and resource availability. The continued investigation of different architectures within MATCH presents a promising and interesting avenue for future work.

8 Conclusion

We introduced MATCH, a practical, reference-free metric for evaluating code implementations against natural language descriptions. Addressing the critical challenge of assessing code quality without requiring reference implementations or extensive test cases, MATCH leverages contrastive learning to embed code and text into a shared semantic space. Our method first independently encodes code and natural language, then enriches their representations through Cross-modal Interaction. To quantify alignment using cosine similarity. Empirical experiments demonstrated MATCH's superior performance, achieving significantly stronger correlations with both functional correctness and human preference compared to existing baselines. This positions MATCH as a powerful and flexible solution for efficiently assessing code quality, particularly in scenarios where traditional evaluation methods are impractical.

Looking ahead, MATCH can be extended to evaluate a broader spectrum of code quality beyond functional correctness and human preference. This includes non-functional aspects such as efficiency (time and space complexity), readability, maintainability, and security vulnerabilities, enabling a more holistic and practical assessment of generated code. Another promising direction is testing the method's robustness under imperfect or noisy inputs, which are common in real-world scenarios.

Limitations

MATCH is primarily designed with software developers in mind. As such, it tends to assign high scores to code that is syntactically and semantically correct, even if it contains minor typos. While this is desired for developer-facing tools and early-stage prototyping, it may not suffice for evaluating production-level code, where robustness, error handling, and performance are critical. To address these gaps, we recommend supplementing MATCH with additional evaluation strategies, such as unit tests for edge cases or compilation checks.

Another consideration is the reliance on model fine-tuning. Our results indicate that the performance of the proposed metric improves when the underlying model is fine-tuned on task-specific data. However, this fine-tuning process can be computationally expensive and may not be practical in all settings. Fortunately, our experiments show that even without fine-tuning, MATCH achieves competitive performance, suggesting it can be effectively applied in low-resource or general-use scenarios.

Finally, while our method performs well on established benchmarks that are widely used and well-understood in the community, its effectiveness on entirely new or out-of-distribution tasks remains uncertain. This limitation is not unique to our approach, it is a fundamental challenge shared by all existing and future evaluation metrics aimed at providing universal assessment of code generation and natural language tasks.

Acknowledgments

The authors would like to thank Amir Feder for their valuable feedback and advice on this paper. We are also grateful to the anonymous reviewers for their useful discussion and comments.

References

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, and 1 others. 2023. Santacoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*.

Satanjeev Banerjee and Alon Lavie. 2005. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 65–72.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, and 1 others. 2022. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. *arXiv preprint arXiv:2208.08227*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Lin Chin-Yew. 2004. Rouge: A package for automatic evaluation of summaries. In *Proceedings of the Workshop on Text Summarization Branches Out, 2004*.

Israel Cohen, Yiteng Huang, Jingdong Chen, Jacob Benesty, Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. 2009. Pearson correlation coefficient. *Noise reduction in speech processing*, pages 1–4.

Yihong Dong, Jiazheng Ding, Xue Jiang, Ge Li, Zhuo Li, and Zhi Jin. 2025. Codescore: Evaluating code generation by learning code execution. *ACM Transactions on Software Engineering and Methodology*, 34(3):1–22.

Aryaz Eghbali and Michael Pradel. 2022. Crystalbleu: precisely and efficiently measuring the similarity of code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12.

Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. 2023. Out of the bleu: how should we assess quality of the code generation models? *Journal of Systems and Software*, 203:111741.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.

Ya Gao and GitHub Customer Research. 2024. Research: Quantifying github copilot's impact in the enterprise with accenture. https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-in-the-enterprise-with-accenture/.

Google. 2025. Gemini 1.5 Pro (may 20 2025 version) [large multimodal model]. 2 M-token context; transcript non-recoverable.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, and 5 others. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph E. Gonzalez, and Ion Stoica. 2020. Contrastive code representation learning. *arXiv preprint*.

Maurice G Kendall. 1938. A new measure of rank correlation. *Biometrika*, 30(1-2):81–93.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *Preprint*, arXiv:2306.08568.

Aaron van den Oord, Yazhe Li, and Oriol Vinyals. 2018. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748*.

OpenAI. 2025. ChatGPT (may 14 2025 version) [large language model]. GPT-4o engine; transcript non-recoverable.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Nikita Pavlichenko, Iurii Nazarov, Ivan Dolgov, Ekaterina Garanina, Karol Lasocki, Julia Reshetnikova, Sergei Boitsov, Ivan Bondyrev, Dariia Karaeva, Maksim Sheptyakov, Dmitry Ustalov, Artem Mukhin, Semyon Proshev, Nikita Abramov, Olga Kolomyttseva, Kseniia Lysaniuk, Ilia Zavidnyi, Anton Semenkin, Vladislav Tankov, and Uladzislau Sazanovich. 2025. Mellum-4b-base.

Maja Popović. 2015. chrf: character n-gram f-score for automatic mt evaluation. In *Proceedings of the tenth workshop on statistical machine translation*, pages 392–395.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, and 1 others. 2021. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PmLR.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, and 7 others. 2024. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

Charles Spearman. 1904. The proof and measurement of association between two things. *The American Journal of Psychology*, 15(1):72–101.

Matt Stallone, Vaibhav Saxena, Leonid Karlinsky, Bridget McGinn, Tim Bula, Mayank Mishra, Adriana Meza Soria, Gaoyuan Zhang, Aditya Prasad, Yikang Shen, Saptha Surendran, Shanmukha Guttula, Hima Patel, Parameswaran Selvam, Xuan-Hong Dang, Yan Koyfman, Atin Sood, Rogerio Feris, Nirmit Desai, and 3 others. 2024. Scaling granite code models to 128k context. *Preprint*, arXiv:2407.13739.

Ngoc Tran, Hieu Tran, Son Nguyen, Hoan Nguyen, and Tien Nguyen. 2019. Does bleu score work for code migration? In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 165–176. IEEE.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th international conference on mining software repositories*, pages 476–486.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, and 1 others. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684.

Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. 2023. Codebertscore: Evaluating code generation with pretrained models of code. *arXiv preprint arXiv:2302.05527*.

Shuyan Zhou, Uri Alon, Frank F Xu, Zhiruo Wang, Zhengbao Jiang, and Graham Neubig. 2022. Docprompting: Generating code by retrieving the docs. *arXiv preprint arXiv: 2207.05987*.

Terry Yue Zhuo. 2023. Ice-score: Instructing large language models to evaluate code. *arXiv preprint arXiv:2304.14317*.

A Additional Experimental Detail

Here we provide additional experimental details and full results for our experiment for Correlation to functional correctness and human preference in Appendix A.1 and Appendix A.2, respectively.

for both experiments we do the following, we split the data to 5 splits, and for each split we run all methods, and report the mean correlation for each Method. For CodeBERTScore (Zhou et al., 2023) we obtian the scores according to the instruction in https://github.com/neulab/code-bert-score/tree/main. For ICE-Score we run their method following the instructions in https://github.com/terryyz/ice-score/tree/main. We modify their evaluator by calling the LLM using an interface we have access to. Originally, they use GPT-3.5 (`GPT-3.5-turbo3`) [1]). Due to the fact that we don't have access to this version, we run their method with (`GPT-4o`)[2]. As for MATCH, for the Encoder we use `bert-base-uncased` as the text encoder. For the code encoder, we use `microsoft/codebert-base` for the base encoder, and `neulab/codebert-lang` by Zhou et al. (2023), for lang $\in \{\text{python}, \text{java}, \text{javascript}, \text{cpp}\}$. MATCH's was implemented using `pytorch_lightning` model as a wrapper model that can receive any model for both encoders. and any model as the enhancement layer. The MATCH models were trained with `learning_rate` $= 3e-5$, `max_epochs` $= 50$, and early stopping with `patience` $= 3$, and a `batch_size=16` In addition, we used `temperature` $= 0.07$ for smoothing the similarities in loss calculation. The final embedding dimensions were `embedding_dim`,$= 768$.

For the Cross-attention enhancements layer, we use one pytorch `MultiheadAttention`, with `num_heads`$= 8$, `dropout_rate`$= 0.2$, and `embed_dim`$= 0.2$, followed by a normalization and linear layers. with `in_features`$= 768$, and `out_features`$= 768$. As for the Linear enhancement layer, we used a straightforward linear layer with the same parameters.

Finally We used 240 GPU-hours on a single A100 GPU.

Baseline Metrics In Table 5 we summarize the used existing packages for each of the baselines metrics we compare MATCH to in Section 6. These packages are all licensed under either the MIT or Apache-2.0 license.

[1]https://platform.openai.com/docs/models/gpt-3-5

[2]https://platform.openai.com/docs/models/gpt-4o

A.1 Functional Correctness Experiment

HumanEval Dataset The HumanEval benchmark (Chen et al., 2021) is a widely used dataset for evaluating code generation models, consisting of 164 Python programming problems, each with a natural language goal in English, human-written input-output test cases (averaging 7.7 per problem), and a human-written reference solution. Each example aims to evaluate the model on functional correctness. While the original HumanEval is in Python, an extension by Cassano et al. (2022) translated the problems to 18 other languages, including Java, C++, and JavaScript, facilitating evaluation across diverse programming paradigms. This translated version also includes predictions from `code-davinci-002` along with their corresponding functional correctness scores. Utilizing data from the HumanEval-X dataset Zheng et al. (2023) to provide reference solutions in the translated languages, in work we focuses on evaluating code generation performance in Java, C++, Python, and JavaScript.

A.2 Human preference Experiment

CoNaLa Dataset The CoNaLa benchmark (Yin et al., 2018) serves as a dataset for assessing the capability of models to generate Python code from natural language instructions written in English. which focuses on generating Python code from natural language descriptions sourced from Stack-Overflow. Evtikhiev et al. (2023) provided human annotations for the dataset. Experienced software developers graded the code snippets generated by five models on a scale from zero to four, where zero indicates irrelevance and four signifies that the code effectively addresses the problem.This dataset features a collection of 2,860 code snippets, produced by five different models across 472 distinct examples from CoNaLa. Each generated snippet has been assessed by approximately 4.5 experienced software developers. These human annotations provide a means to correlate automated evaluation metrics with human preferences in code generation quality.

A.3 Human Preference

Dataset To evaluate the correlation between each metric and human preferences, we utilize human annotations from Evtikhiev et al. (2023) for the

Metric	Package
BLEU	nltk
CodeBLEU	CodeBlEU
ROUGE	nltk
METEOR	nltk
chRF	sacrebleu
CodeBERTScore	CodeBERTScore
ICE-Score	ICE-Score

Table 5: Packages we used for the baselines we compare our method to in the experiments.

Metric	Java			Python			JavaScript			C++		
	τ	r_s	r_p	τ	r_s	r_p	τ	r_s	r_p	τ	r_s	r_p
BLEU	.460±.017	.301±.006	.291±.012	.361±.036	.334±.010	.274±.015	.219±.044	.255±.047	.242±.039	.140±.019	.215±.023	.131±.015
CodeBLEU	.492±.014	.308±.011	.318±.016	.388±.027	.315±.011	.323±.018	.238±.052	.267±.024	.296±.031	.202±.012	.158±.008	.157±.006
ROUGE-1	.481±.009	.341±.007	.356±.005	.390±.044	.334±.019	.343±.017	.238±.057	.276±.029	.309±.030	.244±.014	.319±.011	.333±.010
ROUGE-2	.436±.014	.278±.010	.308±.009	.365±.035	.307±.014	.303±.015	.199±.043	.233±.041	.281±.040	.221±.018	.270±.009	.293±.011
ROUGE-L	.464±.015	.343±.006	.360±.006	.382±.037	.352±.021	.356±.018	.207±.069	.264±.033	.303±.036	.245±.016	.323±.013	.336±.013
METEOR	.511±.022	.343±.007	.356±.011	.426±.036	.400±.013	.408±.015	.257±.048	.314±.040	.337±.040	.204±.030	.210±.014	.219±.010
chrF	.527±.009	.346±.014	.369±.012	.439±.024	.385±.012	.393±.016	.316±.031	.344±.032	.376±.030	.319±.014	.331±.010	.348±.001
CodeBERTScore	.547±.007	.403±.009	.406±.008	.464±.028	.418±.015	.388±.012	.331±.066	.389±.025	.327±.017	.331±.017	.390±.013	.379±.010
ICE-Score	.616±.021	.504±.013	.499±.012	.341±.015	.310±.013	.315±.013	**.540±.072**	.437±.033	.435±.031	**.509±.012**	.485±.015	.486±.015
MATCH (Base(T), CA)	.576±.046	.684±.012	.675±.017	.613±.026	.673±.013	**.688±.016**	.438±.074	.602±.041	.593±.047	.395±.081	.639±.027	**.680±.015**
MATCH (LS(F), Linear)	**.673±.025**	**.701±.013**	**.700±.017**	**.668±.029**	**.701±.011**	.672±.015	.439±.107	**.630±.021**	**.626±.021**	**.494±.015**	**.688±.013**	.674±.012

Table 6: Mean and standard deviation of the correlations of various metrics (τ = Kendall, r_s = Spearman, and r_p = Pearson) across programming languages from HumanEval Dataset. Correlations are presented in Table 1.

.

Metric	Python		
	τ	r_s	r_p
BLEU	.148 ± .061	.272 ± .010	.286 ± .022
CodeBLEU	.256 ± .039	.374 ± .034	.424 ± .030
ROUGE-1	.505 ± .049	.633 ± .036	.638 ± .030
ROUGE-2	.357 ± .063	.525 ± .027	.549 ± .030
ROUGE-L	.488 ± .060	.617 ± .036	.627 ± .027
METEOR	.168 ± .053	.272 ± .009	.330 ± .033
chrF	.507 ± .066	.622 ± .024	.626 ± .028
CodeBERTScore	**.577** ± .074	.662 ± .033	.660 ± .033
ICE-Score	.311 ± .060	.561 ± .036	.636 ± .027
MATCH (Base(T),CA)	.568 ± .065	**.721** ± .027	**.741** ± .021

Table 7: Mean and standard deviation of the correlations of various metrics (τ = Kendall, r_s = Spearman, and r_p = Pearson) on CoNaLa Dataset. Correlations are presented in Table 3.

CoNaLa benchmark (Yin et al., 2018), which focuses on generating Python code from natural language descriptions sourced from StackOverflow. Experienced software developers graded the code snippets generated by five models on a scale from zero to four, where zero indicates irrelevance and four signifies that the code effectively addresses the problem.

In this experiment, the dataset contains continuous labels. Therefore, we will use $\mathcal{L}_{\text{cont}}$, as defined in Section 4.2.2, as our optimization objective. Further details can be found in Appendix A.2.

A.4 Analysis Standard Deviations

Standard deviations of the analysis presented in Table 4 are reported in Table 8.

Metric	Java			Python			JavaScript			C++			CoNala		
	τ	r_s	r_p	τ	r_s	r_p	τ	r_s	r_p	τ	r_s	r_p	τ	r_s	r_p
MATCH (LS(F),Linear)	**.673 ± .025**	**.701 ± .013**	**.700 ± .017**	**.668 ± .029**	**.701 ± .011**	.672 ± .015	.439 ± .107	**.630 ± .021**	**.626 ± .021**	**.494 ± .015**	**.688 ± .013**	.674 ± .012	.503 ± .059	.679 ± .016	.712 ± .015
MATCH (LS(F),CA)	.635 ± .020	.692 ± .006	.663 ± .021	.604 ± .028	.687 ± .010	.645 ± .016	.385 ± .099	.603 ± .020	.566 ± .032	.450 ± .032	.678 ± .009	.665 ± .007	.442 ± .104	.642 ± .014	.671 ± .022
MATCH (Base(T),CA)	.576 ± .046	.684 ± .012	.675 ± .017	.613 ± .026	.673 ± .013	**.688 ± .016**	.438 ± .074	.602 ± .041	.593 ± .047	.395 ± .081	.639 ± .027	**.680 ± .015**	**.568 ± .065**	**.721 ± .027**	**.741 ± .021**
MATCH (Base(T),Linear)	.510 ± .119	.581 ± .131	.539 ± .193	.566 ± .042	.657 ± .022	.661 ± .035	**.464 ± .118**	.624 ± .030	.613 ± .024	.305 ± .063	.636 ± .017	.647 ± .030	.560 ± .063	**.726 ± .024**	**.744 ± .025**

Table 8: Mean and standard deviation of the correlation to functional correctness of different variations of MATCH (τ = Kendall, r_s = Spearman, and r_p = Pearson) across programming languages from HumanEval and correlation to human preferences on CoNaLa Datasets. Correlations are presented in Table 4.

B Examples

Example 1: MATCH identifies a mostly correct candidate with a minor bug.
The code is mostly correct but contains a minor bug related to case sensitivity in vowel removal, affecting the consonant count. The MATCH score indicates its potential despite this issue.

Language	Aspect	Task Description	Code Candidate	Label	ICE	MATCH

Python	Functional Correctness	Given a string s and a natural number n, implement a function that returns all words from string s with exactly n consonants in order. Return empty list if s is empty.	`arr = []`
			`for word in s.split():`
			`s = word.replace("a", "").replace("e", "")\`
			`.replace("i", "").replace("o", "")\`
			`.replace("u", "")`
			`if len(s) == n:`
			`arr.append(word)`
			`return arr`
				0	0	0.774

Example 2: MATCH shows a clearer separation between wrong and right solutions.
The first candidate fails to correctly implement the task by mis-checking the oddness of the first digit and neglecting the condition of being greater than 10 while the second candidate performs correctly. MATCH score reflects this significant error, while CodeBERTScore gives close scores for both candidates.

Language	Aspect	Task Description	Code Candidate	Label	ICE	MATCH

Java	Functional Correctness	Write a function that counts numbers >10 with odd first and last digits.	`return (int) nums.stream()`
			`.filter(i -> Math.abs(i) % 2 == 1`
			`&& i % 10 % 2 == 1`
			`&& i / 10 % 2 == 1)`
			`.count();`
				0	0.699	0.218

			`return (int) nums.stream()`
			`.filter(num -> num > 10)`
			`.filter(num ->`
			`Integer.toString(num)`
			`.charAt(0) % 2 != 0`
			`&& Integer.toString(num)`
			`.charAt(Integer.toString`
			`(num).length() - 1)`
			`% 2 != 0`
			`).count();`
				1	0.763	0.714

Example 3: MATCH has a better understanding of the code semantics.
Both code candidates fail to correctly implement the task. The first candidate is incorrect, while the second candidate sums the binary digits instead of the decimal digits. The MATCH score accurately reflects the errors in both candidates, unlike the ICE-Score, which misclassifies the second candidate as correct.

Language	Aspect	Task Description	Code Candidate	Label	CodeBERTScore	ICE	MATCH
Java	Functional Correctness	Given a positive integer N, return the total sum of its digits in binary.	`return null;`	0	0.617	0	0.125

			`String s = Integer.toBinaryString(N);`
			`int sum = 0;`
			`for (int i = 0; i < s.length(); i++) {`
			`sum += Integer.parseInt(`
			`String.valueOf(s.charAt(i)));`
			`}`
			`return Integer.toString(sum);`
				0	0.896	4	0.331

Example 4: MATCH Effectively Differentiating between correct and incorrect code
The first code candidate correctly downloads a file, while the second incorrectly attempts to open a file without downloading it. The MATCH score reflects this, but the CodeBERTScore does not adequately differentiate between the candidates, and the ICE-Score gives both a score of 0.

Language	Aspect	Task Description	Code Candidate	Label	CodeBERTScore	ICE	MATCH
Python	Human Preference	Download a file "url" over HTTP and save to "mp3.mp3".	`importurllib2,os;` `urllib2.urlretrieve("http://www.example.com/songs/mp3.mp3","mp3.mp3")`	3.75	0.945	0	0.727
			`withopen("filename.txt", shell=True)`	0	0.641	0	0.139

Example 5: MATCH Effectively Differentiating between correct and incorrect code

The first candidate is not useful at all as it only reads the file without erasing its contents. In contrast, the second candidate correctly opens the file in write mode, successfully erasing its contents. This example shows that the MATCH score better differentiates between the candidates than CodeBERTScore.

Language	Aspect	Task Description	Code Candidate	Label	CodeBERTScore	MATCH
Python	Human Preference	Erase all the contents of a file filename.	`withopen("filename.txt")asf: #NEWLINE# #INDENT#f.read()`	0.667	0.78	-0.55
			`open(filename,w).close()`	4	0.984	0.919