MATCH: Task-Driven Code Evaluation through Contrastive Learning

Anonymous ACL submission

Abstract

AI-based code generation is increasingly prevalent, with GitHub Copilot estimated to generate 46% of GitHub code. Accurately evaluating the alignment of generated code with developer intent remains a critical challenge. Traditional evaluation methods, like 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 that is not always available. Recognizing the gap in reference-free evaluation, with few alternatives like ICE-Score, this paper introduces MATCH, a novel reference-free metric. MATCH employs 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 demonstrate that MATCH achieves stronger correlations with functional correctness and human preference than existing metrics across several programming languages.

1 Introduction

Software developers are increasingly utilizing code generation techniques powered by large language models (LLMs), also known as neural natural-language-to-code (NL2Code) (Allal et al., 2023; Zhou et al., 2022; Fried et al., 2022). 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. While executing generated code against unit tests is considered the gold standard for assessing functional correctness, it's often proves impractical due to the manual effort required for comprehensive test creation, particularly in niche languages like Verilog and COBOL, where community-driven testing efforts are limited and automatic testing tools may be unavailable. Additionally, proprietary codebases may have documentation locked within companies, complicating evaluation further.

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. While effective, these approaches fundamentally depend on the availability of reference code, which poses challenges in novel code generation tasks where such references may be unavailable or impractical to obtain. To address this limitation, ICE-Score metric proposed by Zhuo (2023) takes a different approach by prompting a large language model (LLM) like GPT to evaluate generated code quality directly, without

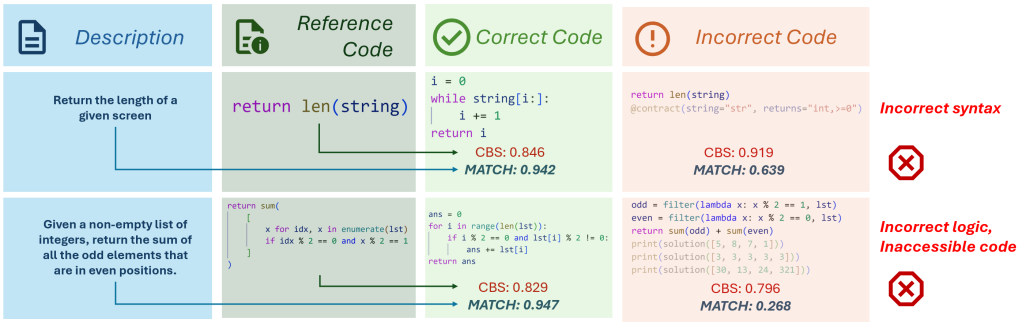


Figure 1: Two examples from the HumanEval dataset that illustrate MATCH's evaluation ability for correct and incorrect code snippets given a description (for MATCH) and a reference code (for CodeBERTScore, denoted as *CBS*). The type of incorrectness for the incorrect code is described as well. We show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

relying on test cases or reference solutions. Instead, the LLM assesses various aspects such as correctness and usefulness, making ICE-Score a flexible, reference-free metric. However, this advantage comes at the cost of increased latency and computational resources due to the reliance on LLM inference.

To overcome these challenges, we introduce MATCH: **M**etric for **A**ssessing **T**ask and **C**ode **H**armony, a novel *Contrastive Learning*-based metric for evaluating code generation that does not require reference code or execution. By learning meaningful embeddings for both code and natural language task descriptions, MATCH provides a more nuanced and comprehensive assessment of code, capturing both functional correctness and semantic similarity. We demonstrate MATCH's effectiveness as a reliable metric for evaluating code generation, particularly in scenarios where traditional evaluation methods are impractical, by showing that it correlates strongly with human judgments of code quality and significantly outperforms existing metrics in evaluating code generated for novel tasks. In Figure 1 we present code snippet evaluation examples for MATCH in comparison to CodeBERTScore. We showcase that by calculating similarity scores to task descriptions, MATCH avoids inherit biases when comparing to a reference code, yielding an informative score that considers syntactic and logical correctness.

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) enhances code evaluation by incorporating the natural language input alongside the generated code and using pre-trained models to assess consistency between them. However, it requires reference code which is not always available and yields relative scores, posing a challenge for consistent evaluation. CodeScore (Dong et al., 2025) teaches an LLM to mimic execution-based correctness for fast, reference-free evaluation, but requires extensive training data and is limited in generalization. ICE-Score (Zhuo, 2023) offers another reference-free alternative by using an LLM to evaluate code quality; however, this approach suffers from high latency and computational costs. Our work leverages *Contrastive Learning* to create meaningful embeddings for both code and task descriptions, further enhancing the evaluation process.

Contrastive Learning is a self-supervised learning paradigm that focuses on learning representations by contrasting positive and negative examples. The core idea is to bring similar instances closer together in the embedding space while pushing dissimilar instances apart, making it particularly effective in scenarios with limited labeled data. Key works in this area include (Oord et al., 2018), which introduced a self-supervised framework using InfoNCE loss to learn high-level representations from raw data, and (Radford et al., 2021), which showcased the power of CLIP in a multi-modal setting by training on a large dataset of image-caption pairs. Additionally, (Jain et al., 2020) emphasized semantic functionality in code representation learning through self-supervised Contrastive Learning, employing compiler-based transformations to create embeddings that capture the essence of code behavior. These works highlight the potential of Contrastive Learning to capture the underlying semantics of code, which we aim to 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 $f : \mathcal{T} \times \mathcal{C} \rightarrow [-1, 1]$, 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

In this section, we present MATCH, our method for designing a metric for code evaluation. The proposed metric aims to effectively assess the quality of code snippets with respect to a natural language task descriptions, using a new neural architecture. We start by outlining this architecture in Section 4.1. Following that, in Section 4.2, we discuss our selection of the contrastive loss objective, which we optimize for this architecture.

4.1 Architecture

MATCH aims to generate a similarity score between task descriptions and corresponding code snippets by creating embeddings using the new architecture. These embeddings are closely aligned for descriptions with successful code snippets while kept distinct for unrelated pairs. This process is illustrated in Figure 2.

In terms of functional correctness, a close (positive) pair (t, c) is defined as a task description t paired with a code snippet c that successfully implements the described task. Additionally, we consider human preferences: code snippets favored by developers are regarded as being closer to their corresponding task descriptions, with higher preference ratings indicating a stronger relationship.

We now provide a detailed explanation of the components of the architecture.

Initial Embeddings Our architecture's first layer consists of two encoders: a text encoder dedicated to processing the task description t and a code encoder focused on the code snippet c. This layer generates initial embeddings for both inputs, which are then forwarded to the next layer aimed at enhancing and aligning both embeddings within a shared space. It is important to note that this layer can be either trained or kept frozen. We offer an analysis of this aspect in Section 7.

Enhanced Embeddings Layer The purpose of this layer is to enrich the initial embeddings of both the task description and the code snippet with relevant information from each other, thereby enhancing their contextual understanding. This design enables the model to focus on the most relevant aspects of each input, facilitating the learning of relationships between the task description and the code snippet. Importantly, this layer 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 consists of two cross-attention components, one for each input, based on the transformer cross-attention module outlined in (Vaswani et al., 2017). The component for the code snippet uses the code as the query and the task description as both the key and value, enriching the code embedding with relevant information from the text. Conversely, the component for the task description uses the text as the query and the code as the key and value, enhancing the text embedding with relevant features from the code. This integration is followed by a linear layer for each component, which projects the embeddings into a shared space where they can be compared and aligned more effectively. This design not only integrates contextual information from both inputs but also supports the Contrastive Learning process by ensuring

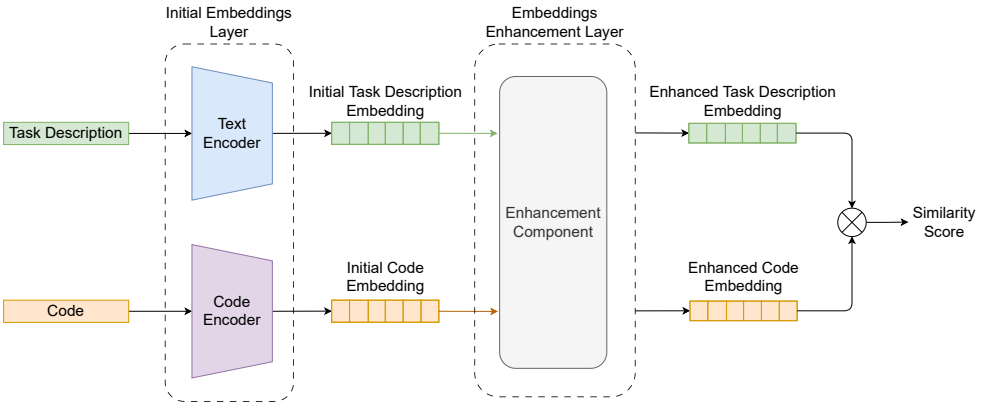


Figure 2: MATCH Architecture: This figure illustrates the general architecture of MATCH, featuring an Enhanced Embeddings Layer that integrates contextual information from task descriptions and code snippets. Specific implementations, including Linear and Cross-Attention enhancements, are shown in Figure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

that related embeddings are closer together in this shared space. A detailed illustration of this approach can be found in Figure 3a.

- **Linear:** This approach employs a separate linear transformation for each embedding, enabling the model to independently refine the features from both the task description and the code snippet. The embeddings are then projected into a shared space, allowing for effective comparison and alignment. Here $\mathbf{e}_t = f_t(t)$, likewise $\mathbf{e}_c = f_c(c)$. This method simplifies the learning process while preserving essential relationships between the inputs, ensured by a *Contrastive Learning* objective as defined in Section 4.2. This approach is illustrated in Figure 3b.

We explore these alternatives further in Section 7.

Similarity Score Calculation: After both the code and text embeddings have been calculated, we compute the similarity between them using *Cosine Similarity*. This measure is widely used for assessing semantic similarity between vectors. *Cosine Similarity* quantifies the cosine of the angle between two embedding vectors, effectively capturing their directional similarity while remaining invariant to their magnitudes.

Formally, given two embedding vectors $\mathbf{e}_\text{t}$ and $\mathbf{e}_\text{c}$ in the embedding space of the a given task description t and a code snippet c, the *Cosine Similarity* is calculated as:

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

Then, composed with the enhanced embedding functions in Equation (1), we reach the goal defined in Section 3:

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

and thus the MATCH score is computed.

4.2 Contrastive Learning Objective

We now describe the objectives our architecture aims to optimize, distinguishing between binary and continuous label scenarios as explained in Section 3.

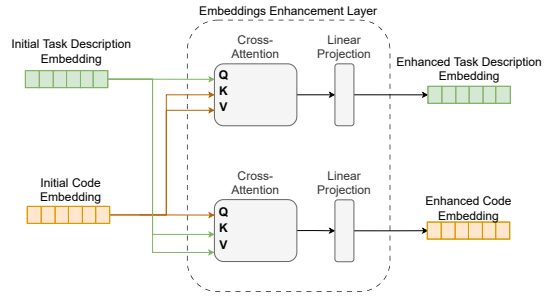
Given a task description $t \in \mathcal{T}$ (in natural language) and a code snippet $c \in \mathcal{C}$, we define the loss functions for both binary and continuous labels.

4.2.1 Binary Labels

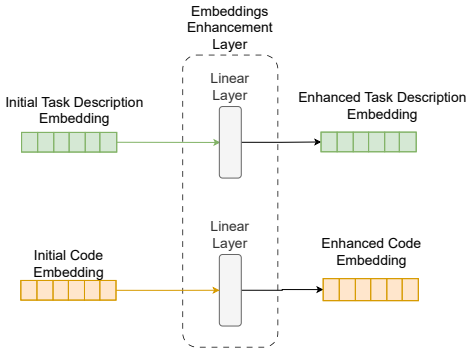
For a binary label $y_{bin} \in \{0, 1\}$ indicating whether the code snippet successfully implements the task, we optimize the following loss function for binary loss $\mathcal{L}_{bin}$:

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

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.



(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.

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 $\mathcal{L}_{cont}$:

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

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

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

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. 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.

In our experiments (see section 6) and analysis (refer to Section 7), we explore various versions of MATCH. We define MATCH as MATCH (CodeEncoder(X), E), where CodeEncoder can be either **Base**, which refers to a base code encoder, or **LS**, which is a language-specific encoder pretrained on a particular programming language. For our experiments, we use the language-specific encoders developed by Zhou et al. (2023). The variable X indicates the status of the code encoder, with T representing a trained encoder and F representing a frozen encoder, meaning it remains unchanged during the training of MATCH. The variable E denotes the type of enhancement layer used, which can be either CA for cross-attention or $Linear$ for linear enhancement.

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 are in accordance with best practices in natural language evaluation and the experiments conducted in (Zhou et al., 2023) and (Zhuo, 2023), allowing us to assess the correlation between the scores of each metric and the reference values.

6 Experiments

In this section, we examine the utility of MATCH by conducting experiments to assess the correlations between the scores generated by MATCH and both functional correctness and human preferences, as detailed in Section 6.1 and Appendix A.3, respectively.

6.1 Functional Correctness

Dataset To evaluate alignment with functional correctness, we use the HumanEval dataset (Chen et al., 2021), which includes natural language descriptions of programming tasks, hand-crafted input-output test cases, and reference solutions cre-

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 5

ated by humans. Originally developed for Python, HumanEval has been translated into 18 programming languages by (Cassano et al., 2022), including predictions generated by the Codex model (code-davinci-002) and their associated functional correctness metrics. In our experiments, we focus on four widely-used programming languages: Java, C++, Python, and JavaScript, as highlighted in (Zhou et al., 2023).

It is important to note that the labels in this dataset are binary, which is why we will use $\mathcal{L}_{\text{bin}}$, as defined in Section 4.2.1,as our optimization objective. For additional information about the datasets and their attributes please refer to Appendix A.1.

Results Results for correlation to functional correctness are reported in Table 1. We report results for 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)** that trained a language-specific code encoders introduced by Zhou et al. (2023). Our findings show that MATCH achieves the highest or comparable Kendall-Tau correlation with functional correctness across all four programming languages, although ICE-Score performs slightly better for JavaScript in Kendall-tau Correlation. Additionally, MATCH outperforms all baseline methods in both Spearman and Pearson correlation across all languages. Overall, MATCH demonstrates a stronger correlation with functional correctness than all baseline methods.

6.2 Human Preference

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

Results Results for correlation with human preferences are reported in Table 3. We observe that **MATCH (Base(T), CA)** is comparable to CodeBERTScore in Kendall-Tau correlation with human preferences. Additionally, **MATCH (Base(T), CA)** outperforms all baselines in both Spearman and Pearson correlations, indicating that MATCH exhibits a stronger correlation with human preferences overall.

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 2. We address the following aspects:

Impact of Language-Specific Code Encoders vs. a Base Encoder Table 2 shows that using a language-specific code encoder, when available, yields improved correlations with functional correctness. However, a base code encoder tends to

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 2: 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.

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 6

achieve better correlations with human preference.

Impact of Training the Code Encoder vs. Freezing It Table 2 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 2 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 have presented MATCH, a framework for evaluating the semantic similarity between blocks of code and their corresponding text descriptions, with a specific focus on evaluating generated code, for example by LLMs. Our approach independently encodes code and text, generating enriched embeddings through a process of information mixing between the encoded representations. A cosine similarity score is then computed to quantify the similarity. MATCH offers flexibility with respect to the chosen encoders and the architecture of the information mixing block; we evaluated and compared several architectural variants. Our results demonstrate that MATCH outperforms existing code evaluation metric baselines, as evidenced by improvements in Kendall Tau, Spearman's, and Pearson's correlation metrics.

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.

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/.

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.

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

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.

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*.

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

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 $\in$ {python, java, javascript, 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 4 we summarize the used existing packages for each of the baselines metrics we compare MATCH to in Section 6.

[1] https://platform.openai.com/docs/models/gpt-3-5
[2] https://platform.openai.com/docs/models/gpt-4o

These packages are all licensed under either the MIT or Apache-2.0 license.

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 StackOverflow. 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 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 2 are reported in Table 7.

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

Table 4: 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	.017	.006	.012	.036	.010	.015	.044	.047	.039	.019	.023	.015
CodeBLEU	.014	.011	.016	.027	.011	.018	.052	.024	.031	.012	.008	.006
ROUGE-1	.009	.007	.005	.044	.019	.017	.057	.029	.030	.014	.011	.010
ROUGE-2	.014	.010	.009	.035	.014	.015	.043	.041	.040	.018	.009	.011
ROUGE-L	.015	.006	.006	.037	.021	.018	.069	.033	.036	.016	.013	.013
METEOR	.022	.007	.011	.036	.013	.015	.048	.040	.040	.030	.014	.010
chrF	.009	.014	.012	.024	.012	.016	.031	.032	.030	.014	.010	.001
CodeBERTScore	.007	.009	.008	.028	.015	.012	.066	.025	.017	.017	.013	.010
ICE-Score	.021	.013	.012	.015	.013	.013	.072	.033	.031	.012	.015	.015
MATCH (Base(T), CA)	.046	.012	.017	.026	.013	.016	.074	.041	.047	.081	.027	.015
MATCH (LS(F), Linear)	.025	.013	.017	.029	.011	.015	.107	.021	.021	.015	.013	.012

Table 5: 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	.061	.010	.022
CodeBLEU	.039	.034	.03
ROUGE-1	.049	.036	.03
ROUGE-2	.063	.027	.03
ROUGE-L	.06	.036	.027
METEOR	.053	.009	.033
chrF	.066	.024	.028
CodeBERTScore	.074	.033	.033
ICE-Score	.060	.036	.027
MATCH (Base(T),CA)	.065	.027	.021

Table 6: 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.

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)	.025	.013	.017	.029	.011	.015	.107	.021	.021	.015	.013	.012	.059	.016	.015
MATCH (LS(F),CA)	.020	.006	.021	.028	.010	.016	.099	.020	.032	.032	.009	.007	.104	.014	.022
MATCH (Base(T),CA)	.046	.012	.017	.026	.013	.016	.074	.041	.047	.081	.027	.015	.065	.027	.021
MATCH (Base(T),Linear)	.119	.131	.193	.042	.022	.035	.118	.030	.024	.063	.017	.030	.063	.024	.025

Table 7: 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 2.