

# CRScore: Grounding Automated Evaluation of Code Review Comments in Code Claims and Smells

Atharva Naik Marcus Alenius Daniel Fried Carolyn Rosé

Language Technologies Institute

Carnegie Mellon University

{arnaik, malenius, dfried, cprose}@cs.cmu.edu

## Abstract

The task of automated code review has recently gained a lot of attention from the machine learning community. However, current review comment evaluation metrics rely on comparisons with a human-written reference for a given code change (also called a *diff*). Furthermore, code review is a one-to-many problem, like generation and summarization, with many “valid reviews” for a diff. Thus, we develop CRScore — a reference-free metric to measure dimensions of review quality like conciseness, comprehensiveness, and relevance. We design CRScore to evaluate reviews in a way that is grounded in claims and potential issues detected in the code by LLMs and static analyzers. We demonstrate that CRScore can produce valid, fine-grained scores of review quality that have the greatest alignment with human judgment among open source metrics (0.54 Spearman correlation) and are more sensitive than reference-based metrics. We also release a corpus of 2.9k human-annotated review quality scores for machine-generated and GitHub review comments to support the development of automated metrics<sup>1</sup>.

## 1 Introduction

Code Review is an essential quality control tool for software engineers to ensure that source code is free of bugs and upholds standards (McIntosh et al., 2014; Bavota and Russo, 2015). Software engineers prefer lightweight, asynchronous review processes, as enabled through GitHub’s review comment feature, over formal, in-person reviews (Beller et al., 2014; Badampudi et al., 2023). This has led to the creation of benchmarks for automated generation of natural language (NL) review comments (Li et al., 2022; Tufano et al., 2022). However, these benchmarks use reference-based evaluation metrics like BLEU (Papineni et al., 2002), which have been shown to have low validity (Re-

iter, 2018; Evtikhiev et al., 2023), especially when paired with limited and low-quality references.

Code review is fundamentally a one-to-many problem, where a given diff can have multiple possible issues that a review can tackle. Having a limited number of reference reviews (e.g., one per diff in CodeReviewer (Li et al., 2022)) leads to unfairly low scores with reference-based metrics. For example, for the diff shown in Figure 1, the ground truth review focuses on whether the `ToHexString()` function could cause a performance issue. However, the model-generated review focuses on `ToHexString().Equals("0000000000000000")` being an odd condition with a scenario where its being triggered is unlikely, which is also a valid review for the diff. However, the BLEU score value for the model-generated review is very low, specifically at 0.0458, due to poor n-gram overlap. Additionally, the references can also be low-quality due to missing context, as shown in Table 5, or can focus on trivial and tangential issues (Pangakulyanont et al., 2014). Such low-quality references, paired with reference-based metrics, can harshly and unfairly penalize models.

Motivated by these drawbacks, we propose CRScore, an automated but reference-free evaluation metric that uses dimensions of review quality from prior work (Piorkowski et al., 2020; Turzo and Bosu, 2024). In particular, *Comprehensiveness* – does the review convey all the necessary information? *Conciseness* – does the review only convey the necessary information in an efficient way and *Relevance* – is all the information on topic? We operationalize our metric through a two-step process: ① generate a list of *pseudo-references* spanning information like possible claims, issues, and implications of a code change, and ② use semantic textual similarity (STS) to align parts of the review to the pseudo-references. To generate the pseudo-references, we use a neuro-symbolic approach that

<sup>1</sup><https://github.com/atharva-naik/CRScore>

combines **Large Language Models** (LLMs) and **Code Analysis Tools** (CATs) that can detect formatting errors, faulty design patterns (code smells [Rasheed et al. 2024](#)), etc. We combine these methods for more exhaustive pseudo-references and to overcome drawbacks of each method (section [A.1](#)).

Finally, we demonstrate the validity of CRScore by human evaluation of the quality of pseudo-references and by measuring the alignment of our metrics with human judgment. We show that a large number (82.6%) of pseudo-references generated by CRScore are correct, and that it has the greatest alignment with (Spearman correlation 0.5431) and sensitivity to (Figure 8) the human judgment of review quality, as compared to that of the reference-based metrics.

**Contributions.** (1) We propose an automated reference-free metric that combines the advantages of LLMs and CATs to measure review quality along fine-grained dimensions. (2) We collect human annotations of pseudo-reference quality and review conciseness, comprehensiveness, and relevance to validate our approach. We plan to make these scores publicly available as a resource for continued development of automated metrics. (3) We benchmark several LLMs of code for code review on the CodeReviewer dataset using CRScore and compare it with reference based metrics.

```
@@ -133,7 +133,7 @@ namespace OpenTelemetry.Trace
    private void RunGetRequestedDataOtherSampler(
        Activity activity)
    {
        ActivityContext parentContext;
-       if (string.IsNullOrEmpty(activity.ParentId))
+       if (string.IsNullOrEmpty(activity.ParentId)
|| activity.ParentSpanId.ToHexString().Equals(
"0000000000000000"))
{
        parentContext = default;
    }
}
```

**Ground Truth:** this **maynot** be a perf issue, if `ToHexString()` is not actually allocating a string, but returns the caches string value. to be confirmed.

**Model Generated Review:** This is a bit of a guess, but it seems like the parentContext ...

Also, the `check for activity.ParentSpanId.toHexString().Equals("0000000000000000")` seems a bit odd. It seems like it's checking if the parent span id is all zeros, which is not a common scenario. It might be better to check if the parent id is null or empty.

**BLEU:** **0.0458**

Figure 1: Example diff with multiple valid reviews. The ground truth and model-generated reviews focus on different topics, like the performance of the added check, and how likely it is to be triggered. However, a reference-based metric like the BLEU score assigns this review a low score of 0.0458.

## 2 Related Work

In this section, we summarize the limitations of reference-based evaluation, the need for better code review evaluation metrics, inspiration from reference-free evaluation for other tasks, and how code smells can be leveraged to evaluate reviews.

### 2.1 Reference Based Evaluation Metrics:

Reference-based metrics like BLEU ([Papineni et al., 2002](#)), ROUGE ([Ganesan, 2018](#)), and BERTScore ([Zhang et al., 2020](#)) have seen widespread adoption for text generation tasks like translation and summarization due to their convenience. While metrics like BLEU, ROUGE, and character F-score ([Popović, 2015, 2017](#)) use n-gram overlap between the reference and candidate text, metrics like BERTScore ([Zhang et al., 2020](#)) try to capture the semantic similarity. However prior studies have shown metrics like BLEU to have low validity (overlap with human judgment) and reliability ([Reiter, 2018; Evtikhiev et al., 2023](#)). Meanwhile, BERTScore can fail for candidates with errors that are lexically and stylistically similar to references ([Hanna and Bojar, 2021](#)).

### 2.2 Code Review Evaluation

Due to the high time and resource demands of code review, automated approaches have gained popularity ([Yang et al., 2024](#)). [Tufano et al. \(2021, 2022\)](#); [Li et al. \(2022\)](#) proposed large datasets for code review tasks like code changes quality detection, review comment generation, and code refactoring. However, these datasets used reference-based metrics and thus suffer from the issues highlighted in sec [2.1](#). While many studies have focused on modeling methods for code review tasks ([Pornprasit and Tantithamthavorn, 2024; Lu et al., 2023; Dong-Kyu, 2024; Fan et al., 2024; Yu et al., 2024; Lin et al., 2024](#)), they either retain the same reference-based automated metrics like BLEU ([Papineni et al., 2002](#)), or use human evaluation. Some studies have focused on evaluating style and presentation of review comments for usefulness ([Rahman et al., 2017; Ochodek et al., 2022; Yang et al., 2023](#)) (see Appendix [B.3](#) for detailed comparison) or negativity and toxicity ([Ahmed et al., 2017; Sarker et al., 2023](#)). This work focuses on content-focused and reference-free automated evaluation. We show that reference-based metrics combined with noisy references fail to capture human preferences. We propose CRScore, the first automated

content-focused reference-free metric to overcome these limitations.

### 2.3 Reference Free Evaluation

Reference-free evaluation metrics have been proposed for various text-generation tasks to capture multiple valid outputs. Instead of using references, these metrics try to measure general “quality dimensions” like relevance, informativeness, etc. VIFIDEL (Madhyastha et al., 2019), InfoMetIC (Hu et al., 2023), and ClipScore (Hessel et al., 2021) evaluate dimensions like faithfulness, informativeness, and relevance, respectively for image captioning. FED (Mehri and Eskenazi, 2020a), and USR (Mehri and Eskenazi, 2020b) evaluate dimensions like informativeness, relevance, and overall quality for dialog. Studies on the helpfulness of software documentation and code review (Piorkowski et al., 2020; Turzo and Bosu, 2024) propose quality dimensions like conciseness, completeness, understandability, relevance, and supporting evidence. The common trend across these studies is some notion of conciseness, informativeness/comprehensiveness, and relevance being useful, prompting us to focus on them. However, our work is the first to operationalize these dimensions for automated evaluation.

### 2.4 Code Smell Detection

“Code smells” (Fowler, 1997) are design flaws and bad practices (also called *anti-patterns*) that can lead to maintainability issues. Detecting code smells automatically has traditionally been accomplished by static analysis-based approaches (Tsantalis et al., 2008; Paiva et al., 2017; Liu and Zhang, 2017). Recently machine learning (Sandouka and Aljamaan, 2023) and transfer learning (Sharma et al., 2021) based approaches have been proposed to learn more complex heuristics. Recent approaches have leveraged LLMs via prompting (Liu et al., 2024) and agents (Rasheed et al., 2024) to achieve improvement and tackle repository-level code smell detection. However, these approaches are limited in the types of smells they target (Liu et al., 2024; Sandouka and Aljamaan, 2023), training data requirements (Zhang et al., 2024), or lack comprehensive evaluation (Rasheed et al., 2024). Also, code smells differ across programming languages (Abidi et al., 2019), and transfer learning approaches can only be leveraged for similar languages (Sharma et al., 2021). Due to these limitations of learning-based methods and to mitigate the

self-selection bias of LLMs (sec A.1) we use code analysis tools.

## 3 Operationalizing CRScore

Motivated by the one-to-many nature of code reviews, noisy references, and the pitfalls of reference-based automated metrics, we develop CRScore — a reference-free, quality dimension-based automated metric. As shown in Figure 2, instead of relying on explicit references, our metric generates “pseudo-references” from the code change spanning claims, implications, and issues/smells that hurt maintainability — in other words, some topics that a review should address (Rasheed et al., 2024). Then we use semantic textual similarity (STS) measures to quantify how much these topics are addressed by a code review as shown in Figure 2 through the lens of three quality dimensions: conciseness, comprehensiveness, and relevance. They capture review quality, similar to precision, recall, and f-score used for classification and retrieval. We describe the three main components of our framework — the quality dimensions, pseudo-reference generation, and similarity measurement — below.

### 3.1 Quality Dimensions

We are inspired by (Piorkowski et al., 2020; Turzo and Bosu, 2024) to pick the dimensions of “conciseness”, “comprehensiveness” (similar to completeness), and “relevance”. The former two capture and strike a balance between comprehensive reviews with a lot of detail and concise, minimalist reviews. Conciseness and comprehensiveness play the role of precision and recall respectively, capturing how much of the review is on topic, and how much information is covered. Relevance strikes a balance between the two, like f-score, measuring the overall quality of the review. Additionally, we also do a human evaluation of the validity of the pseudo-references focusing on aspects like supporting evidence (Piorkowski et al., 2020).

### 3.2 Pseudo Reference Generation

To generate pseudo references we develop an LLM-based pipeline for generating **claims** about the code changes on two levels of abstraction: 1) Low-level changes and 2) High-level “implications” of the change. To allow for reproducibility and ease of deployment we use a 6.7B parameter open source LLM, Magicoder (Wei et al., 2023). We fine-

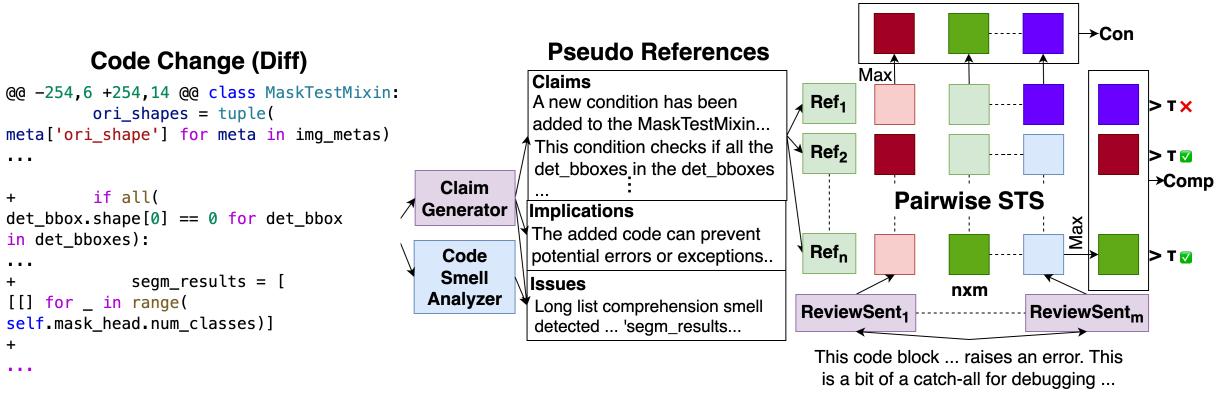


Figure 2: **Operationalization of CRScore:** Our metric first generates pseudo-references for the diff — claims, implications and issues. Then each pseudo-reference is embedded by a sentence transformer along with each review sentence and the pairwise semantic textual similarity (STS) is computed. The high similarity threshold  $\tau$  is used to compute the Con and Comp metrics whose harmonic mean yields the Rel score.

tune Magicoder-S-DS-6.7B for this task using task-specific data produced by a stronger model (GPT-4) as shown in Figure 3. We generate claims by prompting GPT-4 for a random subset of 1k code changes from the CodeReviewer validation data.

In addition to the claims, we also utilize program analysis tools that can detect **issues** and “code smells” (characteristics that indicate deeper flaws in programs as shown in Table 14) like PySmell<sup>2</sup> as well as static analyzers like PMD<sup>3</sup> and JSHint<sup>4</sup>. They can detect unused variables, unnecessary object creation, syntax errors, leaking variables, type conversion issues, etc. These tools target Python, Java, and Javascript respectively, and can use rule-based analysis to detect issues at a file or repository level. They complement the aspects that might be missed by an LLM-based analysis. We combine the code smells and claims to get the final set of pseudo references.

### 3.3 Computing Similarity with Pseudo References:

We use a Sentence Transformer (Reimers, 2019) model (mxbai-embed-large-v1<sup>5</sup>) to compute Semantic Textual Similarity (STS) between pseudo-references and review sentences. The pairwise similarities are then used to compute the conciseness, comprehensiveness, and relevance, as shown by equations 1, 2, and 3. We picked this model because it has the best performance, as of July 2024,

on English STS on the MTEB benchmark (Muenighoff et al., 2022) for models with less than 1B parameters.

We start by computing token embeddings for the pseudo-references (p-refs) and review sentences (r-sents), excluding the stopwords and pooling the rest of the token embeddings to build representations for the whole sentence. Then we compute STS scores with these sentence embeddings via pairwise cosine similarity  $s()$  between the p-refs ( $\mathcal{P}$ ) and r-sents ( $\mathcal{R}$ ). The Conciseness ( $Con$ ) which is computed as:

$$Con = \frac{\sum_{r \in \mathcal{R}} I[\max_{p \in \mathcal{P}} s(c, r) > \tau]}{|\mathcal{R}|} \quad (1)$$

represents the fraction of r-sents from the model-generated review with greater similarity to any p-ref above a threshold  $\tau$ . Here,  $I$  is an indicator variable such that:

$$I[x] = \begin{cases} 1, & \text{if } x \text{ is true} \\ 0, & \text{otherwise} \end{cases}$$

$Con$  resembles precision as it captures the fraction of r-sents (candidate set) that are “on topic” concerning the p-refs (reference/gold set). The Comprehensiveness ( $Comp$ ) computed as:

$$Comp = \frac{\sum_{p \in \mathcal{P}} I[\max_{r \in \mathcal{R}} s(c, r) > \tau]}{|\mathcal{P}|} \quad (2)$$

represents the fraction of p-refs that have greater similarity to any of the r-sents than the threshold  $\tau$ . This metric resembles recall as it captures the fraction of the p-refs (gold set) covered by the model-generated r-sents (candidate set). There is a trade-off between conciseness and comprehensiveness

<sup>2</sup><https://github.com/whyjay17/Pysent>

<sup>3</sup><https://pmd.github.io/>

<sup>4</sup><https://jshint.com/>

<sup>5</sup><https://huggingface.co/mixedbread-ai/mxbai-embed-large-v1>

just like precision and recall, and to capture the trade-off between these metrics like the F1 score, we define the Overall Relevance  $Rel$  as the harmonic mean of  $Con$  and  $Comp$ :

$$Rel = \frac{2 \cdot Con \cdot Comp}{Con + Comp} \quad (3)$$

Based on the definition of  $I[x]$  all of our metrics range in value from  $[0, 1]$ .

**Selecting threshold  $\tau$  for high similarity:** For our similarity threshold  $\tau$  we use the average similarity between a pseudo-reference and the review sentence talking about it. To compute it we use the distribution of pseudo-reference and review similarity scores for the best model (GPT-3.5 in our case). Starting with the CodeReviewer test set we exclude the data used for collecting human annotations. For the remaining 9869 instances, we compute the similarity between each review sentence and pseudo-reference for its corresponding code change. Then we associate each review sentence with the most similar pseudo-reference to capture correspondence. Finally, we average the similarity score value across all review sentences, giving us a value for the threshold<sup>6</sup>  $\tau_{best} = 0.7314$ . While choosing the right threshold is important for our metric, we note that it is robust to some variations in the value as shown in Table 18.

## 4 Validating CRScore

To be a valid metric, CRScore needs to satisfy a few properties. Firstly, the generated pseudo-references should have few errors and unverifiable claims while being exhaustive. Additionally, we want our dimension-level scores ( $Con$ ,  $Comp$ , and  $Rel$ ), especially  $Rel$ , to correlate with human judgment for each dimension. The most important property, is arguably the ability of our metric (especially  $Rel$ ) to rank review generation systems in the same order as humans. In the subsequent sections, we describe how we design the experiments to collect annotations for these properties (section 4.1, 4.2), choose a set of systems to be rated by humans (section 4.4), and set up reference-based metrics (section 4.5) for comparison CRScore.

### 4.1 Rating Quality of Pseudo-References

To show that LLM-generated pseudo-references used by the CRScore evaluation pipeline are high-

<sup>6</sup>We also show that computing the threshold with ground truth reviews yields a comparably good threshold  $\tau_{GT} = 0.6576$  with respect to correlation with human annotations.

quality, we gather annotations capturing incorrect, unverifiable, and missing claims. We had two trained human annotators (co-authors of this paper) judge the quality of the pseudo-references for 100 randomly sampled code changes from CodeReviewer (Li et al., 2022) each in Python, Java, and Javascript (300 total, see Appendix D.5 for details). The annotators were asked to code the pseudo-references as 1 (correct based on evidence), 0 (incorrect based on evidence), and -1 (unverifiable due to lack of evidence). They were also asked to add any pseudo-references about issues/claims not covered by the pseudo-references. We report the fraction of correct claims (accuracy), incorrect claims (error rate), unverifiable claims (unverifiable rate), and missing claims (missing rate). If  $N_c$ ,  $N_u$ ,  $N_i$ , and  $N_a$  represent the number of correct, unverifiable, incorrect, and added claims then each of these rates can be calculated as:

$$\begin{aligned} \text{Accuracy} &= \frac{N_c}{N_c + N_u + N_i} \\ \text{Error Rate} &= \frac{N_i}{N_c + N_u + N_i} \\ \text{Unverifiable Rate} &= \frac{N_u}{N_c + N_u + N_i} \\ \text{Missing Rate} &= \frac{N_a}{N_c + N_u + N_i} \end{aligned}$$

Based on these expressions: Accuracy + Unverifiable Rate + Error Rate = 1. The results for each language are shown in table 1. The annotators follow guidelines laid out in a codebook (section D.1) which includes examples for each category. We measured the coding reliability of our approach by collecting annotations from both annotators on a common set of 100 pseudo-references. These annotations yielded a Cohen Kappa of 0.804 which indicates great inter-annotator reliability (Landis and Koch, 1977). Also, we only evaluate the LLM-generated claims here as we know the static code analysis tools are rule-based and reliably correct.

### 4.2 Rating Review Quality Dimensions

To show that CRScore aligns with the human judgment of review quality along the proposed dimensions: comprehensiveness, conciseness, and relevance we gather annotations from the same annotators on reviews generated by 9 systems (section 4.4) and the ground truth references. We use the same code changes used in section 4.1.

The raters again follow a codebook (section D.2) that contains guidelines and examples for annotating each dimension using the pseudo-references on a 5-point Likert scale. The raters use the updated set of pseudo-references based on the first phase of annotations described in section 4.1. Incorrect and unverifiable references are removed and missing claims are added. We also add the pseudo-references (issues and code smells) generated by the code analysis tools for each code change. Raters are also asked to annotate any claims they find unnecessary for a given code change, which are then excluded while rating the review on each quality dimension. Again, we measure the reliability of the codebook by collecting annotations from the two raters, this time on a common set of 100 reviews. We compute Krippendorff’s alpha reliability (Krippendorff, 2018) for each dimension, yielding values of 0.8868, 0.8505, and 0.8806 for conciseness, comprehensiveness, and relevance respectively. Alpha values  $> 0.8$  are generally considered reliable. Despite the steps taken to ensure reliability of the annotations, there might be some concerns about potential biases, which we address in Appendix D.9.

Based on these annotations, we measure the agreement between CRScore (*Rel*) and the reference-based metrics with the human-annotated relevance Likert scores using Kendall and Spearman correlation (Table 3). We also compute the correlation between the system rankings generated by the metric and human annotations (Table 2).

Language	Accuracy	Error Rate	Unverifiable Rate	Missing Rate
Python	83.65	12.02	4.33	5.77
Java	79.09	13.22	7.69	8.89
Javascript	85.21	6.52	8.27	2.51

Table 1: Quality of pseudo-references – the fraction of correct (accuracy), incorrect (error rate), unverifiable (unverifiable rate), and added claims (missing rate) based on human annotations.

### 4.3 Annotating Similarity between Pseudo-References and Reviews

While annotating the review quality dimensions using the pseudo-references the annotators are also asked to mention the pseudo-references covered by each system-generated review. This gives us a dataset of 5.7k pseudo-reference and review pairs, where 1948 are positive examples or cases where

Metric	$r_s$	$p$	$\tau$	$p$
BLEU	-0.3	0.433	-0.1667	0.612
BLEU (without stop)	-0.15	0.7	-0.0556	0.919
BERTScore	0.35	0.356	0.2222	0.477
Normalized Edit Distance	0.1667	0.668	0.0556	0.919
ROUGE-L	0.0167	0.966	0.556	0.919
F-measure	0.4833	0.187	0.3889	0.18
chrF	0.6	0.088	0.4444	0.119
chrF++	0.8	0.010	0.6667	0.013
LaaJ-Magic	<b>0.9833</b>	<b>1.9e-6</b>	<b>0.9444</b>	<b>5.0e-5</b>
<b>Rel</b> ( $T_{best}$ )	<u>0.95</u>	<u>8.7e-5</u>	<u>0.8889</u>	<u>2.4e-4</u>
<b>ours</b>				

Table 2: Spearman ( $r_s$ ) and Kendall ( $\tau$ ) correlations between system rankings produced by each metric and human annotations for relevance. Only our metric and the Laaj variants achieve a high, statistically significant correlation. Our metric has a comparable correlation to Laaj-GPT which uses a much more powerful closed-source model, while being much better than Laaj-Magic which uses Magicoder-S-DS-6.7B, the same based model as CRScore.

the review addresses a claim according to a human, while the rest are negative examples or pairs where the reviews don’t address a pseudo-reference. Using this data we evaluate the STS model with various similarity thresholds ( $tau$ ) using it to convert the similarity score into a binary classifier and report its precision, recall and F1-score in predicting whether in a pseudo-reference and review pair the review addresses the pseudo-reference. The results are shown in Table 4

### 4.4 Review Generation Systems

To see if CRScore can rank code review systems of varying capacity, we choose a diverse set of review generation models. They span various parameter sizes, pre-training, and fine-tuning strategies:

**Simple baselines:** We create two simple baselines, namely, a BM-25 retriever and an LSTM as described in section D.3. We choose these models with the expectation that they will likely perform the worst, to see if our metric assigns them a low score.

**CodeReviewer:** We pick the CodeReviewer model from (Li et al., 2022) as it is a transformer-based model trained on code review-specific data and objectives.

**Open source LLMs:** We prompt several open-

source LLMs in a few-shot manner with a fixed set of three example code changes and review pairs from the validation set. We use LLMs in the 3-13B parameter range: Stable-Code-3B (Pinnaparaju et al.), DeepSeekCoder-6.7B (Guo et al., 2024), Magicoder-6.7B (Wei et al., 2023), CodeLLaMA-7B and 13B (Roziere et al., 2023) and LLaMA-3-8B (AI@Meta, 2024).

**Closed source LLMs:** We prompt closed-source LLMs like GPT-3.5 in a manner similar to the open-source LLMs.

#### 4.5 Reference-based Metrics

We pick commonly used reference-based metrics for code review and other text generation tasks to compare with CRScore:

**BLEU** (Papineni et al., 2002) measures the n-gram precision between the generated text and references with an additional brevity penalty to discourage short outputs. It is used for evaluation in both (Tufano et al., 2021) and CodeReviewer (Li et al., 2022). We report the results with and without stop word removal.

**Normalized Edit Distance** is a normalized Levenshtein distance used in prior work (Tufano et al., 2021; Bairi et al., 2024) to measure the number of edits required to match candidate and target reviews or code.

**ROUGE-L F-measure** is a popular recall-oriented metric originally proposed for summarization and machine translation. We use the longest common subsequence-based sentence level f-measure implementation.<sup>7</sup>

**chrF:** (Popović, 2015) Is a machine translation metric which is essentially a character level F score computed using character level n-grams.

**chrF++:** (Popović, 2017) Is a variant of chrF that additionally incorporates word n-grams.

**BERTScore:** (Zhang et al., 2020) We use the BERTScore F1 measure to capture the semantic similarity between the reference review and the generated review.

**LLM-as-a-judge (LaaJ):** Following recent developments (Zheng et al., 2023) we develop an LLM prompting-based approach that compares the model-generated reviews against CodeReviewer references to generate relevance scores. We prompt them with descriptions of the same Con, Comp, and Rel dimensions developed in section 3.1 with the prompt shown in D.4. We evaluate two variants

of this metric one that uses GPT-4o as the judge LLM (LaaJ-GPT) and one that uses Magicoder-S-DS-6.7B (LaaJ-Magic), the opensource model used by CRScore, for a fair comparison.

### 5 Results

#### 5.1 Validity of Pseudo-References

We show the rates of correct, incorrect, unverifiable, and missing claims as explained in section 4.1 in Table 1. The pseudo-references produced by our pipeline were relatively accurate with roughly 82.6% accuracy across the languages. The best performance is for Javascript and the worst is for Java. The most frequent issues in the code claims were incorrect claims. Most of the errors were in code comprehension (“misreading the code”) and over-generalization (incorrect assumptions made from the limited context, contradicting file level context). Some examples are shown in Table 12. For Javascript, we observed unverifiable claims to be the biggest issue, e.g., claims about code efficiency or functionality made without evidence. E.g.: “However, this change could also potentially enable less strict ... behavior, ... This could make the code less efficient ...”.

Metric	$\tau$	$r_s$
BLEU	0.001	-0.0001
BLEU (without stopwords)	0.0425	0.0542
BERTScore	0.081	0.1083
Normalized Edit Distance	0.0193	0.0249
ROUGE-L F-measure	0.0757	0.0989
chrF	0.1484	0.1966
chrF++	0.1555	0.2057
LaaJ-Magic	0.2464	0.2748
<b>LaaJ-GPT</b>	<b>0.5247</b>	<b>0.605</b>
<b>Rel (<math>\tau_{best}</math>) (Ours)</b>	<b>0.4567</b>	<b>0.5431</b>

Table 3: Comparing Kendal-Tau ( $\tau$ ) and Spearman Rank ( $r_s$ ) correlation of reference-based evaluation metrics and our reference-free relevance score (Rel) with human annotations for the relevance dimension. Results that don’t achieve statistical significance are grayed out. Our metric archives the second-best correlation behind Laaj-GPT however it does so with a much smaller 6.7B parameter open-source model. Additionally, CRScore outperforms Laaj-Magic which uses the same base LLM (Magicoder) as the judge model.

<sup>7</sup><https://pypi.org/project/rouge-score/>

## 5.2 Validity of Review Quality Dimension Scores (Con, Comp and Rel)

**Correlation with human Likert score annotations:** We compute the Spearman and Kendall rank correlations between the human-annotated Likert scores and the metric values. These values were gathered for the 300 CodeReviewer test instances mentioned in section 4.1, (results in Table 3). We exclude human annotations done on the ground truth references (“Ground Truth” row in Table 10) because the reference-based metric value for these would be 1 by default unfairly lowering their correlations. We observe that while almost all the reference-based metrics have weak to no correlations with human judgment our metric *Rel* and LaaJ-GPT achieve the greatest correlations. We also show correlations between human Likert scale annotations for all dimensions and all metrics (including *Con* and *Comp*) in Table 9.

**Comparing system rankings:** Arguably the most important desideratum for our metric is the ability to rank systems similar to human evaluators. We compare the system rankings produced by our metric *Rel* with rankings produced by human annotations for relevance, showing the correlations in Table 2. Only the LaaJ metrics and our metric achieve a strong, statistically significant correlation with the rankings computed from human annotations. The system rankings (shown in Table 15) reveal that our metric gets the ranking mostly right, except for LLaMA-3-8B-Instruct, which is ranked slightly lower by our metric compared to human relevance annotations. We also report the quality dimension scores for each system according to our metrics (*Con*, *Comp*, and *Rel*) and human annotations in Table 10. Our metrics also have a similar spread of values to the human annotations and greater sensitivity to human preferences as shown in Figure 8 and Table 13.

## 5.3 CodeReviewer Dataset Reference Quality

We compare the quality of the CodeReviewer reference reviews with the reviews generated by the 9 system evaluated by the human annotators. The average scores for conciseness, comprehensiveness, and relevance attained by the CodeReviewer references are 3.05, 1.88, and 2.13, while the average scores obtained by all 9 systems are 2.57, 1.84, and 1.99. This suggests that the average reference review is barely better than the average evaluated system according to the human annotators for rel-

STS Threshold ( $\tau$ )	P	R	F1
0.6	0.4055	0.9512	0.5686
0.65	0.4433	0.8794	0.5895
0.6576 ( $\tau_{GT}$ )	0.4475	0.8665	0.5902
0.7	0.4832	0.77	0.5938
0.7314 ( $\tau_{best}$ )	0.5173	0.6899	0.5913
0.75	0.5401	0.633	0.5828
0.8	0.6223	0.444	0.5183

Table 4: Precision (P), Recall (R), and f1-score (F1) achieved by the STS embedding model for various STS thresholds ( $\tau$ ) evaluated on the human annotations linking each review sentence to the pseudo-references addressed by it.

evance, but they are more concise. Additionally, the best system according to human annotators (as evident from Table 10), GPT-3.5 achieves average scores of 3.63, 2.65, and 2.9 for each dimension – much better than the reference reviews. This provides further motivation for the development of reference-free evaluation metrics like CRScore for code review.

## 5.4 Failure Cases

We analyze the cases where our metric greatly overestimates or underestimates the quality of a review with respect to human annotations. We find such cases using the procedure described in Appendix E.1. This is also supported by the results in Table 4 that show that the STS model used while having a decent amount of recall, has relatively low precision values for most of the thresholds evaluated including  $\tau_{best}$ , the threshold used in Table 2 and 3. For underestimation cases, we observe our pseudo-reference generation pipeline generates fewer references on average (2.44) compared to the whole data (4.76). This suggests that even though the STS model has a high recall, having fewer pseudo-references, makes it harder to evaluate the relevance of reviews since CRScore might underestimate comprehensiveness compared to humans. Additionally, we observe reviews like “Why do we need these imports” which are brief, contain stopwords, and have fewer relevant tokens, making it hard for STS to recognize their relevance to pseudo-reference. For overestimation cases, we observe the presence of inline code snippets at a higher rate (45%) compared to all reviews (28%) and underestimation cases (12%). Some example reviews for each case are shown in Table 16, 17.

## 6 Discussion

In this work we identify issues with current code review evaluation benchmarks like CodeReviewer (Li et al., 2022), which fail to capture the one-to-many nature of code review and contain noisy references. To enable auditing current evaluation metrics and aid the development of a better metric, we propose three review quality dimensions – conciseness, comprehensiveness, and relevance based on current literature on reference-free evaluation (Mehri and Eskenazi, 2020b; Piorkowski et al., 2020; Turzo and Bosu, 2024).

To ground these dimensions in topics that reviews should address (Rasheed et al., 2024), we propose an automated pseudo-reference generation pipeline that leverages LLMs and code smell detectors. We validate the quality of these pseudo-references via human evaluation. Based on these dimensions, we develop reliable guidelines for coding review quality using pseudo-references and collect annotations for 9 review generation systems and “ground truth” reviews for the CodeReviewer dataset spanning Python, Java, and Javascript.

The collected annotations show that current reference-based metrics except for LLM-as-a-judge with powerful closed source models like GPT-4o fail to capture human preferences, which is further compounded by humans preferring some models over the references. We propose CRScore as a metric to capture the three dimensions using the pseudo-references generated by open source LLMs and static analysis tools like code smell detectors through STS models like sentence transformers (Reimers, 2019). Our approach has the second greatest alignment with human preferences, lagging only behind Laaj-GPT and the greatest alignment among open source metrics. It achieves the best review quality correlation scores ( $0.4577 \tau$  and  $0.5425 r_s$ ), system ranking correlations ( $0.95 r_s$  and  $0.8889 \tau$ ), as well as the greatest sensitivity (Figure 8) among open source metrics. Additionally, it greatly improves over using the Magicoder LLM directly as a judge, while also being more efficient (since the LLM needs to be run only once to produce the pseduo-references as compared to the LLM-as-a-judge, where it needs to be re-run for every review model evaluated). However, we also note that despite its reproducibility, efficiency and greater alignment with human preferences, CRScore exhibits only moderate correlations with review quality, and systematically underestimates

or overestimates it in some cases.

## 7 Conclusion

Our work takes the first steps towards addressing the challenges involved in evaluating the quality of code reviews. We propose useful dimensions for capturing review quality, and offer CRScore as an automated, efficient, open-source and reproducible metric for capturing them. We collect a dataset of human judgment of review quality scores to show the validity of CRSore. We compare CRScore with 7 reference-based metrics, for 9 review generation systems using the collected annotations. Our metric achieves the best alignment with human preferences among open source metrics and is the most sensitive. However, there is scope for improvement by developing better pseudo-reference generation and STS matching methods to try and match the performance of closed-source models like GPT-4o in judging review quality.

## 8 Future Work

Although our metric is a great first step towards reference-free evaluation of code reviews, it still suffers from systematic under and over-estimation errors in certain cases, which causes it to fall short of powerful closed source models. We believe these limitations stem from pseudo-reference coverage issues and the limitations of STS methods when it comes to matching data containing both code and text. Future work can extend the pseudo-references by adding components for detecting code security, efficiency issues, etc. Also, better embedding models should be developed to measure how faithfully review sentences capture the pseudo-references. Additionally, code smell detection can be expanded beyond Python, Java, and Javascript to languages like C/C++, Ruby, and Go, present in the CodeReviewer dataset.

## Limitations

- While annotating reviews for the review quality dimensions of conciseness, comprehensiveness, and relevance, the human annotators are given a list of pseudo-references that overlap with the one used by CRScore. We believe this isn’t a source of anchoring bias because the human annotators are allowed to add and remove claims from the pseudo-references in the first stage of annotation (while rating pseudo-reference quality). The final list of

pseudo-references used for review quality annotations is different from the one used by CRScore. Additionally, the pseudo-references provide a common ground for the annotators to rate review quality and comprehensiveness. This also helped us achieve high reliability for coding review quality (section 4.2).

- Semantic textual similarity (STS) models are imperfect at matching relevant pseudo-references to the review sentences, especially when there are very few claims and the reviews contain inline code snippets, where the latter can inflate the STS scores.
- Our pseudo-reference generation pipeline is not comprehensive enough in some cases which can lead to underestimation of review quality as shown by our failure case analysis. Additionally, it could be extended by adding more modules like code smell detectors for aspects like code security, code efficiency, etc. similar to Rasheed et al. (2024). Also, code smell detector tools can be added for languages other than Python, Java, and Javascript, like Go, C/C++, and Ruby present in the CodeReviewer dataset.
- Our metric only achieves a moderate correlation with human annotations of review quality, and while it is much better than the reference-based metrics in terms of alignment with human judgment and sensitivity to review quality as judged by humans, it is only a first step towards developing better metrics for code review. Future work should try to address the limitations of our claim generation pipeline and STS methods.

## Ethics Statement

Our work doesn't violate any ethical guidelines and is compliant with copyright rules and regulations as we use an existing publicly available dataset and augment it with annotations of review quality using reviews generated by 9 systems and the references in the dataset. While there is a slight risk of harmful or toxic text being a part of the pseudo-references generated by the LLM component in our pseudo-reference generation pipeline we don't believe it to be a major risk based on the annotations done for the pseudo-reference quality.

## References

Apache lucene.

- Mouna Abidi, Manel Grichi, Foutse Khomh, and Yann-Gael Guéhéneuc. 2019. [Code smells for multi-language systems](#). In *Proceedings of the 24th European Conference on Pattern Languages of Programs*, EuroPLop '19, New York, NY, USA. Association for Computing Machinery.
- Toufique Ahmed, Amiangshu Bosu, Anindya Iqbal, and Shahram Rahimi. 2017. Senticr: A customized sentiment analysis tool for code review interactions. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 106–111. IEEE.
- AI@Meta. 2024. [Llama 3 model card](#).
- Deepika Badampudi, Michael Unterkalmsteiner, and Ricardo Britto. 2023. [Modern code reviews—survey of literature and practice](#). *ACM Transactions on Software Engineering and Methodology*, 32(4):1–61.
- Dzmitry Bahdanau. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.
- Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B Ashok, and Shashank Shet. 2024. Codeplan: Repository-level coding using llms and planning. *Proceedings of the ACM on Software Engineering*, 1(FSE):675–698.
- Gabriele Bavota and Barbara Russo. 2015. [Four eyes are better than two: On the impact of code reviews on software quality](#). In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 81–90.
- Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. 2014. [Modern code reviews in open-source projects: Which problems do they fix?](#) In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, page 202–211, New York, NY, USA. Association for Computing Machinery.
- Wikipedia Contributors. 2019a. [Cohesion \(computer science\)](#).
- Wikipedia Contributors. 2019b. [Cyclomatic complexity](#).
- Lee Dong-Kyu. 2024. A gpt-based code review system for programming language learning. *arXiv preprint arXiv:2407.04722*.
- Aryaz Eghbali and Michael Pradel. 2023. [Crystalbleu: Precisely and efficiently measuring the similarity of code](#). In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA. Association for Computing Machinery.

- Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. 2023. Out of the bleu:: How should we assess quality of the code generation models?
- Lishui Fan, Jiakun Liu, Zhongxin Liu, David Lo, Xin Xia, and Shanping Li. 2024. Exploring the capabilities of llms for code change related tasks. *arXiv preprint arXiv:2407.02824*.
- Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Dixin Jiang, and Ming Zhou. 2020. [Codebert: A pre-trained model for programming and natural languages](#). In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics.
- Martin Fowler. 1997. Refactoring: Improving the design of existing code.
- Alexander Frömmgen, Jacob Austin, Peter Choy, Nimesh Ghelani, Lera Kharatyan, Gabriela Surita, Elena Khrapko, Pascal Lamblin, Pierre-Antoine Manzagol, Marcus Revaj, Maxim Tabachnyk, Daniel Tarlow, Kevin Villela, Dan Zheng, Satish Chandra, and Petros Maniatis. 2024. Resolving code review comments with machine learning. In *2024 IEEE/ACM 46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*.
- Kavita Ganesan. 2018. Rouge 2.0: Updated and improved measures for evaluation of summarization tasks. *arXiv preprint arXiv:1803.01937*.
- Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming—the rise of code intelligence. *CoRR*.
- Michael Hanna and Ondřej Bojar. 2021. [A fine-grained analysis of BERTScore](#). In *Proceedings of the Sixth Conference on Machine Translation*, pages 507–517, Online. Association for Computational Linguistics.
- Jack Hessel, Ari Holtzman, Maxwell Forbes, Ronan Le Bras, and Yejin Choi. 2021. [CLIPScore: A reference-free evaluation metric for image captioning](#). In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 7514–7528, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.
- Anwen Hu, Shizhe Chen, Liang Zhang, and Qin Jin. 2023. Infometric: An informative metric for reference-free image caption evaluation. *arXiv preprint arXiv:2305.06002*.
- Klaus Krippendorff. 2018. *Content analysis: An introduction to its methodology*. Sage publications.
- J R Landis and G G Koch. 1977. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174.
- Zhen Li, Xiaohan Xu, Tao Shen, Can Xu, Jia-Chen Gu, and Chongyang Tao. 2024. Leveraging large language models for nlg evaluation: A survey. *arXiv preprint arXiv:2401.07103*.
- Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. 2022. Automating code review activities by large-scale pre-training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1035–1047.
- Hong Yi Lin, Patanamon Thongtanunam, Christoph Treude, and Wachiraphan Charoenwet. 2024. Improving automated code reviews: Learning from experience. In *Proceedings of the 21st International Conference on Mining Software Repositories*, pages 278–283.
- Haiyang Liu, Yang Zhang, Vidya Saikrishna, Quanquan Tian, and Kun Zheng. 2024. Prompt learning for multi-label code smell detection: A promising approach. *arXiv preprint arXiv:2402.10398*.
- Xinghua Liu and Cheng Zhang. 2017. The detection of code smell on software development: a mapping study. In *2017 5th International Conference on Machinery, Materials and Computing Technology (ICMMCT 2017)*, pages 560–575. Atlantis Press.
- Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. 2023. Llama-reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 647–658. IEEE Computer Society.
- Pranava Madhyastha, Josiah Wang, and Lucia Specia. 2019. [VIFIDEL: Evaluating the visual fidelity of image descriptions](#). In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 6539–6550, Florence, Italy. Association for Computational Linguistics.
- Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. 2014. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, page 192–201, New York, NY, USA. Association for Computing Machinery.
- Shikib Mehri and Maxine Eskenazi. 2020a. Unsupervised evaluation of interactive dialog with dialogpt. In *Proceedings of the 21th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, pages 225–235.
- Shikib Mehri and Maxine Eskenazi. 2020b. [USR: An unsupervised and reference free evaluation metric for dialog generation](#). In *Proceedings of the 58th*

- Annual Meeting of the Association for Computational Linguistics*, pages 681–707, Online. Association for Computational Linguistics.
- mschwager. 2016. [Github - mschwager/cohesion: A tool for measuring python class cohesion](#).
- Niklas Muennighoff, Nouamane Tazi, Loïc Magne, and Nils Reimers. 2022. [Mteb: Massive text embedding benchmark](#). *arXiv preprint arXiv:2210.07316*.
- Miroslaw Ochodek, Miroslaw Staron, Wilhelm Meding, and Ola Söder. 2022. Automated code review comment classification to improve modern code reviews. In *Software Quality: The Next Big Thing in Software Engineering and Quality*, pages 23–40, Cham. Springer International Publishing.
- Thanis Paiva, Amanda Damasceno, Eduardo Figueiredo, and Cláudio Sant’Anna. 2017. [On the evaluation of code smells and detection tools](#). *Journal of Software Engineering Research and Development*, 5(1):7.
- Thai Pangakulyanont, Patanamon Thongtanunam, Daniel Port, and Hajimu Iida. 2014. [Assessing mcr discussion usefulness using semantic similarity](#). In *2014 6th International Workshop on Empirical Software Engineering in Practice*, pages 49–54.
- Arjun Panickssery, Samuel R Bowman, and Shi Feng. 2024. Llm evaluators recognize and favor their own generations. *arXiv preprint arXiv:2404.13076*.
- 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, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.
- Nikhil Pinnaparaju, Reshinth Adithyan, Duy Phung, Jonathan Tow, James Baicoianu, and Nathan Cooper. [Stable code 3b](#).
- David Piorkowski, Daniel González, John Richards, and Stephanie Houde. 2020. Towards evaluating and eliciting high-quality documentation for intelligent systems. *arXiv preprint arXiv:2011.08774*.
- 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, Lisbon, Portugal. Association for Computational Linguistics.
- Maja Popović. 2017. [chrF++: words helping character n-grams](#). In *Proceedings of the Second Conference on Machine Translation*, pages 612–618, Copenhagen, Denmark. Association for Computational Linguistics.
- Chanathip Pornprasit and Chakkrit Tantithamthavorn. 2024. [Fine-tuning and prompt engineering for large language models-based code review automation](#). *Information and Software Technology*, page 107523.
- Md Tajmilur Rahman, Rahul Singh, and Mir Yousuf Sultan. 2024. Automating patch set generation from code reviews using large language models. In *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering-Software Engineering for AI*, pages 273–274.
- Mohammad Masudur Rahman, Chanchal K. Roy, and Raula G. Kula. 2017. [Predicting usefulness of code review comments using textual features and developer experience](#). In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 215–226.
- Zeeshan Rasheed, Malik Abdul Sami, Muhammad Waseem, Kai-Kristian Kemell, Xiaofeng Wang, Anh Nguyen, Kari Systä, and Pekka Abrahamsson. 2024. Ai-powered code review with llms: Early results. *arXiv preprint arXiv:2404.18496*.
- N Reimers. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*.
- Ehud Reiter. 2018. [A structured review of the validity of BLEU](#). *Computational Linguistics*, 44(3):393–401.
- Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, M. Zhou, Ambrosio Blanco, and Shuai Ma. 2020. [Codebleu: a method for automatic evaluation of code synthesis](#). *ArXiv*, abs/2009.10297.
- Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémie Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.
- Rana Sandouka and Hamoud Aljamaan. 2023. Python code smells detection using conventional machine learning models. *PeerJ Comput Sci*, 9:e1370.
- Jaydeb Sarker, Asif Kamal Turzo, Ming Dong, and Amiangshu Bosu. 2023. Automated identification of toxic code reviews using toxicir. *ACM Transactions on Software Engineering and Methodology*, 32(5):1–32.
- Tushar Sharma, Vasiliki Efstratiou, Panos Louridas, and Diomidis Spinellis. 2021. [Code smell detection by deep direct-learning and transfer-learning](#). *Journal of Systems and Software*, 176:110936.
- 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.
- Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. 2008. [Jdeodorant: Identification and removal of type-checking bad smells](#). In *2008 12th European Conference on Software Maintenance and Reengineering*, pages 329–331.

- Rosalia Tufano, Simone Masiero, Antonio Mastropaoletti, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. 2022. Using pre-trained models to boost code review automation. In *Proceedings of the 44th international conference on software engineering*, pages 2291–2302.
- Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota. 2021. Towards automating code review activities. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 163–174. IEEE.
- Asif Kamal Turzo and Amiangshu Bosu. 2024. What makes a code review useful to opendev developers? an empirical investigation. *Empirical Software Engineering*, 29(1):6.
- Manushree Vijayvergiya, Małgorzata Salawa, Ivan Budiselić, Dan Zheng, Pascal Lamblin, Marko Ivanković, Juanjo Carin, Mateusz Lewko, Jovan Andonov, Goran Petrović, et al. 2024. Ai-assisted assessment of coding practices in modern code review. *arXiv preprint arXiv:2405.13565*.
- Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*.
- Lanxin Yang, Jinwei Xu, Yifan Zhang, He Zhang, and Alberto Bacchelli. 2023. *Evacrc: Evaluating code review comments*. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023*, page 275–287, New York, NY, USA. Association for Computing Machinery.
- Zezhou Yang, Cuiyun Gao, Zhaoqiang Guo, Zhenhao Li, Kui Liu, Xin Xia, and Yuming Zhou. 2024. A survey on modern code review: Progresses, challenges and opportunities. *arXiv preprint arXiv:2405.18216*.
- Yongda Yu, Guoping Rong, Haifeng Shen, He Zhang, Dong Shao, Min Wang, Zhao Wei, Yong Xu, and Juhong Wang. 2024. *Fine-tuning large language models to improve accuracy and comprehensibility of automated code review*. *ACM Trans. Softw. Eng. Methodol.* Just Accepted.
- Fengji Zhang, Zexian Zhang, Jacky Wai Keung, Xiangru Tang, Zhen Yang, Xiao Yu, and Wenhua Hu. 2024. *Data preparation for deep learning based code smell detection: A systematic literature review*.
- Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. 2020. *Bertscore: Evaluating text generation with BERT*. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.
- Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36:46595–46623.
- Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. 2023. *Codebertscore: Evaluating code generation with pretrained models of code*. *Deep Learning for Code (DL4C) workshop, 11th International Conference on Learning Representations, ICLR 2023*, abs/2302.05527.

## A Introduction Details

This section contains details to corroborate some of the points made in the introduction section.

### A.1 Benefits of Neuro Symbolic Pseudo-Reference Generation

While LLMs have recently shown promise for evaluating natural language generation (NLG) (Li et al., 2024) they suffer from biases like favoring their own generations (“self-selection bias”) (Panickssery et al., 2024) or in other words, if we were to have Magicoder or GPT-3.5 as the evaluator LLM it would assign higher scores to the text generated by Magicoder and GPT-3.5 respectively. Code analysis tools (CATs) on the other hand are limited in scope compared to LLMs in detecting issues like best practice violations (Vijayvergiya et al., 2024) but don’t have any self-selection bias. However, combining these methods can reduce the self-selection bias of LLMs, while supplementing the narrow coverage of code analysis tools. Indeed the results show that despite using Magicoder as the evaluation LLM, our metric CRScore doesn’t preferentially rank Magicoder above any models other than LLaMA-3 when compared to the human ranking.

### A.2 Release of Code and Data

We plan to make the human preferences annotation dataset and code for the evaluation metric and evaluated models public if the paper is accepted. We will add the GitHub link to the code base and the dataset to the camera-ready submission.

## B More Related Work

### B.1 Code Specific Reference Based Metrics

Due to the popularity and convenience of automated reference-based metrics like BLEU (Papineni et al., 2002), ROUGE (Ganesan, 2018), and BERTScore (Zhang et al., 2020) the research community has developed several code-specific versions like CodeBLEU (Ren et al., 2020), RUBY (Tran et al., 2019), CrystalBLEU (Eghbali and Pradel, 2023) and CodeBERTScore (Zhou et al.,

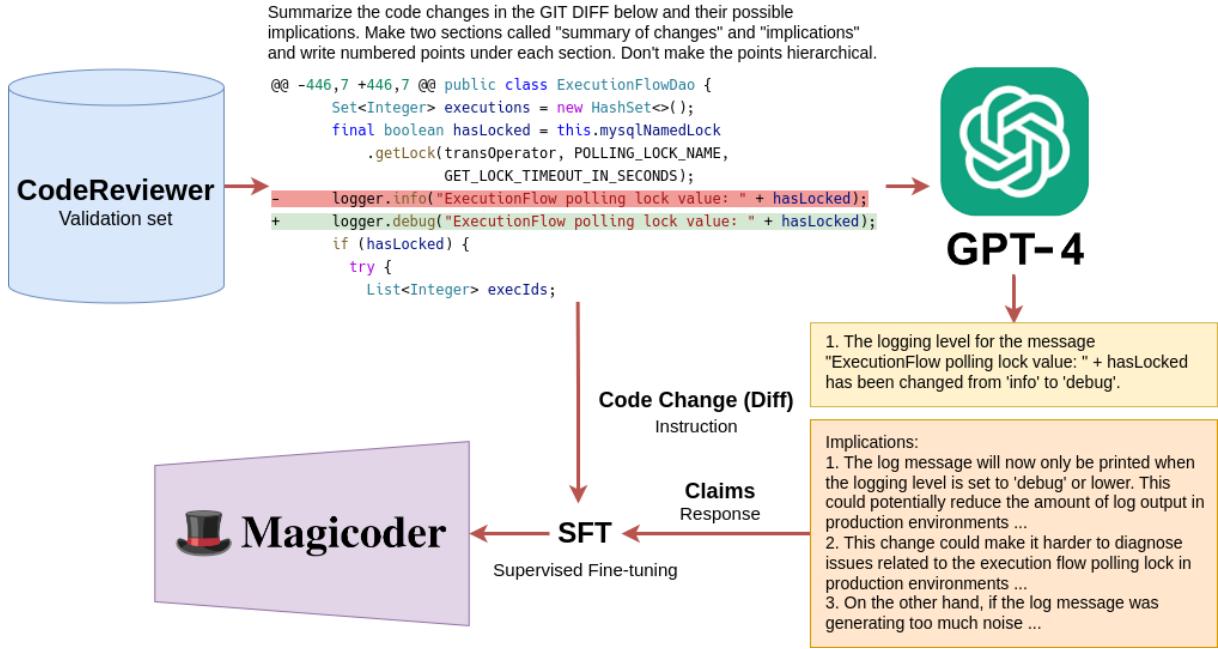


Figure 3: Supervised fine-tuning pipeline for training Magicoder-6.7B for claim generation. We generate synthetic data by using GPT-4 to generate claims for the code changes in CodeReviewer validation set.

Review	Missing Context
Don't redefine, just import the existing one in <code>cmdline.py</code> . :)	Folder structure, Codebase organization
I think we can remove this function, right? (duplicate with <code>ses_starter.py</code> )	Folder structure, Codebase organization
MPRester(os.environ(["MP_API_KEY"]) can be left simply as MPRester() and it will pick up the API key from the environment. What ...	Class definition, Environment variables

Table 5: Ground truth reviews in the automatically mined CodeReviewer data that assume contextual information about the code base not available in the dataset

2023). CodeBLEU extends BLEU by incorporating code structure through dataflow and syntax match between generated code and references, while CrystalBLEU filters out trivially shared n-grams. RUBY incorporates the distance between the syntax tree and program dependency graph of references and generated code. CodeBERTScore extends the embedding-based BERTScore by replacing BERT with a pre-trained CodeBERT (Feng et al., 2020)

model. However prior studies have shown metrics like BLEU to have low validity (overlap with human judgment) and reliability for text generation (Reiter, 2018), code generation (Evtikiev et al., 2023), and code migration (Tran et al., 2019). However, metrics like ROUGE, BERTScore, and CodeBERTScore all have a notion of precision, recall, and f-score which is captured by conciseness, comprehensiveness, and relevance respectively.

## B.2 Code Review Automation

Due to the high time and resource demands of code review automated approaches have gained popularity (Yang et al., 2024). Pornprasit and Tantithamthavorn (2024); Lu et al. (2023); Dong-Kyu (2024); Fan et al. (2024); Yu et al. (2024) propose fine-tuning and prompt engineering approaches to leverage LLMs for code review and code-change related tasks. Frömmgen et al. (2024); Rahman et al. (2024) propose methods for code refactoring based on review comments. Vijayvergiya et al. (2024) propose the detection of “best practice violations”. Lin et al. (2024) propose oversampling reviews from experienced reviewers as a proxy of review quality improving informativeness and correctness of generated reviews. Rasheed et al. (2024) propose an LLM agent for code review and code smell detection.

### B.3 Comparing with Review Usefulness/Presentation Style Methods

Our approach evaluates review informativeness and content related to issues such as code smells compared to prior work which has focused more on linguistic features, style, and presentation. (Yang et al., 2023) proposes EvaCRC an automated approach for review quality evaluation by categorizing reviews along quality attributes like emotion, question, suggestion, and evaluation and rating quality along those attributes on a four-tier grading scale. While their conceptualization is similar to ours, they focus on quality attributes related to review style and understandability rather than the actual content. Additionally, they don’t ground their assessments for these dimensions into a list of claims and issues which makes it harder to understand the scores. Rahman et al. (2017) propose RevHelper as an approach for predicting the usefulness of reviews based on a mixture of three reviewer expertise (e.g. Code Authorship) and five textual features: stop word ratio, reading ease, question ratio, and conceptual similarity. Most of the textual features identified by RevHelper deal with review style and presentation, and as pointed out in section 5.4 some of them like code elements could lead to overestimation of review quality. While the notion of conceptual similarity comes close to the conceptualization of our *Rel* score their operationalization of it is very different from ours. They directly measure similarity over lines of the code change and review comments. We believe this is redundant with the notion of preferring reviews that reference code elements and can be gamed by review generation models that reference code snippets incorrectly. Ochodek et al. (2022) develop CommentBERT an approach for automatically classifying review comments based on their purpose and the type of change required using a taxonomy of comments with 12 categories with dimensions like code\_design, code\_logic, etc. While they do classify the review content, they don’t look at what makes them useful.

### B.4 Code Smell Detection

The problem of detecting “code smells” or symptoms of design flaws and bad practices (also called *anti-patterns*) has been traditionally tackled by analysis-based approaches (Tsantalis et al., 2008; Paiva et al., 2017; Liu and Zhang, 2017). Recent work has explored learning-based methods for po-

tentially more nuanced detection of code smells. Sandouka and Aljamaan (2023) create a dataset of 1k Python code smells like “Long Method” and “Large Class” to train traditional ML models like random forests. Sharma et al. (2021) leverage deep learning models like autoencoders and transfer learning for adapting to unseen programming languages. Liu et al. (2024) propose a prompting-based approach for “Long Method” and “Long Parameter List” code smells in Java. Rasheed et al. (2024) create an LLM agent to detect code smells in repositories. However, these approaches are limited in the types of smells they target, training data requirements (Zhang et al., 2024), or lack comprehensive evaluation. Also, code smells differ across programming languages (Abidi et al., 2019), and transfer learning approaches can only be leveraged for similar languages (Sharma et al., 2021). Due to these limitations of learning-based methods we use traditional language-specific code analysis tools.

## C Method Details

This appendix contains additional details on the implementation of our CRScore metric.

### C.1 Distribution of Sentence Similarity Scores

We plot the histogram of values of the sentence similarity scores in Figure 5 showing a roughly normal distribution. We also plot the quantile-quantile (Q-Q) plot in Figure 6 that compares the quantile of a normal distribution with the empirically observed distribution of sentence similarity scores. Ideally, the Q-Q plot should be a straight line (shown in red) but we observe deviation towards really high ranges among the actual values (shown in blue). Due to computational constraints, these plots are constructed out of a randomly sampled subset of 100k similarity scores from the original 100M+ sentence similarity scores computed from the CodeReviewer test set review pairs.

### C.2 Code Smell Detection Details

In this section, we cover some of the details of the code smell detectors used in this study.

#### C.2.1 Class Cohesion

Class cohesion captures the degree to which the elements of a class belong together (Contributors, 2019a). In other words, cohesion measures the strength of the relationship between pieces of functionality (attributes and methods) within a given

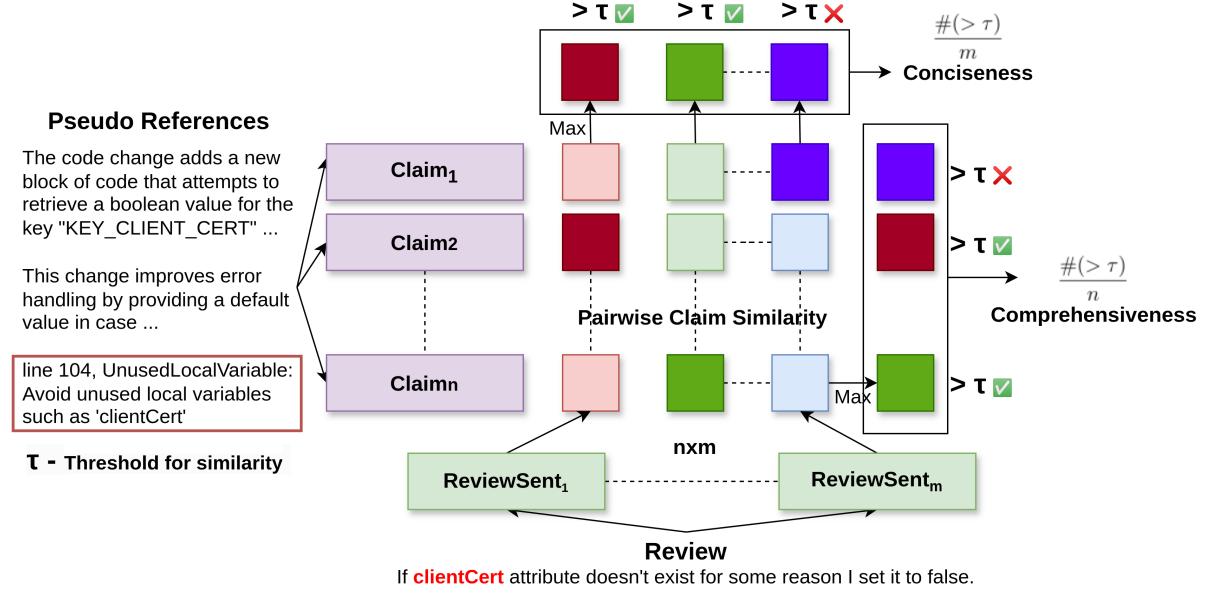


Figure 4: This figure shows how semantic textual similarity (STS) is used to measure the coverage of pseudo-references by the review sentences. We compute pairwise semantic similarities between all pseudo references and review sentences and employ a threshold to compute comprehensiveness as the fraction of pseudo references for which at least one review sentence has higher similarity than the threshold. Meanwhile, conciseness is the fraction of review sentences which have higher similarity than the threshold with any pseudo reference.

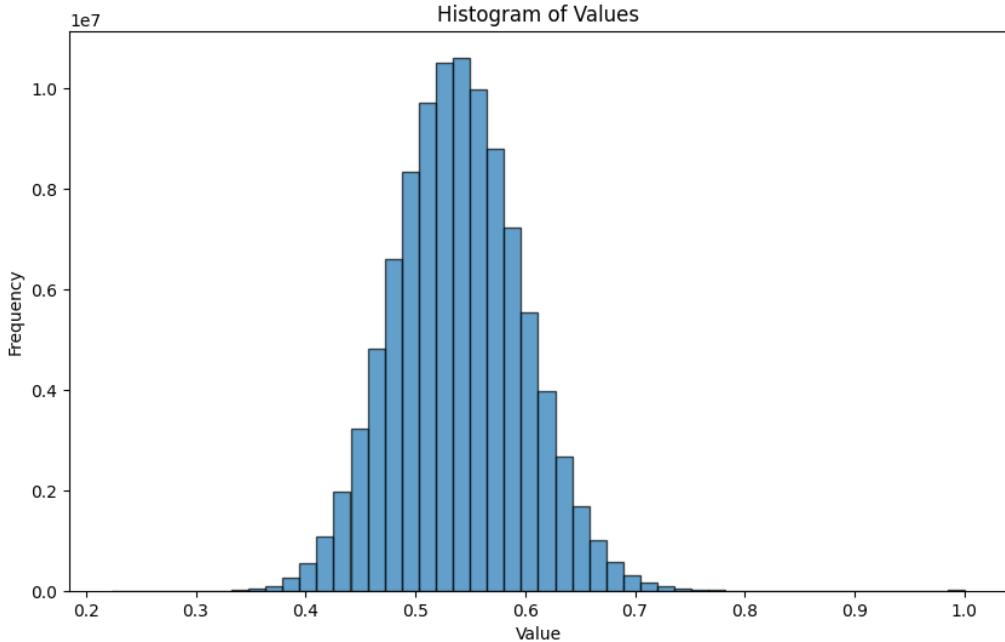


Figure 5: Histogram of sentence similarity of randomly sampled 100K sentence pairs from the CodeReviewer test set showing the scores are roughly normally distributed, justifying the usage of the 5-sigma rule for coming up with the threshold of 0.85 for high similarity used in metric computation.

class. For example, in highly cohesive classes functionality is strongly related and methods and attributes are more co-dependant and hang together as a logical whole (mschwager, 2016).

### C.2.2 Cyclomatic Complexity

Cyclomatic complexity is a software metric used to indicate the complexity of a program (Contributors, 2019b). It is a quantitative measure of the number

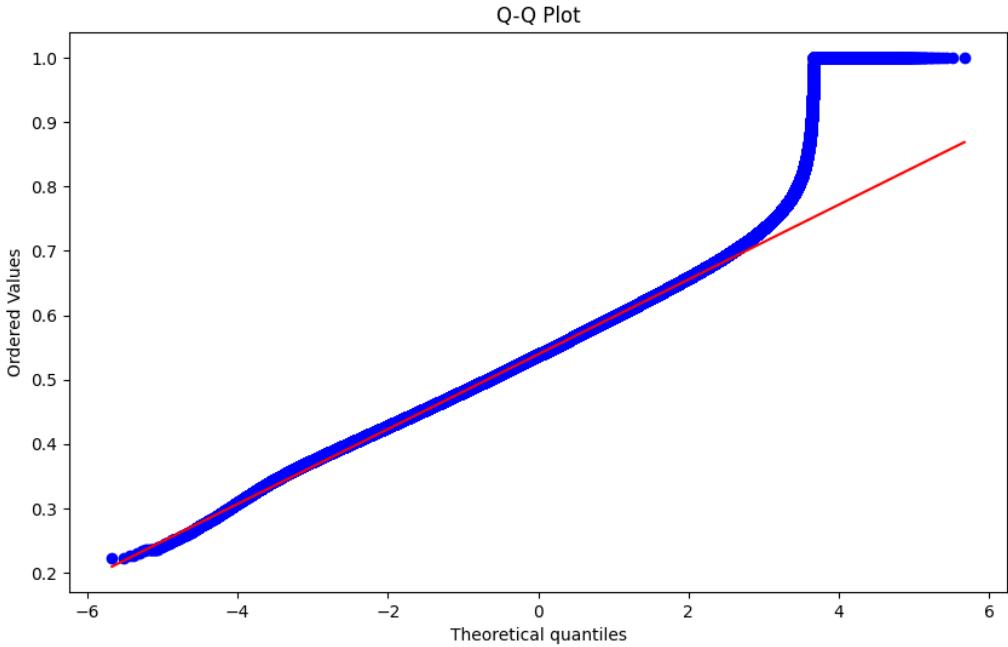


Figure 6: Q-Q plot comparing quantiles of empirically observed sentence similarity scores computed over 100K sentence pairs from the CodeReviewer test set showing the theoretical quantiles match a normal distribution except for really high values. The discrepancy seen here is likely due to the random sample being a smaller subset of the whole 100M+ sentence pairs for which we compute similarities.

of linearly independent paths through a program’s source code.

Popular tools like Radon<sup>8</sup>, a cyclomatic complexity computation tool for Python often resort to a rank-based system that categorizes code blocks based on their cyclomatic complexity and associates them with letter grades as shown in Table 6. Here “A” is the best grade and “F” is the worst grade. Code blocks are flagged for code smells if they have a complexity higher than or equal to “C” in Table 6.

CC Score	Rank	Risk
1-5	A	low - simple block
6-10	B	low - well structured and stable block
11-20	C	moderate - slightly complex block
21-30	D	more than moderate - more complex block
31-40	E	high - complex block, alarming
41+	F	very high - error-prone, unstable block

Table 6: Code complexity ranks, associated score thresholds, and descriptions of the potential risks.

<sup>8</sup><https://pypi.org/project/radon/>

### C.3 Hyperparameters for Training Pseudo-Reference Generation Pipeline

We train the ise-uiuc/Magicoder-S-DS-6.7B, with flash attention, random seed of 42, evaluation every 100 steps (eval\_steps), max length padding, train-val split of 0.1 (10% data used for validation), batch size of 2, maximum training sequence length of 1500 and 5 training epochs. The training is done on a single A100, 80 GB GPU using the Magicoder training script<sup>9</sup>.

## D Experimental Details

Further experimental details like guidelines for annotation of pseudo-reference accuracy, review quality, etc.

### D.1 Codebook for Rating Pseudo-Reference Quality

The raters are shown model-generated pseudo references (claims and implications) about the “diff”, which either describe the changes or speculate about potential implications. The claims are statements about the diff related to what changes took

<sup>9</sup><https://github.com/ise-uiuc/magicoder/blob/main/src/magicoder/train.py>

place, while the implications cover the effects of the changes or their interpretations like whether they implement a new functionality or even the potential intent of the developers. Additionally, the raters are also given source files as context, including the versions of the file before and after the code change captured by the diff. The raters are expected to refer to them if they need more information than just the diff to judge the accuracy of the pseudo-references.

Given these inputs, the raters are supposed to code each pseudo-reference as shown in Table 8. In further analysis, we excluded the ambiguous claims (code 2) because they were rarely encountered.

## D.2 Rating Review Quality using Pseudo-References

The raters are given pseudo-references corresponding to the diff but unlike the pseudo-reference quality annotations, they also include issues/smells detected by the static analysis tools. The issues span formatting issues, bad programming practices, or more abstract patterns known as “code smells”. Code smells are heuristics or code characteristics associated with deeper problems concerning system design. It is important to note that they are not bugs, but rather subjective principles that vary across programming languages, developers, teams, etc. Given the diff and the pseudo-references, the raters evaluate the quality of the review along the three dimensions — comprehensiveness, conciseness, and relevance. The raters are again given access to the source files for context and are asked to rate the quality of 10 reviews per code change that are generated by a diverse set of review generation systems and one of them is also the ground truth review from the CodeReviewer dataset.

The rater’s task is to first go through the list of pseudo-references and rate their necessity with respect to the code change. This stage is meant to remove any pseudo-references that are unnecessary or redundant with respect to the whole set. The second stage of the annotation is to link/associate various pseudo-references to each of the 10 reviews based on which of them are addressed in the review. Finally, the raters assign a score on a Likert scale of 1 to 5 to each review for each dimension. Some rules of thumb for assigning each score for each dimension are given in Table 7.

Raters are also given a helpful mental framework to help with the process of linking pseudo-

references to the reviews. To explain what it means for a review to address a pseudo-reference, we give an analogy to aspect-based product reviews. The list/set of claims is similar to a list/set of product aspect descriptions, while the code review comment is similar to a product review. For example, for the product description and review below:

### Description of a newer version of a Phone:

- The new phone improves battery life by 50% (battery life)

- The new screen has a higher resolution (screen)

**Review:** The new screen quality is great but the battery runs out quickly.

The review here is talking about both the screen and battery life aspects of the product so you can say it is addressing both aspects (or claims).

Now we can consider an example from the domain of code review:

### Pseudo-references for code change:

- The code change is in the logging of errors in the response handler. (error logging)

- The formatting of the error message has been changed. (formatting change of error message)

- The previous formatting used ‘%s %’ at the end of the error message, which was removed in the updated code. (% at the end of the error message)

- The change in the formatting of the error message will affect the way errors are logged and displayed. (effect of formatting on error logging)

**Code review:** The %s in the error message is redundant, and the indent=4 in json.dumps is unnecessary.

The review talks about the third claim by mentioning the %s style string at the end of the error logging.

## D.3 Simple Baseline Implementation Details

We create two simple baselines for code review generation a BM-25, kNN style retriever-based approach, and a seq2seq LSTM style model. The implementation details of both approaches are described below:

**BM-25 retriever:** The BM-25 retriever retrieves a relevant review by matching the closest code change from the CodeReviewer train set to the code change to be reviewed from the test set. For an efficient implementation, we create an inverted index from all the code changes in the train set using the Lucene searcher ([luc](#)) class from Pyserini<sup>10</sup>.

---

<sup>10</sup><https://pypi.org/project/pyserini/>

Dimension	Score	Rule of thumb
Conciseness	1	none of the review is related to the claims+issues.
	2	some of the review is related to the claims+issues
	3	roughly half of the review is related to the claims+issues
	4	most of the review is related to the claims+issues
	5	basically the whole review is related to the claims+issues
Comprehensiveness	1	whole review doesn't cover any of the claims+issues
	2	review covers at least 1 claim or issue
	3	review covers roughly cover half the claims+issues
	4	review covers more than half/most of the claims+issues
	5	review covers practically all the claims+issues
Relevance		relevance score must be between conciseness and comprehensiveness scores if one of the two dimensions is "limiting", i.e. has a very low score or poor quality for a given review then the relevance score should be biased/limited by it.  E.g. a highly concise review but with very low comprehensiveness should have a relevance score close to the comprehensiveness

Table 7: Some rules of thumb for scoring the quality of reviews for each dimension. These rules are meant as guidelines to calibrate multiple annotators and reduce the impact of learning effects in the early stages of annotation.

Code	Description	Examples	Explanation
1	Correct: You can find concrete evidence to validate or confirm a claim (either in the diff, the context/source files, or by looking up domain-specific knowledge on the web)	Any existing calls to the ‘push’ function will need to be updated to include the new ‘hash’ parameter. This could potentially break compatibility with older code.	This is true since if the parameters are passed by value and there is a parameter after the hash then its value will be accidentally passed as the hash
0	Incorrect: You can find concrete evidence to contradict a claim (either in the diff, the context/source files, or looking up domain-specific knowledge on the web)	The code changes involve the modification of the parameters passed to the ScalarSpaceEncoder function.	False because ScalarSpaceEncoder is a class and not a function.
-1	Unverifiable: You can’t find evidence to confirm or contradict a claim (even after looking at the diff, source files, or looking up domain-specific knowledge on the web)	The changes could potentially affect the performance of the code as the order of the arguments does not matter any longer.	For this claim, the arguments were being passed by keyword so the order didn’t matter for functional correctness but we can’t comment on the performance
2	Ambiguous: The claim is underspecified and has multiple interpretations making it hard to determine what is to be tested/validated	The function will now only work with valid elements, preventing potential issues down the line.	It is underspecified (unclear) what “valid elements” means.

Table 8: Guidelines for coding accuracy of pseudo-references with example pseudo-references falling within each category and explanations for why they fall in that category.

**LSTM reviewer:** We train a single hidden layer encoder-decoder seq2seq LSTM model with Bahdanau attention (Bahdanau, 2014) from scratch on the CodeReviewer training data. We train it with an Adam optimizer, and negative log-likelihood loss for about 10 epochs, saving the model with the least loss on the CodeReviewer validation set.

#### D.4 LLM-as-a-judge Prompting

The system prompt is as follows:

You are a highly skilled software engineer who has a lot of experience reviewing code changes. Your task is to rate the relevance of any given code change

The task-specific, review comment evaluation

prompt is as follows:

<p>TASK PROMPT: You will be asked to rate the relevance of reviews for given Python, Java, or Javascript code changes. A relevant review is one which is both concise and comprehensive. A concise review contains very little text not related to the code change. A comprehensive review contains all the information about a code change that should be covered by a review. A relevant review is comprehensive while being concise.</p> <p>Now look at the {lang} code change and review below and score the relevance of the review on a scale of 1 to 5</p> <p>Code Change: {code_change}</p> <p>Review: {review}</p> <p>Your score:</p>
--

## D.5 Dataset Statistics

We perform all our annotations on the test set of the CodeReviewer dataset, which contains 10169 samples in total and is publicly available with the Apache 2.0 license<sup>11</sup>. The license is permissive and allows us to use the dataset for research purposes. We randomly sampled 300 samples, with 100 samples across Python, Java, and Javascript – the languages we have code smell detectors. We found a few mislabeled code changes for each language that belonged to a different language, which we discarded. This led to a final dataset of 99, 98, and 96 code changes for Python, Java, and Javascript respectively. Also, we end up with 416, 416, and 399 claims that are evaluated as correct, incorrect, or unverifiable. For the review dataset, we annotate 2.9k reviews (9 systems + ground truth) for each code change from the previous annotation stage. For all the correlation experiments we exclude the ground truth review annotations around (300), giving us a dataset of roughly 2.6k reviews corresponding to the automated review generation systems. As mentioned before we excluded the ground truth while computing correlations and system rankings because reference-based metrics by definition would favor (and perfectly rate) the ground truth, unfairly disadvantaging them for computing correlations.

## D.6 Computational Budget

For running all the systems mentioned in section 4.4 we utilize an 80 GB A100 for the LLM systems, running them on the 10k CodeReviewer

test instances. For running GPT-3.5 (unfortunately no longer publicly available) we utilized around 10k API calls. For the synthetic dataset creation for Magicoder for pseudo-reference generation modeling, we called the GPT-4 model (gpt-4-0613) for 1000 CodeReviewer validation instances. For the Magicoder model training, we also use an A100 GPU, for 5 epochs (or approximately 8-10 hrs).

## D.7 Annotator Information

We hired a graduate student and an undergraduate student with experience in writing Java, Python, and Javascript code. The annotation process took several weeks with additional training provided about code review quality, especially about code smells. The annotation guidelines were also iteratively improved till the high level of reliability reported here was achieved. For the compensation, the annotators took up this project for research credits and thus weren't compensated monetarily. They are also co-authors of this project. They were extensively briefed about the research project and how their data would be used as preference data for comparing evaluation metrics and would be publicly released after anonymization.

## D.8 Ethics Review for Data Collection

The data collection protocol used in this work was approved by the institutional review board (IRB) of the university the authors are affiliated with.

## D.9 Does Including the Pseudo-References in the Review Quality Dimension Rating Lead to Biased Annotations?

While at the first glance it might seem that giving the annotators access to the pseudo-reference could bias their operationalization of the review quality dimensions in the favor of CRScore, we believe this is not likely. Since this study uses a modified set of pseudo-references which is edited by the human annotators to remove incorrect claims and add correct or add missing claims, the set of pseudo-references used by the human annotators is actually different from the automatically generated pseudo-references used by CRScore. In fact the results demonstrate that the Laaj-GPT metric which doesn't rely on the pseudo-references can obtain a greater correlation than CRScore with the human review quality dimension annotations despite not having access to the pseudo-references. This demonstrates that the annotations aren't biased towards CRScore. Finally the pseudo-references play

<sup>11</sup><https://huggingface.co/microsoft/codereviewer>

a crucial role in helping the annotators concretely assess the dimension of comprehensiveness, since judging the coverage of a review is very hard without a list of things that it should cover. Thus we believe it contributes to the high reliability of the annotations.

## E Additional Results and Analysis

### E.1 Finding Failure Cases of CRScore

We divide the scores spanned by our metric into 5 equally sized bins (Q1, Q2, Q3, and Q4 being the quantiles) and then find reviews where the metric underestimates the Rel value (value less than Q1 but a human rating of 5) and overestimates the Rel value (value greater than Q4 but a human rating of 1). We find 16 (0.61%) and 98 (3.74%) cases respectively for underestimation and overestimation, moreover, without these cases, our metric ( $\tau_{best}$ ) attains correlations of  $\tau = 0.5462$  and  $r_s = 0.6431$  and  $\tau = -0.5403$  and  $r_s = 0.6131$  for these cases, indicating their influence despite being less than 4% of the data.

### E.2 Impact of Varying STS Threshold ( $\tau$ )

Metric	Human Annotations					
	Con ( $\tau$ )	Comp ( $\tau$ )	Rel ( $\tau$ )	Con ( $r_s$ )	Comp ( $r_s$ )	Rel ( $r_s$ )
BLEU	0.0306	-0.0227	0.001	0.0358	-0.0293	-0.0001
BLEU (without stopwords)	0.0632	0.0221	0.0425	0.0776	0.0293	0.0542
BERTScore	0.1035	0.0622	0.081	0.1378	0.0813	1.083
Normalized Edit Distance	-0.0146	0.0443	0.0193	-0.0218	0.0584	0.0249
ROUGE-L	0.0921	0.0577	0.0757	0.1173	0.0758	0.0989
F-measure						
chrF	0.1236	0.1431	0.1484	0.1628	0.1874	0.1966
chrF++	0.1294	0.1496	0.1555	0.1707	0.1959	0.2057
Con ( $\tau_{GT}$ ) ours	0.4168			0.475		
Comp ( $\tau_{GT}$ ) ours		<b>0.4982</b>				<b>0.5832</b>
Rel ( $\tau_{GT}$ ) ours			0.4437			<b>0.5405</b>
Con ( $\tau_{best}$ ) ours	<b>0.4491</b>			<b>0.5049</b>		
Comp ( $\tau_{best}$ ) ours		0.4974				0.5754
Rel ( $\tau_{best}$ ) ours			<b>0.4567</b>			0.5431

Table 9: Kendall and Spearman rank correlations between human annotation for all dimensions: conciseness (Con), comprehensiveness (Comp), and relevance (Rel) and all metrics including our proposed Con, Comp and Rel metrics for both threshold values  $\tau_{GT}$  and  $\tau_{best}$ .

Model	Human Annotations			Our Metric		
	Con	Comp	Rel	Con	Comp	Rel
BM-25 retriever	<u>0.0301</u>	<u>0.0112</u>	<u>0.0163</u>	<u>0</u>	<u>0</u>	<u>0</u>
LSTM	0.1048	0.0361	0.0515	0.0372	0.0123	0.0179
CodeReviewer	0.5146	0.1692	0.2311	0.4639	0.2413	0.2974
Stable-Code-3B	0.3222	0.1383	0.1718	0.3353	0.2091	0.2319
DeepSeekCoder-6.7B-Instruct	0.6108	0.3153	0.3797	0.5037	0.3989	0.4043
Magicoder-S-DS-6.7B	0.4915	0.3127	0.3351	0.4381	0.4746	0.4032
LLaMA-3-8B-Instruct	0.6091	0.3368	0.4046	0.3503	0.3967	0.3404
CodeLLaMA-13B	0.1985	0.1546	0.1564	0.2493	0.2528	0.2309
GPT-3.5	<b>0.6564</b>	<b>0.4132</b>	<b>0.4759</b>	<b>0.5622</b>	<b>0.6301</b>	<b>0.5507</b>
Ground Truth	0.5129	0.219	0.2819	0.251	0.1598	0.1741

Table 10: Comparison of our proposed metrics with human annotations for the review quality dimensions. We normalize the human annotated Likert scores per dimension ( $DimVal$ ) from 1 to 5 to 0 to 1 ( $NormDimVal$ ) as:  $NormDimVal = \frac{DimVal - 1}{4}$ . The results are reported on the subset of 300 annotated CodeReviewer test instances. We highlight the best-performing model (bold) and the worst-performing model (underlined) according to the human annotations and our metric.

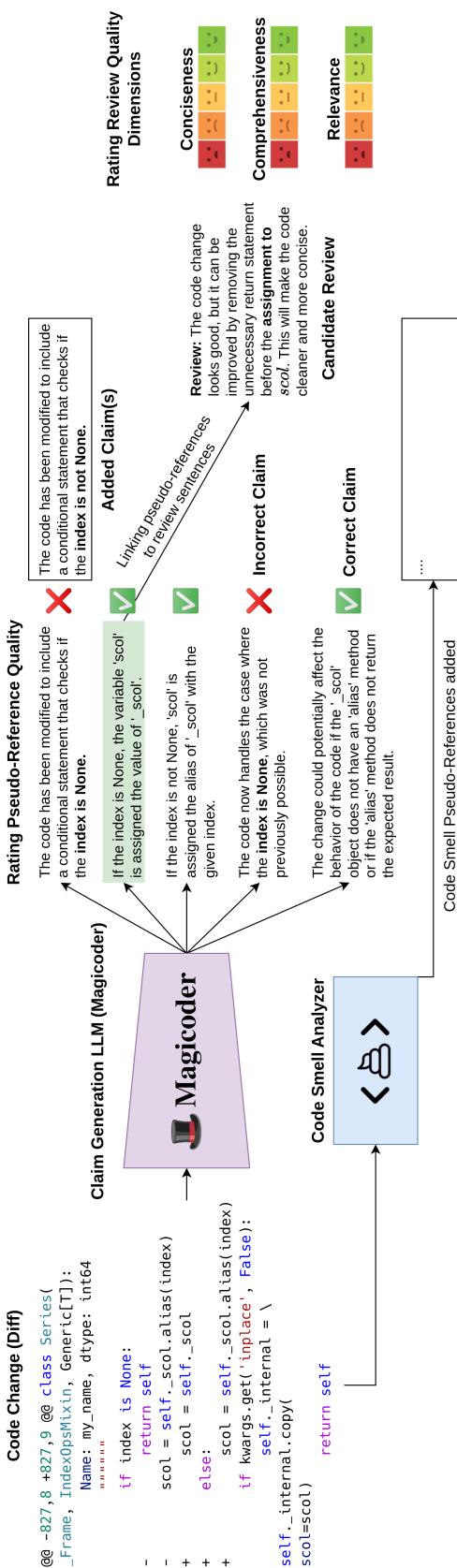


Figure 7: The stages of the annotation pipeline. For the first stage — Rating Pseudo-Reference Quality the annotators mark correct, incorrect, and unverifiable claims, while also adding any missing claims. In the second stage — Rating Review Quality Dimensions the annotators are given the updated set of pseudo-references from the first stage along with any pseudo-references generated by the code smell analysis tools to rate candidate reviews on the quality dimensions — conciseness, comprehensiveness, and relevance on a Likert scale using the pseudo-references.

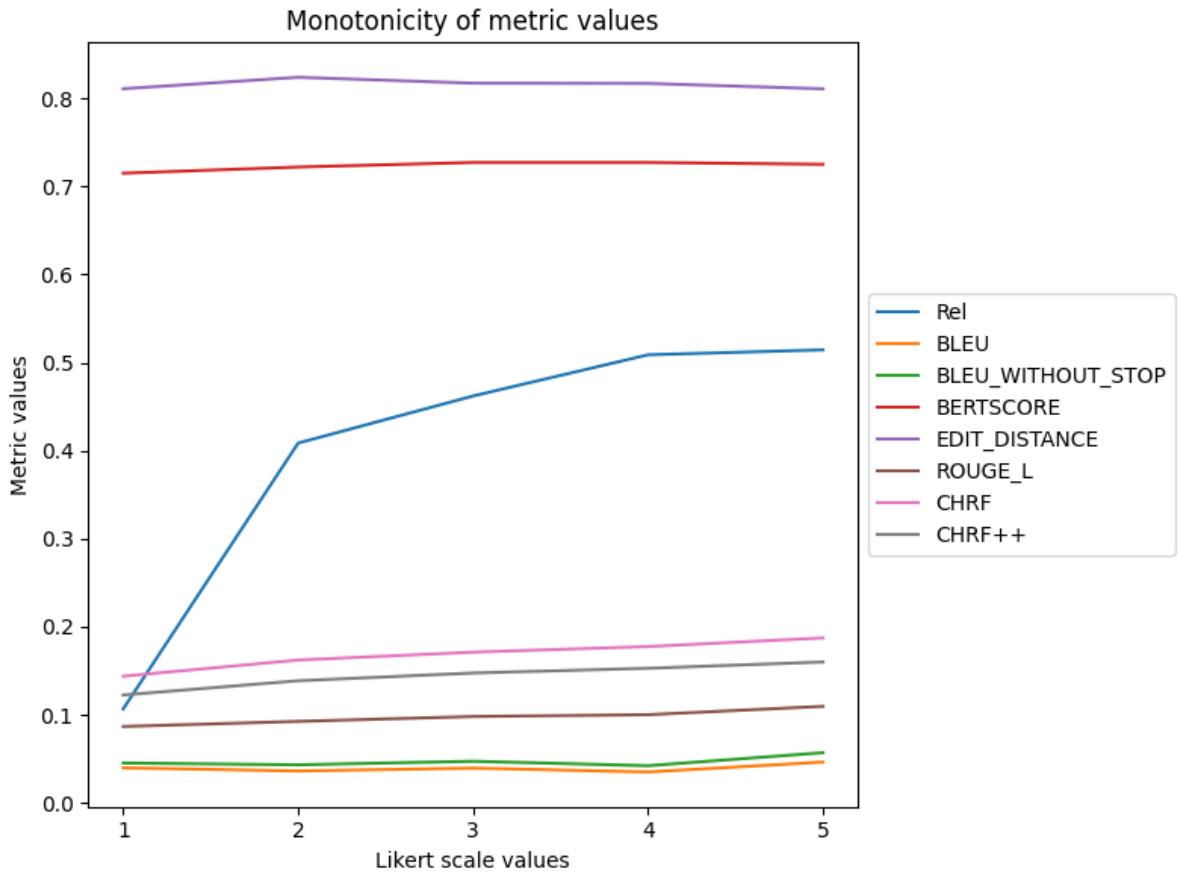


Figure 8: Monotonicity of metric values for reviews of various quality based on the Likert scale human annotations. The Rel metric exhibits the most variation across reviews of different quality while the other metrics have flat plots indicating that they fail to distinguish between reviews of varying quality meaningfully

Metric	Python		Java		Javascript	
	$\tau$	$r_s$	$\tau$	$r_s$	$\tau$	$r_s$
BLEU	-0.0017	-0.0006	-0.0101	-0.015	0.007	0.009
BLEU (without stopwords)	0.0532	0.0665	0.035	0.0442	0.0313	0.0405
BERTScore	0.0696	0.0942	0.0808	0.1081	0.0846	0.1128
Normalized Edit Distance	0.0367	0.046	0.0161	0.0217	0.0109	0.0129
ROUGE-L	0.0935	0.1201	0.0553	0.0724	0.075	0.0981
F-measure	0.1309	0.1746	0.1573	0.2077	0.1497	0.1982
chrF	0.1387	0.1848	0.1676	0.2208	0.1531	0.2021
<b>Rel (<math>\tau_{GT}</math>) (Ours)</b>	0.4742	0.5788	0.3889	0.4738	0.473	0.5738
<b>Rel (<math>\tau_{best}</math>) (Ours)</b>	0.4746	0.5666	0.4093	0.4849	0.4904	0.5816

Table 11: Comparing Kendall-Tau ( $\tau$ ) and Spearman Rank ( $r_s$ ) correlation of reference-based evaluation metrics across each language annotated (Python, Java and Javascript) and our reference-free relevance score (Rel (F)) with human annotations for relevance. Correlations that are not statistically significant (p-value < 0.05) are grayed out.

Error Type	Description	Incorrect Reference	Corrected Reference	Frequency (%)
Knowledge Error	Pseudo-reference exhibits incorrect domain knowledge	The 'optparse' module is being imported with a comment indicating that it is being disabled due to its deprecation.	The 'optparse' module is being imported with a comment disabling a pylint deprecated-module warning	6.52
Reasoning Error	Pseudo-reference exhibits wrong logic applied by the pseudo-reference generator	Now, the 'can_edit_record' variable is only true if the function 'check_user_can_edit_record' returns true and the 'format' variable does not start with 't' (in lowercase).	Now, the 'can_edit_record' variable is only true if the function 'check_user_can_edit_record' returns true and the 'format' variable does not start with 't' (in any lower or uppercase).	10.87
Localization Error	The pseudo-reference generator misunderstands where a code change has taken place	The assertion in the test method "test_idxmapping_add_dimension" has been modified.	The assertion in the test method "test_idxmapping_redim" has been modified.	4.35
Over-generalization Error	The pseudo-reference generator makes an incorrect assumption/generalization from the code change	The addition of these import statements suggests that the code in this file will now be using the ResLayer and SimplifiedBasicBlock classes from the mmdet.models.utils package.		19.57
Comprehension Error	The pseudo-reference seems to "misread" the code change (like thinking removed lines are added, etc.)	The import statement for 'filter', 'range', and 'zip' has been moved from 'scapy.modules.six.moves' to 'scapy.modules.six'.	'filter' is now also imported from 'scapy.modules.six.moves'	58.7

Table 12: The various types of errors identified, their descriptions and examples (pseudo-references before and after correction of the error are shown) as well as relative frequencies as percentages are shown here. For this analysis, we annotated 46 erroneous pseudo-references

Model	BLEU		BERTScore	Norm. Edit Distance	ROUGE L f-score		chrF	chrF++	Con ( $\tau_{best}$ )	Comp ( $\tau_{best}$ )	Rel ( $\tau_{best}$ )
	BLEU	(without stop)			chrF	chrF++					
BM-25 kNN	0.036	0.043	0.718	0.805	0.069	0.134	0.113	0.002	0.001	0.001	0.001
LSTM	0.047	0.051	0.716	0.802	0.102	0.12	0.105	0.02	0.006	0.009	
Transformer <sup>†</sup>	0.048										
T5 <sup>†</sup>	0.044										
CodeT5 <sup>†</sup>	0.048										
CodeReviewer	0.054	<b>0.071</b>	0.718	0.811	0.102	0.116	0.1	0.412	0.208	0.26	
Stable-Code-Instruct-3B	0.042	0.04	0.733	0.784	0.091	0.16	0.133	0.34	0.199	0.228	
Magicoder-S-DS-6.7B	0.035	0.041	0.72	0.815	0.101	0.175	0.151	0.445	0.491	0.42	
DeepSeekCoder-Instruct-6.7B	0.045	0.054	<b>0.734</b>	0.782	<b>0.112</b>	0.183	0.157	0.513	0.418	0.422	
CodeLLaMA-Instruct-7B	0.023	0.026	0.71	0.843	0.071	0.171	0.145	0.156	0.151	0.14	
Llama-3-8B-Instruct	0.014	0.016	0.699	<b>0.898</b>	0.058	0.14	0.122	0.347	0.425	0.352	
LLama-Reviewer <sup>†</sup>	<b>0.057</b>										
CodeLLaMA-Instruct-13B	0.025	0.029	0.715	0.839	0.079	0.179	0.152	0.274	0.272	0.253	
GPT-3.5-Turbo	0.037	0.044	<b>0.734</b>	0.812	0.1	<b>0.2</b>	<b>0.171</b>	<b>0.563</b>	<b>0.635</b>	<b>0.558</b>	

Table 13: Results of all eval. metrics and models on the entire test set. All metrics have been normalized to be between 0 and 1. <sup>†</sup> signifies reported scores. The reference based metrics have a very narrow range of values.

Smell Name	Description
Long method	There exist methods with too many lines (more lines than a set threshold).
Long parameter list	There exist methods with more than “n” parameters (n = 6 is used in this study).
Long branch	When conditional statement branches extend too long or are too nested.
Many attributes	When a single class has too many methods or attributes.
Many methods	When a single class has too many methods.
Shotgun surgery	When a single functionality is fragmented across various classes.
Class cohesion	There are some classes with low cohesion ( <a href="#">C.2.1</a> ).
Code complexity	The code includes blocks with cyclomatic complexity (section <a href="#">C.2.2</a> ) of rank-C or worse (moderate to slightly complex blocks). Please read through the cyclomatic complexity and ranks section for more details.
Long lambda	The code includes lambda functions that exceed a threshold on length (number of characters).
Long list comprehension	The code includes list comprehensions that exceed a threshold on length (number of characters).

Table 14: Python code smells detected by the PyScent code smell static analysis tool

Rank	Human Annotated Relevance	Rel (ours)	chrF++ (best reference based metric)
1	GPT-3.5	GPT-3.5	GPT-3.5
2	LLAMA-3-8B-Instruct	DeepSeekCoder-6.7B-Instruct	DeepSeekCoder-6.7B-Instruct
3	DeepSeekCoder-6.7B-Instruct	Magicoder-S-DS-6.7B	CodeLLaMA-13B
4	Magicoder-S-DS-6.7B	LLaMA-3-8B-Instruct	Magicoder-S-DS-6.7B
5	CodeReviewer	CodeReviewer	Stable-Code-3B
6	Stable-Code-3B	Stable-Code-3B	LLaMA-3-8B-Instruct
7	CodeLLaMA-13B	CodeLLaMA-13B	BM-25 retriever
8	LSTM	LSTM	CodeReviewer
9	BM-25 retriever	BM-25 retriever	LSTM

Table 15: Rankings of the systems over the 300 human-annotated CodeReviewer instances according to the human-annotated relevance and our relevance metric. The rankings reveal that our method gets the rankings exactly right except for the LLaMA-3-8B-Instruct model (grayed out) which is ranked lower than Magicoder and DeepSeekCoder by our metric but preferred more by humans.

Language	Code Change	Claims	Review
Python	<pre> @@ -24,6 +24,7 @@ CREATE_TABLE = """     'complete_time' datetime DEFAULT NULL,     'status' enum('SUCCESS','RUNNING',     'FAILURE',     'PARTIAL_SUCCESS','TIMEOUT')     DEFAULT NULL,     + 'has_all_data' bool DEFAULT NULL,     'schema_version' varchar(255)     DEFAULT NULL,     'cycle_timestamp' varchar(255)     DEFAULT NULL,     PRIMARY KEY (`id`),   </pre>	<p>A new column named 'has_all_data' has been added to the table. This column is of boolean data type and allows NULL values by default.</p> <p>The addition of the 'has_all_data' column implies that there might be a need to track whether all the data has been processed or not in the table. This could be used for further data processing or analysis.</p>	<p><b>Magicoder:</b> This column is not being used anywhere in the codebase. It's a waste of space.</p>
Python	<pre> @@ -14,7 +14,9 @@ import abc import logging +import datetime import parameter +import target import warnings import traceback import pyparsing as pp   </pre>	<p>Two new import statements have been added to the code. The first one imports the datetime module, and the second one imports the target module.</p> <p>The addition of the datetime module suggests that the code may now involve operations related to date and time. This could be for logging purposes, tracking the execution time of the code, or handling dates/times in the program.</p> <p>The addition of the target module indicates that the code may now involve operations related to the target environment or system. This could be for interacting with the target system, or for handling target-specific tasks.</p>	<p><b>CodeReviewer:</b> Why do we need these imports?</p>

Table 16: Cases where our metric underestimates the relevance of a review by scoring it as zero while the human scores it as 5 (max relevance). We observed that these cases tend to have fewer claims associated with the code change, briefer reviews with few relevant tokens, and fewer inline code snippets.

Language	Code Change	Claims	Review
Python	<pre> @@ -971,7 +971,7 @@ class JobTask(BaseHadoopJobTask):     if self._module__ == '__main__':         d = pickle.dumps(self)         module_name = os.path.basename(             sys.argv[0]).rsplit('.', 1)[0]         - d = d.replace(b'c__main__', "(c" + module_name)         + d = d.replace(b'c__main__', b'c' + \                       module_name.encode('ascii'))         open(file_name, "wb").write(d)      else:   </pre>	<p>The code change is in the replacement of a part of the byte string 'c__main__', with 'c' followed by the encoded ASCII representation of 'module_name'.</p> <p>The original code was replacing '(c__main__' with '(c' followed by 'module_name'.</p> <p>The new code change corrects this by encoding 'module_name' to ASCII before replacing 'c__main__', with 'c' followed by the encoded 'module_name'.</p> <p>This change could potentially affect the behavior of the code if 'module_name' contains non-ASCII characters.</p>	<p><b>CodeReviewer:</b> Shouldn't this be 'module_name.encode('ascii')'?</p> <p><b>StableCode:</b> This seems unnecessary, the pickle module already encodes the module name.</p>

Table 17: Cases where our metric overestimates the relevance of a review by assigning it the maximum score while the human scores it at the lowest Likert score value of 1. We observed that these cases tend to have far more inline-code snippets than the rest of the data and underestimation cases.

<b>STS Threshold (<math>\tau</math>)</b>	$\tau$	$r_s$
$\tau=0.6$	0.3975	0.4874
$\tau=0.65$	0.4572	0.5512
$\tau_{GT}=0.6576$	0.4437	0.5405
$\tau=0.7$	0.4433	0.5407
$\tau_{best}=0.7314$	0.4567	0.5431
$\tau=0.75$	0.4473	0.5277
$\tau=0.8$	0.3946	0.4539

Table 18: Effect of varying STS threshold ( $\tau$ ) on the Kendal-Tau and Spearman Rank correlation of CRScore relevance measure (Rel) with human-annotated relevance. We note that our approach is robust to slight variations in the threshold making it robust.

Table 19