ODASim: Ordered, Distinctive and Absolute Semantic Similarity for Code Explanation Evaluation

Anonymous ACL submission

Abstract

Code explanations are increasingly generated by large language models and used in software engineering workflows, making reliable evaluation essential. However, existing model-based and embedding-based methods often fail to distinguish correct explanations from partially or fully incorrect ones, and their similarity scores are poorly calibrated and do not reflect meaningful differences in explanation quality. To address this, we propose ODASim(**O**rderly, **D**istinctive, and **A**bsolute **Sim**ilarity), a model-agnostic graded fine-tuning framework for embedding models that learns calibrated similarity representations between code and explanations. To support fine-grained supervision and evaluation, we also introduce ODA-X, a novel benchmark for code-to-explanation quality grading, comprising code–explanation pairs graded similarity labels derived from strategic perturbations of gold explanations. We apply our ODASim approach to multiple embedding models and evaluate it on two benchmarks: widely popular CodeXGLUE and our proposed benchmark ODA-X, spanning four programming languages - Python, Java, JavaScript, and Go. Results show that our method achieves up to 35% improvement in F1 score and 85% reduction in Expected Calibration Error (ECE), enabling reliable evaluation of code to explanation quality.

1 Introduction

Code explanation aims to generate natural language descriptions that accurately capture the functionality of code snippets (Cui et al., 2022). Whether written by humans or automatically generated by state-of-the-art large language models (LLMs), the quality of such explanations varies widely-ranging from precise and informative to incomplete or even misleading (Shi et al., 2022; Zhang, 2024; Kang et al., 2024). Detecting and quantifying these variations is non-trivial. Early work explored rule-based techniques to identify inconsistencies between code and comments (Tan et al., 2007, 2012), but these methods yield binary outputs and focus only on narrow classes of mismatches. Similarly, traditional word-overlap metrics such as BLEU, ROUGE, and METEOR fail to reliably capture semantic equivalence when explanations use different phrasing or abstraction levels (Gros et al., 2020; Roy et al., 2021). While neural embedding models better encode semantic similarity, their raw similarity scores are poorly calibrated: they often fail to distinguish subtle errors or partially correct explanations, assigning deceptively high or low scores. Compounding this challenge, human evaluation does not scale-when we provided just 34 explanations to expert reviewers, the average turnaround time was approximately 8 hours, underscoring the high cost and impracticality of manual assessment at scale.

These limitations point to the need for embedding-based similarity models that go beyond coarse semantic matching. In particular, an effective code–explanation similarity model should satisfy three key properties: (1) **Orderliness**-the ability to correctly rank explanations by semantic relevance and correctness; (2) **Distinctiveness**-the ability to separate semantically similar, partially correct, and incorrect explanations; and (3) **Absoluteness**-the ability to produce calibrated similarity scores that reflect true semantic alignment rather than relative proximity alone. Recent work (Virk et al., 2024) explores the use of decoder-only LLMs, with prompting or fine-tuning, to better align model judgments with human preferences. However, even state-of-the-art models such as **Granite-embedding-english-r2** (Figure 3 (A) and **Llama-3.3-70B-Instruct** (Figure 3 (C)) exhibit significant confusion in the medium-quality range, frequently overestimating or underestimating explanations that are only partially correct.

To concretely illustrate why such distinctions are essential, Figure 1 presents a Python counter imple-

mentation (C) along with three natural language explanations exhibiting varying degrees of semantic alignment, denoted by $sim()$. Explanation *E1* accurately captures all key entities in C-the constructor ($__init__$), the modifying methods (increment and decrement), and the getter (get_value)-and therefore should receive a similarity score close to 1. In contrast, *E2* partially aligns with the code but misrepresents core entities by incorrectly identifying the modifying methods as ($__init__$) and get_value, and by misclassifying get_value itself as a modifying function; consequently, its similarity score should be lower. Finally, *E3* is generic and unrelated to the code, warranting a similarity score near 0. The expected ordering $sim(C, E1) > sim(C, E2) > sim(C, E3)$ highlights the necessity of entity-level understanding and calibrated similarity scoring-particularly for reliably distinguishing medium-quality explanations, where most existing methods fail. However, the out-of-the-box *granite-embedding-english-r2* model assigns scores of 0.9644, 0.9642, and 0.7959 to *E1*, *E2*, and *E3*, respectively, failing to distinguish a partially incorrect explanation from a fully correct one and overestimating an unrelated explanation. This highlights poor discrimination and score calibration in medium-quality cases.

To address these limitations, we introduce ODASim, a model-agnostic framework for calibrated similarity scoring, capable of robustly evaluating code explanations across different embedding model. ODASim employs a graded fine-tuning strategy that produces absolute similarity scores, distinguishing gold explanations from partially correct or unrelated ones. Extensive experiments show that ODASim satisfies Orderliness, Distinctiveness, and Absoluteness across four programming languages: Python, Java, JavaScript, and Go. Our contributions are summarized as follows:

- **ODASim Framework** – Developed ODASim, a model-agnostic framework for calibrated similarity scoring. Validated across *ModernBERT-large, bge-code-v1, granite-embedding-english-r2, and Qwen3-Embedding-0.6B*, demonstrating robustness across diverse embedding architectures.
- **Enterprise-Ready Dataset** – Introduced ODA-X, a multiple programming language dataset with graded similarity scores.
- **Graded Fine-Tuning Pipeline** – Designed a self-supervised fine-tuning process with targeted code–explanation perturbations.
- **Gap Analysis of Current Solutions** – Systematically evaluated leading models and LLM-as-a-Judge(LAAJ) setup, uncovering consistent issues: missing partial correctness, poor calibration, and reliance on syntax over semantics.

The remainder of this paper is organized as follows. Section 2 reviews prior research most relevant to our study. Section 3 describes the dataset design, construction, and preprocessing steps. Section 4 presents our proposed approach, including the problem formulation, and training objectives. Section 5 outlines the evaluation setup, metrics, and baseline systems used for comparison. Section 6 provides a deeper examination of model behaviors and insights drawn from experimental findings. Section 7 summarizes the key takeaways and broader implications of our work while section 8 highlights the limitations of our work.

2 Related Work

Research on semantic correctness in code summarization spans multiple fronts, from language model design to evaluation methodology.

Representation learning for code has evolved from structure-aware models like CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2020), and CodeT5 (Wang et al., 2021) to recent long-context encoders like ModernBERT (Warner et al., 2024) and Jina Embeddings (Günther et al., 2023). While newer models handle longer input sequences, they often require task-specific adaptation to learn meaningful code-summary alignments.

Synthetic data generation has been proposed to enrich training datasets and improve generalization. Examples include CDA-CS (Song et al., 2023), which applies synonym substitution to comments, and IRGen (Li et al., 2022b), which rewrites intermediate representations to produce diverse but semantically consistent code snippets. TransformCode (Xian et al., 2024) further explores program-preserving rewrites. While these augmentations are helpful for improving model robustness, they can introduce noise without careful filtering.

Contrastive learning has been widely used to improve code–text alignment. Prior work explores hard negative mining through adversarial reranking (Li et al., 2022a), dynamic masking with momentum encoders (Shi et al., 2023), and semantic-

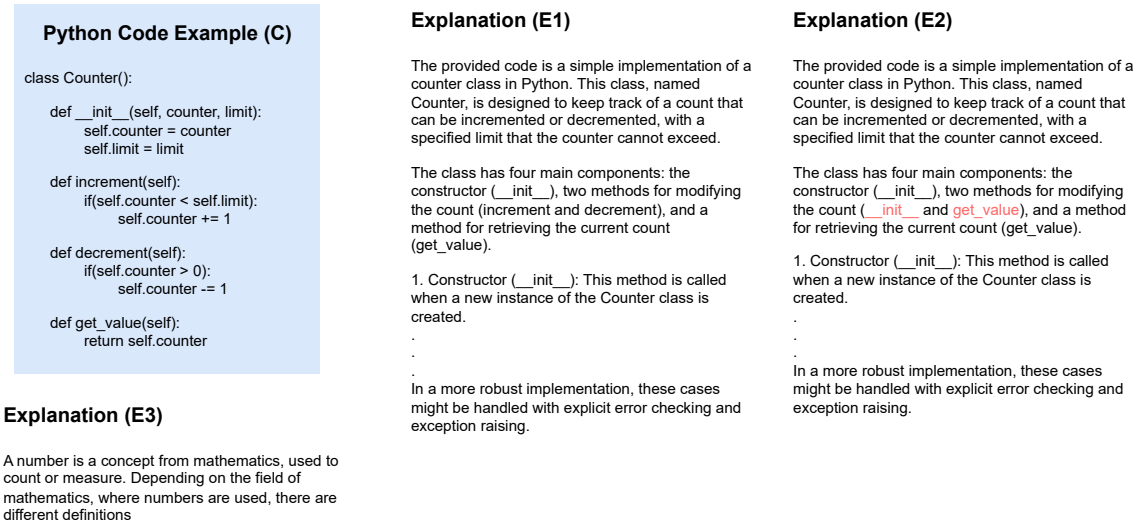

Figure 1: Python example motivating graded and calibrated code-explanation similarity

preserving code transformations. While effective for retrieval, these approaches primarily operate on code or code–text pairs and overlook soft negatives from explanations-such as hallucinated or partially correct summaries-which are crucial for evaluating explanation quality.

Evaluation metrics and calibration remain challenging. Word-overlap metrics (BLEU, ROUGE, METEOR) fail to capture semantic correctness, while alignment-based approaches such as SIDE (Mastropaolo et al., 2024) and calibration methods (Virk et al., 2024) still struggle to produce reliable absolute scores. In contrast, we leverage partial-order supervision and introduce ODASim, a lightweight, model-agnostic framework for calibrated similarity scoring.

3 Data

3.1 Desired Properties of Code-Explanation Pairs

Most existing code–text datasets, such as CodeSearchNet (Husain et al., 2019) and CodeXGLUE (Lu et al., 2021), pair code with docstrings that primarily summarize high-level purpose and interface-level interactions. These explanations are generally short and often omit detailed internal logic or algorithmic reasoning. In contrast, high-quality code explanations should address the "*why*" and "*how*" of code functionality, highlighting the need for datasets with varied-length code snippets and more verbose, detailed explanations.

3.2 Code-Explanation Pairs Preparation

To train our long-context embedding model for code–explanation semantic similarity, we constructed a large-scale synthetic dataset of code–explanation pairs. Due to the limited scale and simplicity of existing human-annotated datasets, we used Mixtral-8x22B-Instruct-v0.1[1] to generate explanations across Java, Python, JavaScript, and Go. Mixtral was selected for its strong reasoning quality, long-context capability, and ability to produce coherent explanations for large functions and classes. Its open-source availability also enables scalable, cost-effective offline generation, making it well suited for large-scale data construction.

We construct our dataset from two sources:

- *CodePile*[2]: A corpus of diverse code snippets across multiple languages and domains. High-quality explanations are generated using the following process:
  - Uniformly sample code snippets up to 8k tokens from repositories.
  - Generate natural language explanations via LLM prompting.
  - Following Maharaj et al. (2024), low-quality code–explanation pairs are filtered using language-specific parsers (e.g., Javalang) to extract code entities and prompt-based methods to identify

[1] https://huggingface.co/mistralai/Mixtral-8x22B-Instruct-v0.1

[2] https://github.com/ibm-granite/granite-3.0-language-models/blob/main/paper.pdf

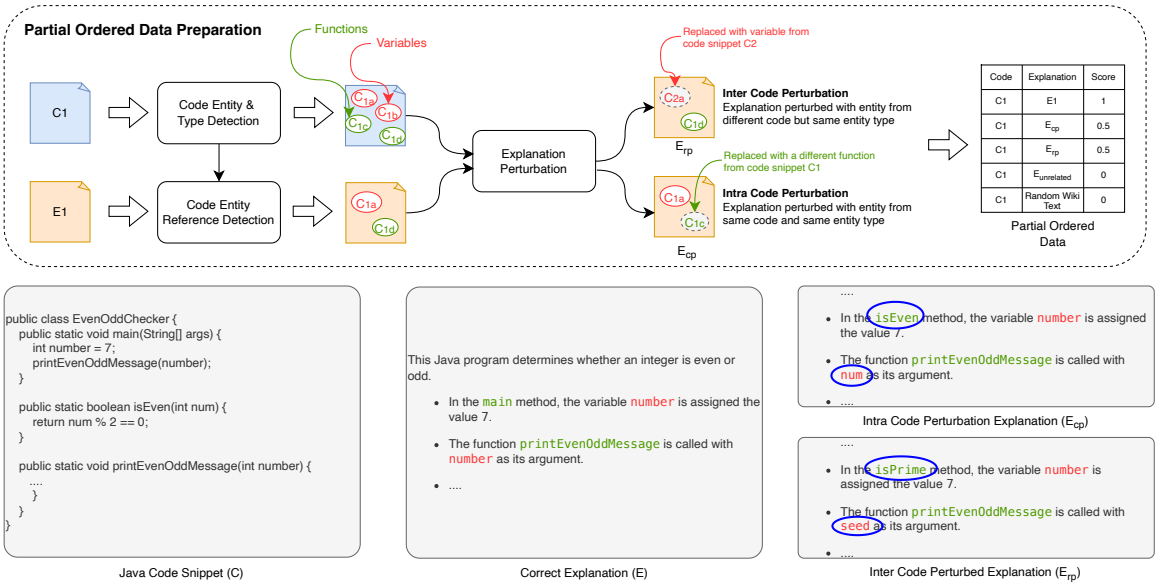

Figure 2: Overview of the Partial Ordered Data Preparation Strategy, illustrating how code entity detection and explanation perturbation (intra-code and inter-code) are used to create partially ordered data for robust code-explanation alignment.

entities in the generated summary. Aligning these entities and checking semantic consistency ensures the factual correctness of explanations. Using this filtering process, approximately 33% of the data were discarded.

- *CodeXGLUE*: We use the *code-to-text* train subset with human-written docstrings. Combined with LLM-generated explanations from CodePile, this mix of human and machine content improves model robustness.

The code–explanation pairs were selected to ensure balanced coverage across context lengths, reflect real-world code complexity, span multiple languages, and enable learning of stable semantic representations for small, medium, and large contexts. Following these criteria, we created **ODA-X**, covering Java, Python, JavaScript, and Go, with snippet lengths up to 8k tokens. More details about the dataset can be found in Table 3. By combining concise human-written docstrings with verbose LLM-generated explanations, ODA-X captures the diversity and documentation patterns of real-world enterprise code.

3.3 Partial Order Data Preparation

Building on the positive examples generated in the previous step, we now introduce a targeted data-perturbation strategy to generate negative samples for model training.

3.3.1 Perturbation Strategy

We implemented a targeted data perturbation pipeline grounded in code entity manipulation. The design is motivated by ETF's entity-centric tracing framework (Maharaj et al., 2024). Our approach follows the principle of tracing entities between code and explanations. This enables controlled semantic variation at the entity level. We use static code analysis to extract semantically meaningful elements from both source code and natural language explanations. This extraction supports controlled and interpretable perturbations. In explanations, function names, variable names, and other code constructs are identified using backtick-based (`) markers. For source-code parsing, we employ language-specific tools. We use *Javalang* for Java, Python's built-in *ast* module, and Tree-sitter[3] for JavaScript and Go. This ensures accurate extraction of syntactic and semantic entities across languages.

From the parsed code, we extract entities including *variable names, method names, exceptions, class names, data types, data structures, libraries*, and *function calls*, which form the basis of our perturbation method.

We define two types of explanation perturbations:

- **Intra Code Perturbation**: In this approach, an entity in the explanation is replaced with

[3] https://tree-sitter.github.io/tree-sitter/

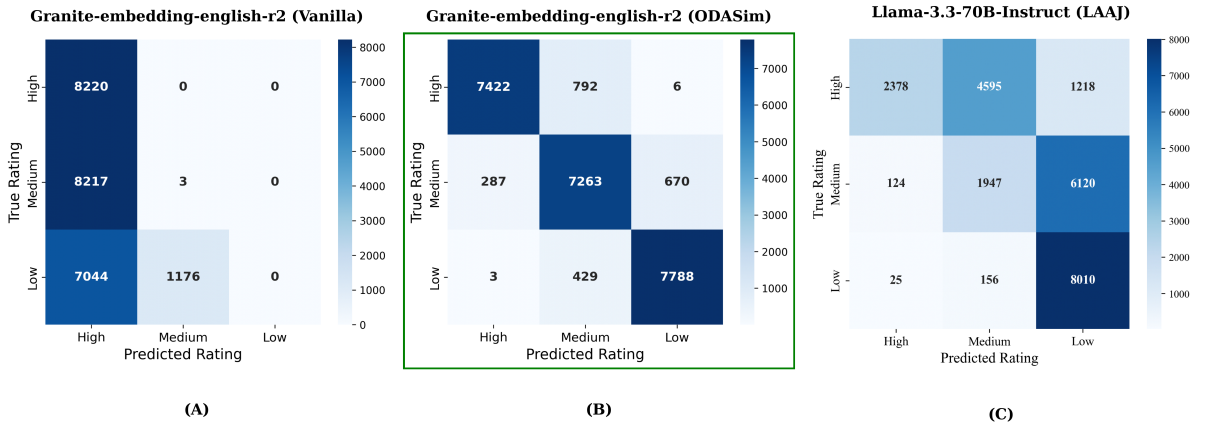

Figure 3: Confusion matrices comparing granite-embedding-english-r2 variants and an LLM baseline, demonstrating that ODASim training substantially improves classification accuracy across High, Medium, and Low ratings over both the vanilla model and Llama-3.3-70B-Instruct (LAAJ).

another entity of the same type from the same code snippet. For instance, as shown in Figure 2, the code snippet has functions *main()*, *isEven()*, and *printEvenOddMessage()*, and we replace the mention of *main()* in explanation E with *isEven* to derive E_{cp}. These perturbations simulate partial correctness or confusion between closely related code components.

- **Inter Code Perturbation**: In this approach, an entity is replaced with another entity of the same type, but sampled from a different code snippet. For example, *isEven()* in E of Figure 2 is replaced with *isPrime()*, a function defined in another code snippet and not present in the current code snippet C, to derive E_{rp}. This more aggressive perturbation often leads to unrelated explanations and serves as a strong negative signal during training.

The partial-order perturbation strategy-based on entity-level static analysis-aligns with the kind of semantic inconsistencies that frequently appear in documentation, such as mismatched function names, incorrect variable references, or outdated logical descriptions.

3.3.2 Partial Ordering

We introduce graded similarity signals, pairing each code snippet with explanation variants scored by semantic accuracy.

The gold explanation-written to accurately and fully describe the behavior of the code, is assigned a score of 1.0. All perturbed explanations are assigned a score of 0.5 to reflect their partial correctness. To introduce varying degrees of semantic distortion, half of the perturbed samples have 25% entities of the explanation replaced, while the other half have 50% entities replaced. These perturbations are applied using the two strategies discussed in Section 3.3.1. Finally, to create the data points for the similarity score of 0.0, unrelated explanations or random Wikipedia texts are paired with code snippets. Figure 2 provides an overview of the partially ordered data preparation strategy along with the scores provided to different code-explanation pairs.

4 Methodology and Training

4.1 Problem Formulation

Let C denote a code snippet and E its natural language explanation. We learn a representation function $f(\cdot)$ that embeds both code and text into a shared semantic space, such that $f(C)$ and $f(E)$ are close only when the explanation is accurate and relevant. This is formulated as a semantic similarity task, where the model assigns high similarity to true explanations and lower similarity to perturbed or unrelated ones.

We adopt flexible training pipeline that can optimize various architectures (e.g., *ModernBERT-large*, *Qwen3-Embedding-0.6B*, *bge-code-v1*) on code–explanation pairs for our partial-order training strategy.

Given a code–explanation pair (C, E), we define $\mathbf{h}_C = f(C)$ and $\mathbf{h}_E = f(E)$, where $\mathbf{h}_C, \mathbf{h}_E \in \mathbb{R}^d$ are model-specific sequence representations. The semantic similarity is $\mathrm{sim}(C, E) = \frac{\langle \mathbf{h}_C, \mathbf{h}_E \rangle}{\|\mathbf{h}_C\| \cdot \|\mathbf{h}_E\|}$. We train by minimizing the mean squared error between the predicted similarity and a supervision

signal $y \in \{1.0, 0.5, 0.0\}$.

The training objective is thus:

$$\mathcal{L}(C, E) = (\text{sim}(C, E) - y)^2 \quad (1)$$

Rationale for Cosine Similarity Loss. We use cosine similarity regression instead of contrastive or classification losses for three reasons:

- **Graded Supervision.** It supports fine-grained labels beyond binary supervision, aligning with our three-tier scoring scheme.
- **Semantic Ordering.** The loss encourages ranking explanations by their explanatory quality:: true (1.0), partial (0.5), and unrelated (0.0).
- **Interpretability via Absoluteness.** Cosine regression provides absolute similarity scores, enabling interpretation of explanation quality.

This formulation enables fine-grained differentiation of explanation quality while preserving a streamlined and stable training objective. Implementation details are provided in Appendix A.3.

5 Benchmarks, Baselines and Evaluations

5.1 Benchmarks

We evaluate our model on two datasets: *CodeXGLUE* and *ODA-X*.

CodeXGLUE: We use the *code-to-text* test set, with original docstrings as ground-truth explanations. Docstrings are systematically perturbed as described earlier. Snippets that could not be parsed reliably were removed from the test set.

ODA-X: Additional code snippets are sampled from *CodePile* (with no training overlap) and processed using the same pipeline to generate benchmark code–explanation pairs.

5.2 Baselines

We evaluate several models as baselines for comparison, including *CodeT5, CodeSage-large-v2, ModernBERT-large, SFR-2B_R, Jina-v2-base-code, Qwen3-Embedding-0.6B, granite-embedding-english-r2, and bge-code-v1*.

5.3 Evaluation Metrics

We evaluate models on three criteria: **orderliness**, **distinctiveness**, and **absoluteness**, mapping each to standard metrics.

1. Orderliness: Measured using **nDCG@3** to assess the relative ranking of three explanations (gold, perturbed, unrelated) per code snippet.

2. Distinctiveness: Measures how well the model separates gold, perturbed, and unrelated explanations. We bucket similarity scores into High (0.7–1.0), Medium (0.3–0.7), and Low (0.0–0.3) categories and compute **Precision, Recall, and F1-score**. Higher F1 indicates stronger separation between explanation types, particularly for medium-quality cases.

3. Absoluteness: Evaluates the calibration of absolute similarity scores. We use **Expected Calibration Error (ECE)** (Guo et al., 2017), which computes the expected absolute difference between predicted similarity scores and the gold similarity values defined by bucket mid-points. Lower ECE indicates better calibration.

6 Analysis

We evaluated **ODASim** and multiple baselines across four programming languages-Java, JavaScript, Go, and Python-on two benchmarks:**CodeXGLUE** and the curated **ODA-X**.

As shown in Table 1, ODASim-trained models consistently achieve high F1 scores on ODA-X, while non-ODASim baselines often fall below 0.80 and, in extreme cases, below 0.25 (e.g., *ModernBERT-large* on JavaScript at 0.231). ODASim also delivers strong calibration, with ECE values reduced to ≤ 0.041, compared to weaker baselines that frequently exceed 0.25. Similar performance trends are observed on the CodeXGLUE benchmark.

Table 2 reports results for the LAAJ setup using *Llama-3-70B-Instruct*. While LAAJ achieves competitive ranking performance (nDCG@3 = 0.961–0.997), its classification accuracy is substantially lower, with F1 scores ranging from 0.394 (Python, CodeXGLUE) to 0.670 (Python, ODA-X). Calibration is also weaker, with ECE values between 0.136 and 0.318. In contrast, ODASim attains higher F1, significantly lower ECE, and does so with far lower computational cost.

As a part of result analysis, we answer the below Research Questions(RQs) that are relevant to this work.

RQ1: How well does ODASim generalize to diverse codebases and explanation styles in production? ODASim generalizes well across languages and benchmarks, consistently outperforming baselines (Table 1). On CodeXGLUE, ODASim-trained models achieve F1 scores of 0.81–0.98, yielding 8–35% improvements over

Language	Model	CodeXGLUE					ODA-X				
		P ↑	**R ↑**	**F1 ↑**	**nDCG@3 ↑**	**ECE ↓**	**P ↑**	**R ↑**	**F1 ↑**	**nDCG@3 ↑**	**ECE ↓**
Java	CodeT5	0.387	0.373	0.302	0.865	0.175	0.586	0.340	0.260	0.896	0.149
	Codesage-large-v2	0.687	0.635	0.620	0.996	0.176	0.762	0.764	0.759	0.976	0.042
	SFR-2B_R	0.686	0.512	0.418	0.998	0.192	0.178	0.343	0.233	0.978	0.206
	Jina-v2-base-code	0.762	0.738	0.739	0.994	0.108	0.707	0.686	0.667	0.969	0.081
	granite-embedding-english-r2	0.118	0.333	0.173	0.995	0.329	0.115	0.333	0.171	0.968	0.356
	ModernBERT-large	0.208	0.333	0.169	0.834	0.340	0.299	0.333	0.227	0.856	0.230
	Qwen3-Embedding-0.6B	0.876	0.868	0.870	0.999	0.108	0.710	0.689	0.687	0.974	0.082
	bge-code-v1	0.249	0.407	0.293	0.971	0.258	0.471	0.618	0.516	0.977	0.264
	granite-embedding-english-r2*	0.913	0.911	0.912	0.999	0.066	0.939	0.932	0.932	0.997	0.021
	ModernBERT-large*	0.942	0.941	0.941	0.998	0.037	0.976	0.976	0.976	0.999	0.013
	Qwen3-Embedding-0.6B*	0.966	0.965	0.965	0.999	0.041	0.987	0.987	0.987	0.999	0.009
	bge-code-v1*	0.964	0.963	0.963	0.999	0.040	0.988	0.988	0.988	0.999	0.009
JavaScript	CodeT5	0.324	0.340	0.277	0.839	0.180	0.567	0.343	0.296	0.873	0.174
	Codesage-large-v2	0.661	0.662	0.646	0.979	0.125	0.735	0.742	0.738	0.968	0.054
	SFR-2B_R	0.609	0.420	0.334	0.986	0.155	0.515	0.343	0.237	0.963	0.197
	Jina-v2-base-code	0.731	0.724	0.723	0.985	0.070	0.681	0.661	0.651	0.953	0.075
	granite-embedding-english-r2	0.119	0.333	0.175	0.975	0.325	0.117	0.333	0.173	0.950	0.349
	ModernBERT-large	0.260	0.331	0.176	0.844	0.320	0.196	0.328	0.231	0.830	0.233
	Qwen3-Embedding-0.6B	0.721	0.694	0.696	0.978	0.074	0.671	0.632	0.644	0.957	0.070
	bge-code-v1	0.287	0.440	0.330	0.947	0.253	0.365	0.546	0.437	0.962	0.202
	granite-embedding-english-r2*	0.815	0.813	0.814	0.988	0.056	0.942	0.941	0.940	0.998	0.038
	ModernBERT-large*	0.971	0.971	0.971	0.998	0.016	0.899	0.896	0.897	0.992	0.035
	Qwen3-Embedding-0.6B*	0.930	0.930	0.930	0.996	0.039	0.983	0.983	0.983	0.999	0.016
	bge-code-v1*	0.932	0.931	0.932	0.997	0.039	0.985	0.985	0.984	0.999	0.015
Go	CodeT5	0.358	0.344	0.249	0.791	0.110	0.236	0.331	0.268	0.884	0.175
	Codesage-large-v2	0.715	0.670	0.657	0.996	0.156	0.717	0.722	0.719	0.966	0.055
	SFR-2B_R	0.655	0.478	0.386	0.998	0.181	0.506	0.337	0.228	0.959	0.202
	Jina-v2-base-code	0.749	0.735	0.726	0.994	0.115	0.630	0.623	0.589	0.955	0.098
	granite-embedding-english-r2	0.119	0.333	0.175	0.997	0.327	0.116	0.333	0.172	0.954	0.362
	ModernBERT-large	0.316	0.321	0.210	0.851	0.378	0.324	0.330	0.257	0.844	0.205
	Qwen3-Embedding-0.6B	0.931	0.931	0.931	0.999	0.084	0.654	0.639	0.625	0.960	0.076
	bge-code-v1	0.252	0.412	0.296	0.982	0.261	0.323	0.490	0.382	0.939	0.232
	granite-embedding-english-r2*	0.946	0.946	0.945	0.999	0.046	0.937	0.931	0.931	0.996	0.030
	ModernBERT-large*	0.967	0.967	0.967	0.999	0.026	0.981	0.981	0.981	0.999	0.014
	Qwen3-Embedding-0.6B*	0.980	0.980	0.980	0.999	0.024	0.988	0.988	0.988	0.999	0.013
	bge-code-v1*	0.981	0.981	0.981	1.000	0.025	0.990	0.990	0.990	0.999	0.011
Python	CodeT5	0.374	0.340	0.266	0.882	0.238	0.219	0.336	0.265	0.887	0.204
	Codesage-large-v2	0.760	0.766	0.750	0.995	0.056	0.735	0.739	0.737	0.970	0.059
	Jina-v2-base-code	0.812	0.808	0.802	0.995	0.043	0.638	0.619	0.597	0.960	0.096
	SFR-2B_R	0.588	0.411	0.314	0.998	0.197	0.507	0.338	0.229	0.973	0.205
	granite-embedding-english-r2	0.117	0.333	0.173	0.997	0.363	0.117	0.333	0.173	0.968	0.358
	ModernBERT-large	0.205	0.330	0.178	0.835	0.310	0.325	0.331	0.257	0.855	0.214
	Qwen3-Embedding-0.6B	0.728	0.701	0.690	0.996	0.111	0.678	0.657	0.656	0.963	0.062
	bge-code-v1	0.491	0.440	0.333	0.943	0.259	0.273	0.384	0.259	0.931	0.284
	granite-embedding-english-r2*	0.932	0.925	0.923	0.999	0.025	0.951	0.947	0.947	0.998	0.040
	ModernBERT-large*	0.969	0.968	0.968	1.000	0.006	0.981	0.981	0.981	0.999	0.015
	Qwen3-Embedding-0.6B*	0.983	0.982	0.982	1.000	0.010	0.990	0.990	0.990	1.000	0.013
	bge-code-v1*	0.982	0.982	0.982	1.000	0.011	0.991	0.991	0.991	1.000	0.013

Table 1: Comparison of various code models across four programming languages on two evaluation benchmarks: CodeXGLUE and ODA-X. ODASim-trained models (highlighted in green) consistently achieve the highest scores across languages and benchmarks, demonstrating superior capability in code understanding and explanation evaluation. Rows marked with * indicate ODASim-trained models.

strong baselines, while on ODA-X they reach 0.93–0.99, corresponding to 25–40% gains. ODASim also maintains high ranking quality ($\text{nDCG@3} \geq 0.97$) and substantially improves calibration, reducing ECE by 40–85%, indicating robust generalization across codebases and explanation styles.

RQ2: How does ODASim's scoring reveal hidden weaknesses in otherwise strong LLMs? ODASim reveals hidden weaknesses in otherwise strong code LLMs by explicitly exposing misalignment between ranking quality and confidence calibration. For *granite-embedding-english-r2*, ODASim training improves F1 from below 0.2 to above 0.9 (a 4–5× increase), while reducing calibration error by over 80% across both benchmarks. A similar pattern is observed for *bge-code-v1*, where ODASim raises F1 from roughly 0.3 to above 0.95 and lowers ECE by approximately 85–90%. These results show that models with seemingly reasonable ranking behavior can remain severely miscalibrated, a limitation that ODASim both reveals and

Language	CodeXGLUE					ODA-X				
	P ↑	**R ↑**	**F1 ↑**	**nDCG@3 ↑**	**ECE ↓**	**P ↑**	**R ↑**	**F1 ↑**	**nDCG@3 ↑**	**ECE ↓**
Java	0.585	0.502	0.462	0.997	0.256	0.714	0.685	0.631	0.983	0.167
Javascript	0.566	0.544	0.511	0.992	0.241	0.759	0.702	0.652	0.993	0.157
Go	0.609	0.619	0.584	0.994	0.192	0.750	0.674	0.617	0.983	0.170
Python	0.502	0.480	0.394	0.961	0.318	0.768	0.714	0.670	0.990	0.136

Table 2: Evaluation of LAAJ (Llama-3-70B-Instruct) on CodeXGLUE and ODA-X datasets across multiple programming languages, highlighting that even large LLMs struggle with ranking, F1, and calibration.

corrects.

RQ3: What are the resource and cost benefits of using ODASim instead of full LLM evaluation for explanation scoring? ODASim avoids the substantial memory and latency overhead of LLM-as-a-judge evaluation. Long-context inference with Llama-3.3-70B-Instruct typically requires multi-GPU sharding, with memory footprints of ≈360 GiB (FP32) or ≈180 GiB (BF16) and per-example latency of several seconds for 8k-context inputs. In contrast, ODASim uses a lightweight encoder with a single forward pass, achieving millisecond-level latency (≈5–15 ms per pair) on a single GPU. This results in a 100×–1000× speedup, enabling large-scale evaluation at a fraction of the computational and infrastructure cost.

RQ4: Can the model maintain ranking, separation, and calibration when explanations vary in style and length? Yes. As shown in Table 1, ODASim-trained models maintain stable performance across concise docstrings (CodeXGLUE) and verbose explanations (ODA-X). Across all four languages, $\mathrm{nDCG@3} \approx 0.99$–$1.00$, indicating preserved ordering despite large differences in explanation length and style. Calibration remains low, with $\mathrm{ECE} \approx 0.04$–$0.07$ on CodeXGLUE and ≈ 0.02–0.04 on ODA-X, while precision, recall, and F1 scores remain clearly separated from baselines, confirming robustness to variation in explanation style and verbosity.

Overall, the results demonstrate that ODASim produces well-calibrated, human-aligned similarity scores, consistently outperforming strong baselines such as *Codesage* and *Jina-v2* across languages and datasets. These findings confirm ODASim's robustness, cross-language generalization, and suitability for large-scale code explanation evaluation. An ablation study further highlights the importance of using the partially ordered dataset, with details provided in Appendix A.2.

6.1 Human Evaluation and Model Alignment

We evaluated ODASim's alignment with human judgment using two expert annotators, each scoring 50 samples (100 total instances). Annotators were instructed to rate each explanation on a five-point scale ranging from 1 to 5. The model showed strong agreement with human ratings: 80% of predictions lie within a ± 0.20 margin of human scores (Annotator 1: 42 cases; Annotator 2: 38 cases). ODASim achieved a mean absolute error (MAE) of 0.116 and a mean squared error (MSE) of 0.0259.

When scores were grouped into low (0–0.3), medium (0.3–0.7), and high (0.7–1.0) buckets, the model consistently matched human qualitative distinctions. These results indicate that ODASim reliably mirrors expert evaluation patterns in both absolute scoring and coarse-grained judgment. Detailed annotation guidelines provided to the human evaluators, including the rating scale and interpretation criteria, are described in Appendix A.4.

7 Conclusion

We present a graded fine-tuning framework that produces calibrated similarity scores for evaluating code explanations. We also propose a new dataset ODA-X, for grading code-explanation pairs in realistic enterprise use-cases. Our model-agnostic approach shows consistent gains across *ModernBERT*, *bge-code-v1*, *granite-embedding-english-r2* and *Qwen3-Embedding-0.6B* models when evaluated on CodeXGLUE and ODA-X across four programming languages. ODASim achieves state-of-the-art performance in classification (F1), ranking (nDCG), and calibration (ECE), outperforming baselines like *Codesage* and *Jina-v2*. It also generalizes well across explanation styles and aligns with human and LLM judgments.

8 Limitations

While ODASim demonstrates strong performance across languages, model architectures, and evalua-

tion settings, several limitations remain. Although ODA-X includes diverse code snippets up to 8k tokens, it does not fully capture the complexity of enterprise codebases that often contain domain-specific languages, proprietary frameworks, legacy systems such as COBOL or ABAP, and very large modules that exceed typical context limits. Similarly, the explanation data-drawn from both human-written docstrings and LLM-generated summaries-reflects common documentation patterns but does not encompass the full range of ambiguity, inconsistency, and outdated comments found in real repositories. Our perturbation strategy, despite being grounded in static analysis and entity manipulation, cannot reproduce every type of semantic drift or conceptual misunderstanding developers introduce in practice. In addition, ODASim's graded supervision currently models correctness using three discrete tiers (1.0, 0.5, 0.0), which may be insufficient for scenarios requiring finer-grained judgments of completeness, logical cohesion, adherence to business rules, or conceptual accuracy. Finally, while ODASim performs well on short to medium-length explanations across four high-resource languages, extending it to multilingual, low-resource, and open-ended settings-such as collaborative programming, educational feedback, or complex narrative explanations-remains an important direction for future work, along with exploring its applicability to related tasks like bug-fix rationale assessment and code translation evaluation.

References

Haotian Cui, Chenglong Wang, Junjie Huang, Jeevana Priya Inala, Todd Mytkowicz, Bo Wang, Jianfeng Gao, and Nan Duan. 2022. Codeexp: Explanatory code document generation. *arXiv preprint arXiv:2211.15395*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and 1 others. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

David Gros, Hariharan Sezhiyan, Prem Devanbu, and Zhou Yu. 2020. Code to comment" translation" data, metrics, baselining & evaluation. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 746–757.

Michael Günther, Louis Milliken, Jonathan Geuter, Georgios Mastrapas, Bo Wang, and Han Xiao. 2023. Jina embeddings: A novel set of high-performance sentence embedding models. *arXiv preprint arXiv:2307.11224*.

Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. 2017. On calibration of modern neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1321–1330. JMLR. org.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, and 1 others. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Sungmin Kang, Louis Milliken, and Shin Yoo. 2024. Identifying inaccurate descriptions in llm-generated code comments via test execution. *arXiv preprint arXiv:2406.14836*.

Xiaonan Li, Yeyun Gong, Yelong Shen, Xipeng Qiu, Hang Zhang, Bolun Yao, Weizhen Qi, Daxin Jiang, Weizhu Chen, and Nan Duan. 2022a. Coderetriever: A large scale contrastive pre-training method for code search. In *Proceedings of the 2022 conference on empirical methods in natural language processing*, pages 2898–2910.

Zongjie Li, Pingchuan Ma, Huaijin Wang, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2022b. Unleashing the power of compiler intermediate representation to enhance neural program embeddings. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2253–2265.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, and 3 others. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

Kishan Maharaj, Vitobha Munigala, Srikanth G Tamilselvam, Prince Kumar, Sayandeep Sen, Palani Kodeswaran, Abhijit Mishra, and Pushpak Bhattacharyya. 2024. Etf: An entity tracing framework for hallucination detection in code summaries. *arXiv preprint arXiv:2410.14748*.

Antonio Mastropaolo, Matteo Ciniselli, Massimiliano Di Penta, and Gabriele Bavota. 2024. Evaluating code summarization techniques: A new metric and an empirical characterization. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

Devjeet Roy, Sarah Fakhoury, and Venera Arnaoudova. 2021. Reassessing automatic evaluation metrics for code summarization tasks. In *Proceedings of the 29th*

ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 1105–1116.

Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. 2022. On the evaluation of neural code summarization. In *Proceedings of the 44th international conference on software engineering*, pages 1597–1608.

Ensheng Shi, Yanlin Wang, Wenchao Gu, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2023. Cocosoda: Effective contrastive learning for code search. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2198–2210. IEEE.

Zixuan Song, Hui Zeng, Xiuwei Shang, Guanxi Li, Hui Li, and Shikai Guo. 2023. An data augmentation method for source code summarization. *Neurocomputing*, 549:126385.

Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /* icomment: Bugs or bad comments?*. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 145–158.

Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T Leavens. 2012. @ tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 260–269. IEEE.

Yuvraj Virk, Premkumar Devanbu, and Toufique Ahmed. 2024. Enhancing trust in llm-generated code summaries with calibrated confidence scores. *arXiv preprint arXiv:2404.19318*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Benjamin Warner, Antoine Chaffin, Benjamin Clavié, Orion Weller, Oskar Hallström, Said Taghadouini, Alexis Gallagher, Raja Biswas, Faisal Ladhak, Tom Aarsen, and 1 others. 2024. Smarter, better, faster, longer: A modern bidirectional encoder for fast, memory efficient, and long context finetuning and inference. *arXiv preprint arXiv:2412.13663*.

Zixiang Xian, Rubing Huang, Dave Towey, Chunrong Fang, and Zhenyu Chen. 2024. Transformcode: a contrastive learning framework for code embedding via subtree transformation. *IEEE Transactions on Software Engineering*.

Yichi Zhang. 2024. Detecting code comment inconsistencies using llm and program analysis. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 683–685.

A Overview of Appendix

The appendix is organized as follows. In Section A.1, we present detailed statistics of the training and test datasets. Section A.2 provides an ablation study highlighting the impact of key design choices. Section A.3 describes implementation details, including model training and configuration. Finally, Section A.5covers the prompts used in our experiments, including the Explanation Generation Prompt (Section A.5.11) and the LAAJ Prompt (Section A.5.2). Additional results, tables, and figures are provided to supplement the main text and illustrate the model's performance across different settings.

A.1 Dataset Statistics

Table 3 summarizes the training data statistics for each of the four programming languages across different context length ranges. Each original code sample contributes three training pairs: a true explanation (score 1.0), a perturbed explanation (score 0.5), and a random explanation (score 0.0). This results in a total of 780,000 training samples ($195{,}000 \times 4$ languages), ensuring a balanced representation of high-quality, partially correct, and unrelated explanations. The dataset is carefully constructed to cover a wide variety of code lengths, from short snippets to long sequences up to 8k tokens, which allows the model to learn long-context dependencies effectively.

Table 4 shows the distribution of the CodeXGLUE test set across different languages. The dataset includes a diverse set of code snippets, ensuring the evaluation covers various programming paradigms and coding styles. This diversity helps measure the model's cross-language generalization and robustness on real-world code.

Additionally, Table 5 presents statistics for the synthetic test set used to evaluate the model's ability to handle artificially generated explanations. Together, the training and test datasets ensure that the model encounters a wide range of code lengths, explanation quality, and programming languages during both training and evaluation. This comprehensive coverage is critical for learning robust semantic representations and achieving high-quality code explanation predictions across different scenarios.

Context Length Range (Tokens)	#Code Snippets	#Code–Exp Pairs	Avg Exp Length
1 – 1k*	10,000	30,000	456.90
1k – 2k*	10,000	30,000	525.01
2k – 4k*	10,000	30,000	562.68
4k – 6.5k*	10,000	30,000	610.33
6.5k – 8k*	10,000	30,000	644.76
CodeXGLUE	15,000	45,000	50.25
Total	65,000	195,000	–

Table 3: Training dataset statistics per language (k=1024). Rows marked with * correspond to the ODA-X dataset.

Language	#Code Snippets	#Code-Exp Pairs
Java	8,220	24,660
JavaScript	1,259	3,777
Go	3,012	9,036
Python	5,351	16,053

Table 4: CodeXGLUE Test Set Statistics

Language	#Code Snippets	#Code-Exp Pairs
Java	2,000	6,000
JavaScript	2,000	6,000
Go	2,000	6,000
Python	2,000	6,000

Table 5: ODA-X Test Set Statistics

Lang	P	R	F1	nDCG@3	ECE
Java	0.773	0.744	0.720	0.992	0.089
JavaScript	0.710	0.705	0.676	0.969	0.056
Python	0.816	0.752	0.704	0.994	0.093
Go	0.701	0.670	0.567	0.979	0.148

Table 6: Binary Score Training Performance of ModernBERT on CodeXGLUE Test set

Lang	P	R	F1	nDCG@3	ECE
Java	0.763	0.699	0.608	0.970	0.128
JavaScript	0.791	0.726	0.667	0.976	0.094
Python	0.782	0.716	0.647	0.979	0.106
GO	0.804	0.728	0.669	0.983	0.100

Table 7: Binary Score Training Performance of ModernBERT on ODA-X dataset

A.2 Ablation Study

To demonstrate the importance of using a partially ordered dataset for training a semantic similarity model, we conducted an ablation study on the ModernBERT architecture by training it with binary-labeled data. Specifically, we removed all code–explanation pairs with an intermediate similarity score of 0.5 and trained the model using only the remaining samples. The performance of this ablated ModernBERT model on the CodeXGLUE and ODA-X benchmarks is reported in Table 6 and Table 7, respectively. When compared against the results in Table 1, a consistent and substantial degradation across all evaluation metrics is observed, highlighting the critical role of partial ordering in the training data. For instance, on the CodeXGLUE Java subset, the ModernBERT model trained with partially ordered data achieves a 23% improvement in F1 score and a 58% reduction in ECE. A similar trend is observed on the ODA-X Java subset, where the partially ordered training setup yields a 38% improvement in F1 and a 90% improvement in ECE over the binary-labeled counterpart.

A.3 Implementation details

The training was performed using PyTorch's distributed training launcher torchrun across 8×A100 80GB GPUs to leverage parallelism and accommodate the increased memory demands of long input sequences. We fine-tuned the model with a maximum sequence length of 8k tokens using BF16 mixed-precision training, gradient checkpointing, and flash-attention kernels to reduce memory footprint and improve throughput.

Key hyperparameters for fine-tuning were as follows:

- Batch size of 2 per device for both training and evaluation.
- Total of 5 training epochs, with checkpoints saved and evaluated at the end of each epoch.
- A learning rate of 8e-5 with a linear warmup schedule over 5% of total training steps.
- Weight decay regularization set to 1e-5.

A.4 Human Evaluation Instructions

To assess the quality of generated code explanations, expert annotators from a sister team were provided with a standardized rating guideline. The annotations were performed by the Java experts, each with over five years of professional experience in Java development and code review. Each explanation was evaluated using a five-point scale,

designed to capture varying levels of semantic correctness, clarity, and alignment with the underlying code.

The rating scale was defined as follows:

- **1 – Strongly Disagree / Very Poor:** The generated explanation performs very poorly with respect to the evaluated aspect and exhibits significant semantic or factual errors.
- **2 – Disagree / Poor:** The explanation demonstrates below-average quality, with noticeable omissions or inaccuracies.
- **3 – Neutral / Average:** The explanation meets basic expectations but does not provide clear or detailed insight into the code's functionality.
- **4 – Agree / Good:** The explanation shows above-average quality, accurately describing the code with minor limitations.
- **5 – Strongly Agree / Excellent:** The explanation excels in quality, providing a clear, accurate, and comprehensive description of the code's logic and behavior.

Annotators were instructed to apply this scale consistently across samples, focusing on semantic alignment between the code and its explanation.

A.5 Prompts

A.5.1 Explanation Generation Prompt

The below prompt in is designed to guide a code assistant in generating high-quality, detailed explanations for a given code snippet. It emphasizes not only describing the purpose and overall goal of the code but also breaking down the logic flow, functional components, and variable interactions. The prompt explicitly requests attention to error handling mechanisms, encouraging a holistic understanding of the code's behavior under normal and exceptional conditions. By stressing clarity and completeness, the prompt ensures that the generated explanation is both comprehensive.

```
You are a helpful code assistant that
    provides detailed and comprehensive
    summaries of code. Given a code
    snippet, explain its purpose, logic
    flow, and how various parts of the
    code interact. Focus on explaining
    the overall goal of the code and how
     each section contributes to
    achieving it. Describe the key
    functions, methods, and variables,
    their roles, and how they interact
    with other components of the code.
    Additionally, analyze the code's
    approach to error handling,
    explaining any exception mechanisms
    and how errors are caught or
    propagated through the system.
    Ensure your summary is thorough,
    covering all important aspects
    without leaving out any critical
    details, while making the
    explanation clear and easy to follow
    .

{{code_snippet}}
```

A.5.2 LAAJ Prompt

This prompt is used for automated evaluation of code explanations by assigning a numerical score (0–10) based on their accuracy and relevance to the corresponding code snippet. It provides clear scoring criteria along with concrete in-context examples that demonstrate how to differentiate between correct, partially correct or perturbed, and unrelated explanations. The prompt instructs the model to first generate a rationale explaining its assessment, followed by a score, thereby promoting transparency and justifiability in the evaluation process. To compute the different metrics the score from 0-10 is than scaled to 0-1.

```
Given a code snippet and an explanation,
     assign a score (from 0 to 10) based
     on how well the explanation
    describes the code. A score of:
- 10: Perfectly explains the code.
- 5: Somewhat related but contains
    errors or hallucinations.
- 0: Completely unrelated.

Examples:

Code:
package hu.bsmart.framework.
    communication.data.taskdatatype;\n\
    nimport java...
Explanation:
The provided code is a Java class named
    QuizTaskData that ...
Rationale:
The explanation accurately covers all
    aspects of the QuizTaskD ....

Code:
package hu.bsmart.framework.
    communication.data.taskdatatype;\n\
    nimport java...
Explanation:
The provided code is a Java class named
    QuizTaskData that ...
Rationale:
The explanation contains multiple
    inaccuracies and hallucinations. It
```

```
    refers ...
Score: 5

Code:
package hu.bsmart.framework.
    communication.data.taskdatatype;\n\
    nimport java...
Explanation:
The provided code is a Java class named
    PokerHandData that implements the
    Comparable ...
Rationale:
The explanation is completely unrelated
    to the code. It describes a
    PokerHandData ...
Score: 0

Now, use the above examples to give
    rationale first and then score the
    following explanation based on the
    given code. The rationale and score
    should in the format following
    format
Rationale:<Rationale about the
    explanation>
Score:<score between 0-10>

Code:
{{code_snippet}}
Explanation:
{{explanation}}
```