CoQuIR: A Comprehensive Benchmark for Code Quality-Aware Information Retrieval

Anonymous ACL submission

Abstract

Code retrieval is vital to modern software engineering as it boosts reuse and speeds up debugging. However, current benchmarks primarily emphasize functional relevance while neglecting code quality. To address this gap, we introduce CoQuIR, the first large-scale, multilingual benchmark specifically designed to evaluate quality-aware code retrieval across four critical dimensions: *correctness*, *efficiency*, *security*, and *maintainability*. CoQuIR includes fine-grained quality annotations over 42,725 queries and 134,907 code snippets in 11 programming languages. Evaluating 23 retrievers (both open-source and proprietary) reveals that even state-of-the-art models frequently fail to distinguish between buggy or insecure code and robust counterparts. We further investigate methods for explicitly training retrievers to recognize code quality, demonstrating that quality-aware metrics can be improved without loss of semantic relevance; downstream code generation benefits from these gains. CoQuIR underscores the importance of embedding quality signals into retrieval systems as a crucial component for more trustworthy developer tools.

1 Introduction

Code retrieval is a fundamental component of modern software engineering, supporting both human- and LLM-based code development and debugging (Di Grazia and Pradel, 2023; Li et al., 2024a; Luo et al., 2024; Yao et al., 2023; Wang et al., 2025). Existing benchmarks primarily inherit classical criteria from natural language (NL) retrieval, focusing on semantic or functional relevance, i.e., whether the retrieval code addresses the intent of a query. However, these benchmarks largely overlook critical, code-specific quality dimensions such as *correctness*, *efficiency*, *security*, and *maintainability* (Husain et al., 2019; Huang et al., 2021a; Khan et al., 2024b; Li et al., 2024b).

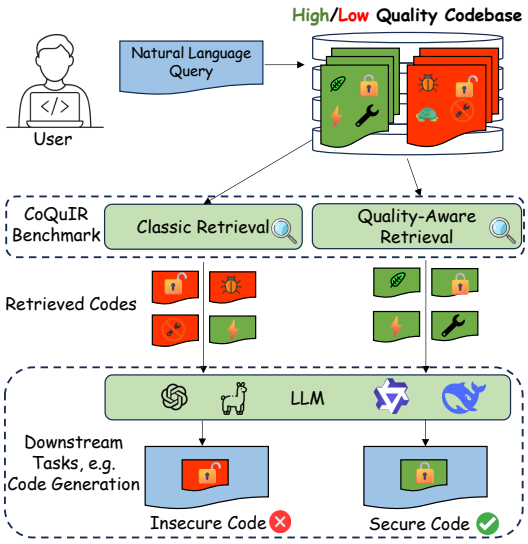


Figure 1: CoQuIR is primarily motivated by the need for retrievers to rank high-quality code above low-quality code, which can severely harm downstream generation.

Recent studies show that retrieval-augmented generation (RAG) is vulnerable to negative examples and knowledge base poisoning (Lin et al., 2025; Yang et al., 2025). As Figure 1 illustrates, functionally relevant code can still be low-quality and ranked highly for conventional retrievers. The code can include subtle bugs, deprecated APIs, or security flaws. When this code is incorporated into downstream tasks, it can propagate technical debt or introduce exploitable vulnerabilities.

The need for quality-aware retrieval is further magnified by practical constraints. (i) *Testing is costly*: the oracle problem frequently necessitates expensive human judgment or the construction of partial oracles, while flaky tests impose additional burdens on developer time and computational resources (Luo et al., 2014; Barr et al., 2014). (ii) *Static filtering incurs overhead*: scalable analyzers inevitably trade precision for recall, producing large numbers of false positives that engineers must manually triage (Johnson et al., 2013). (iii)

Scale and context complicate evaluation: millions of new snippets appear daily across public and private repositories, yet benchmarking their performance and quality under realistic conditions remains difficult. Even curated codebases such as CoIR (Li et al., 2024b) suffer from outdated APIs and inefficient implementations. Given these challenges in building a high-quality codebase and the importance of code quality for downstream tasks, we explore this question orthogonally:

> *Can code retrievers be designed to embed quality-awareness beyond relevance?*

To address this gap, we introduce **Co**de **Qu**ality-aware **I**nformation **R**etrieval (**CoQuIR**), a novel benchmark tailored for quality-aware code retrieval. CoQuIR spans four critical quality dimensions most relevant to practical development: (1) Correctness: whether code is bug-free; (2) Efficiency: whether it avoids unnecessary overhead; (3) Security: whether it prevents vulnerabilities; and (4) Maintainability: whether it relies on stable, recommended APIs. These dimensions are consistent with long-standing software quality models and standards (McCall et al., 1977; für Normung, 2011), ensuring both theoretical grounding and practical relevance. CoQuIR covers 11 widely used programming languages (PLs) and pairs NL queries with multiple code candidates that serve as quality-wise contrastive examples. It is constructed from high-quality code datasets originally curated for tasks such as code pretraining, vulnerability detection, and instruction tuning (Puri et al., 2021; Just et al., 2014; Bhandari et al., 2021; Wang et al., 2024a). All code snippets are drawn from real-world software projects to ensure practical relevance, with quality labels derived from well-documented annotation protocols, without reliance on LLM-based heuristic annotation.

We evaluate 23 retrieval models across six paradigms: (1) unsupervised retrievers (Izacard et al., 2022)); (2) supervised retrievers (Wang et al., 2022); (3) code-specific retrievers (Zhang et al., 2024b; Suresh et al., 2025a); (4) LLM-based retrievers (Wang et al., 2024b; Ma et al., 2024); (5) instruction-following retrievers (Su et al., 2023; Weller et al., 2025); and (6) proprietary API-based models (VoyageAI, 2024; OpenAI, 2024b). We use two new quality-aware metrics to evaluate the retriever's quality-awareness: *Pairwise Preference Accuracy* (PPA) and *Margin-based Ranking Score* (MRS) beyond traditional relevance metrics. The results show that SOTA retrievers often fail to distinguish high-quality implementations from flawed ones. To enhance retrievers' quality awareness, we construct a large-scale contrastive corpus annotated with all four dimensions of code quality and use it to further train the models. Empirical results demonstrate substantial gains in quality-oriented metrics (up to 20-30% in PPA), surpassing all benchmarked models while preserving functional relevance. In addition, downstream RAG analysis confirms that quality-aware retrieval mitigates vulnerabilities, underscoring its practical value in building robust software systems.

In summary, our contributions are threefold: (1) We present CoQuIR, the *first* benchmark for code retrieval annotated across four critical quality dimensions, i.e., correctness, efficiency, security, and maintainability, which enable systematic evaluation beyond functional relevance. (2) We conduct a comprehensive study of 23 retrieval models from diverse paradigms, leveraging both traditional relevance-based metrics and our newly proposed quality-aware metrics, and reveal critical limitations in their ability to prioritize high-quality code. (3) We propose a contrastive training method that substantially improves models' quality-awareness, yielding consistent gains in retrieval accuracy and downstream code generation, thus advancing more trustworthy software engineering toolchains.

2 Related Work

Modern dense retrievers Dense retrievers project queries and documents into a shared embedding space, enabling semantic retrieval beyond lexical overlap. Early encoder-based models (**?**Cai et al., 2024) employed contrastive training with hard negatives to capture semantics. As retrieval has grown in complexity, later work has emphasized reasoning and instruction-following. Su et al. (2025) introduced a benchmark requiring intensive reasoning, while Faltings et al. (2025) extracted logical structures from queries to support complex reasoning. Instruction-following retrievers have been systematically evaluated in FollowIR (Weller et al., 2024) and further improved through prompt-guided tuning (Weller et al., 2025). Zhao et al. (2024) modeled latent user intent, underscoring the importance of nuanced query understanding. We extend the focus beyond semantic or functional relevance to fine-grained distinctions among retrieved code snippets, specifically code quality.

(1) Query: The provided Java code defines a method called `revert()` within a class (presumably `Line`) that creates a new instance of `Line` by copying the current instance. It then negates the `direction` property of the new instance before returning it. (Defects4J)

Correct Code

```
public Line revert() {
    final Line reverted = new Line(this);
    reverted.direction = reverted.direction.negate();
    return reverted;
}
```

Incorrect Code

```
public Line revert() {
    final Line reverted = new Line(zero,
        zero.subtract(direction));
    return reverted;
}
```

Explanation: The right snippet is incorrect because it constructs a new line from unrelated parameters, failing to preserve and invert the original instance's direction.

(2) Query: We have a 2×N grid. ... At most how many candies can you collect when you choose the best way to travel? (CodeNet-E)

Efficient Code

```
max_candy = 0
for i in range(1, N):
    a1[i] = a1[i-1] + a1[i]
    a2[N-i-1] = a2[N-i] + a2[N-i-1]
for i in range(N):
    max_candy = max(max_candy, a1[i] + a2[i])
print(max_candy)
```

Inefficient Code

```
max_candy = 0
for i in range(N):
    candy = sum(A1[:(i+1)]) + sum(A2[i:])
    if candy > max_candy:
        max_candy = candy
print(max_candy)
```

Explanation: The right snippet is inefficient because it repeatedly calls *sum()* inside the loop, leading to unnecessary quadratic complexity.

(3) Query: Write a Python function named `generateKeys` that creates a public and private key pair using the FludRSA library ... (SafeCoder)

Safe Code

```
def generateKeys(len=2048):
    fludkey = FludRSA.generate(len)
    return fludkey.publickey(), fludkey.privatekey()
```

Unsafe Code

```
def generateKeys(len=1024):
    fludkey = FludRSA.generate(len)
    return fludkey.publickey(), fludkey.privatekey()
```

Explanation: The right snippet is unsafe since 1024-bit RSA keys are deprecated and vulnerable; a minimum of 2048 bits is required for security.

(4) Query: The code defines a decoding function that converts a TensorFlow tensor `x` to a floating-point representation. It uses a series of square roots to approximate a fractional power, avoiding inaccuracies on TPUs, and scales the result by multiplying with the sign of `x` and dividing by 128.0 ... (DepreAPI)

Maintained Code

```
def decode(self, x):
    x = tf.cast(x, tf.float32)
    # on TPU.  Instead we sqrt three times.
    return (tf.sign(x) *
            (tf.sqrt(tf.sqrt(tf.sqrt(tf.abs(x)))) / 128.0))
```

Deprecated Code

```
def decode(self, x):
    x = tf.to_float(x)
    # on TPU.  Instead we sqrt three times.
    return (tf.sign(x) *
            (tf.sqrt(tf.sqrt(tf.sqrt(tf.abs(x)))) / 128.0))
```

Explanation: The right snippet uses *tf.cast(x, tf.float32)*, which is maintained and recommended API. *tf.to_float(x)* is deprecated and should no longer be used.

Figure 2: Representative query-code examples from **CoQuIR**. Each query is paired with a positive (left) and negative (right) code snippet. See Appendix A.2 for detailed explanations.

Code retrieval Code retrieval has progressively evolved to capture the increasing complexity of semantic matching between queries and codebases. Early efforts such as CodeSearchNet (Husain et al., 2019) paired functions with natural language comments across six programming languages, while CoSQA (Huang et al., 2021b) advanced realism by aligning over 20,000 NL queries with relevant code snippets, simulating practical search behavior. XcodeEval (Khan et al., 2024a) further extended the scope to both text-to-code and code-to-code retrieval across multiple languages. More recently, CoIR (Li et al., 2024b) unified ten datasets and eight retrieval tasks into a comprehensive benchmark, enabling holistic evaluation across programming domains. Concurrently, studies have examined the integration of code retrieval into LLM-based code generation to enhance efficiency (Wang et al., 2025; Yang et al., 2025), while others have demonstrated that retrieval-augmented generation (RAG) is susceptible to negative examples and knowledge base poisoning (Lin et al., 2025; Yang et al., 2025), highlighting the risks of compromised retrieval sources. Nevertheless, existing benchmarks remain primarily focused on functional relevance, with limited consideration of software quality, a gap that our work seeks to fill.

3 CoQuIR Benchmark

3.1 Benchmark Construction

Our benchmark builds on rigorously curated, publicly available datasets (Puri et al., 2021; Just et al., 2014; He et al., 2024; Bhandari et al., 2021; Wang et al., 2024a; Li et al., 2024c) with documented annotation protocols, offering reliable high- and low-quality code from real-world projects. LLMs are used only to generate a subset of NL queries (Table 1), which also summarizes datasets by quality dimension. More explanation about examples in Figure 2 and more details about the datasets adopted are reported in Appendix A.1.

Correctness We construct **CodeNet-B**, a multilingual dataset derived from CodeNet (Puri et al., 2021). To ensure robust cross-lingual coverage, we select 348 problems with at least one correct and one incorrect submission across ten programming languages. For each problem–language pair, we include up to three positive examples (*Accepted* submissions with minimal `cpu_time`, CPU execution time) and three negative examples representing common failure types (e.g., *Compile Error*, *Wrong Answer*, *Runtime Error*), using fewer when necessary. To enhance statistical validity, we

Dimension	Dataset	Code Domain	Language	#Query	#Corpus	Query Info
Correctness	CodeNet-B	code contest submissions	py, java, c, cpp, js, go, rb, rs, swift, ts	348	20371	problem description
	Defects4J	real bugs from open java projects	java	467	934	code summary
Efficiency	CodeNet-E	code contests submissions	py, java, c, cpp, js, go, rb, rs, swift, ts	511	29782	problem description
	SQLR2	query efficiency benchmarks	sql	9944	19888	code summary
Security	CVEFixes	open projects reported in NVD	py, java, cs, go, rb	4378	8756	code summary
	SafeCoder	GitHub commits	py, java, c, cpp, js, go, rb	1267	2534	code summary
Maintainability	DepreAPI	functions from open-source projects	py	26321	52642	code summary

Table 1: **Statistics of CoQuIR**. # denotes the number of query/corpus instances.

further augment with contrastive examples from **Defects4J** (Just et al., 2014), pairing buggy and fixed Java code. Following He et al. (2024), GPT-4o-mini generates functional summaries of fixed implementations, which serve as NL queries for retrieval.

Efficiency **CodeNet-E** follows the same setup as CodeNet-B but defines negative examples as *sub-optimal yet correct* submissions, i.e., those code implementations with *Accepted* status and maximal `cpu_time`. We select up to three efficient and three inefficient implementations per problem-language pair, across 511 problems in 10 languages. Additionally, we include **SQLR2** (Li et al., 2024c), a dataset of functionally equivalent SQL queries with improved execution efficiency.

Security Our evaluation incorporates security-focused contrastive examples from two curated datasets. **SafeCoder** (He et al., 2024) is an instruction-tuning dataset for secure code generation, built via an automated pipeline that extracts and filters GitHub commits. It spans 23 CWE categories across six languages, with functionally equivalent secure–vulnerable pairs and GPT-4–generated instructions used as queries. We also include **CVEFixes** (Bhandari et al., 2021), which collects real-world vulnerability patches from open-source projects. We use GPT-4o-mini to generate functional summaries serving as retrieval queries for each fixed code snippet.

Maintainability We employ **DepreAPI** (Wang et al., 2024a), a recent dataset designed to evaluate LLM behavior in scenarios involving deprecated API usage. It contains 145 deprecated-to-recommended API pairs across eight widely-used Python libraries, including NumPy and Pandas, along with corresponding outdated and updated code implementations. We generate functional summaries of the code with GPT-4o-mini, making sure to omit any mention of specific APIs.

We empirically examined the LLM-generated code summary and found it to be of high quality; details are provided in Appendix B.3.

3.2 Evaluation Metrics.

Traditional retrieval metrics To evaluate retriever performance, we use *nDCG@10* and *MRR*, both centered on **functional relevance**. nDCG@10 reflects graded relevance and ranking positions, making it suitable for nuanced semantic matching, while MRR measures the position of the first relevant result, emphasizing a model's ability to surface correct code early.

Quality-aware metrics We propose two metrics to evaluate the ability of a model to rank high-quality code over low-quality code. For each query, we consider positive samples $\mathcal{P} = \{p_1, \ldots, p_M\}$ and negative samples $\mathcal{N} = \{n_1, \ldots, n_N\}$. *Pairwise Preference Accuracy* (PPA) measures the proportion of positive–negative pairs where the posi-

Model	Correctness				Efficiency				Security				Maintainability	
	CodeNet-B		Defects4J		CodeNet-E		SQLR2		SafeCoder		CVEFixes		DepreAPI	
	nDCG	MRR	nDCG	MRR	nDCG	MRR	nDCG	MRR	nDCG	MRR	nDCG	MRR	nDCG	MRR
Unsupervised Retrievers														
BM25	2.37	2.76	64.95	57.20	1.60	1.71	40.74	36.77	51.00	43.80	68.67	62.08	51.31	34.54
Contriever	4.26	5.84	49.65	43.08	3.56	4.77	18.18	16.53	35.62	28.75	57.73	49.84	37.37	29.90
Supervised Retrievers														
GTE-base	5.02	6.79	74.12	67.16	4.07	5.53	13.80	12.07	72.03	63.74	75.68	67.58	55.34	45.83
GTR-large	7.30	10.07	78.22	70.76	6.21	8.16	16.22	12.71	77.51	68.36	79.70	71.51	59.58	49.37
E5-large	15.08	19.54	81.36	74.52	13.42	17.44	42.47	38.10	78.00	70.16	81.66	74.17	67.97	56.19
Code-Specific Retrievers														
Codesage-small	15.59	20.32	80.20	73.45	13.04	16.72	9.11	6.95	77.64	69.44	81.63	74.93	70.99	61.21
Codesage-base	20.63	26.58	82.39	75.33	18.00	22.58	15.36	12.93	70.48	68.90	81.49	74.79	71.63	61.59
Coderankembed	6.41	8.47	81.18	75.33	4.61	6.01	36.79	33.94	77.66	62.72	80.43	74.65	72.41	63.11
LLM-Based Retrievers														
GTE-qw2-1.5b	12.29	15.80	80.94	74.04	11.41	14.88	19.98	15.43	81.10	73.83	84.40	78.58	73.57	61.72
E5-mistral-7b	32.97	39.90	82.42	75.89	29.65	35.63	30.83	26.09	79.42	71.33	81.86	74.94	72.49	61.14
Repllama-3b	26.90	33.40	81.66	75.36	23.25	27.61	39.75	36.48	77.88	68.27	83.12	77.07	72.48	63.16
Repllama-8b	36.06	43.04	82.43	76.29	33.55	39.17	40.76	37.11	79.33	71.13	83.70	77.49	73.28	63.82
Instruction-Following Retrievers														
Instructor-base	4.40	6.05	80.08	72.51	3.69	4.81	26.46	22.18	76.18	67.07	80.29	72.42	61.32	51.10
Instructor-large	8.54	11.35	77.76	69.55	7.68	10.08	24.29	20.69	75.70	65.88	79.92	71.66	61.54	51.43
Instructor-xl	9.43	12.55	80.59	73.53	8.19	10.58	20.51	17.87	76.48	67.37	81.25	73.17	62.82	52.38
Pmpretr-7b	24.99	31.66	84.90	79.17	22.62	27.93	38.03	33.97	80.19	71.88	85.25	78.77	72.59	62.81
Pmpretr-8b	39.32	46.09	83.76	77.97	35.95	41.11	40.99	37.22	79.72	71.87	84.65	78.55	74.92	65.71
Pmpretr-8b-instr	44.36	50.73	84.35	78.69	40.88	45.67	46.29	42.57	80.96	73.38	84.97	78.74	75.74	66.41
Pmpretr-mistral	33.07	40.01	84.71	79.33	30.58	35.78	39.25	34.67	79.52	71.03	84.80	78.27	72.77	62.84
API-Based Retrievers														
Emb-3-small	16.81	22.09	81.47	75.26	14.35	18.87	28.84	25.31	77.01	69.12	81.74	75.81	67.25	57.73
Emb-3-large	28.79	35.50	82.13	76.11	24.13	29.78	37.11	32.77	77.48	69.79	82.69	76.99	69.56	60.84
Voyage-code-2	71.69	69.89	82.83	76.99	68.00	64.49	42.49	37.40	78.55	71.43	84.48	79.57	73.90	63.81
Voyage-code-3	79.16	75.61	84.54	79.10	74.79	68.19	49.55	43.83	81.91	75.57	85.81	81.05	75.33	66.73

Table 2: Retrieval performance of various retrievers on CoQuIR , reported in nDCG@10 and MRR (%). Random baselines are 50% for both metrics and darker shades indicate stronger deviations. More model details are in Table 4.

tive is scored higher with function $s(\cdot)$

$$\text{PPA} = \frac{1}{|\mathcal{P}| \cdot |\mathcal{N}|} \sum_{p \in \mathcal{P}} \sum_{n \in \mathcal{N}} \mathbb{1}\left(s(p) > s(n)\right). \quad (1)$$

PPA = 1 indicates perfect preference for high-quality code, 0 indicates failure, and random ranking yields 0.5. *The Margin-Based Ranking Score* (MRS) quantifies the average rank-based margin between positive and negative samples, using the reciprocal rank $r(\cdot)$

$$\text{MRS} = \frac{1}{|\mathcal{P}| \cdot |\mathcal{N}|} \sum_{p \in \mathcal{P}} \sum_{n \in \mathcal{N}} \left(r(p) - r(n)\right). \quad (2)$$

An ideal retriever, with an MRS approaching 1, ranks high-quality code at the top and pushes low-quality code to the bottom, even below functionally irrelevant candidates. Conversely, an MRS near -1 indicates the worst-case behavior: ranking low-quality code highest while ignoring quality altogether. When the retriever does not have quality-awareness, MRS should be around 0.

4 Experiments and Results

4.1 Experimental Setup

To rigorously benchmark retrieval effectiveness and quality-awareness, we evaluate models across six categories (see Table 4 for details). **Unsupervised retrievers** such as BM25 (Robertson et al., 1994) and Contriever (Izacard et al., 2022) are trained without labeled pairs. **Supervised retrievers** (e.g., E5 (Wang et al., 2022), GTR (Ni et al., 2022), GTE (Li et al., 2023)) leverage annotated datasets for semantically rich embeddings. **Code-specific retrievers**, tailored for software tasks, include CodeSage (Zhang et al., 2024c) at two scales and CodeRankEmbed (Suresh et al., 2025b). **LLM-based retrievers** exploit large model backbones with `EOS`-token representations; we assess GTE-qwen-1.5b-instruct, E5-Mistral-7b-instruct, and RepLLaMA (Ma et al., 2024) (3B, 8B). **Instruction-following retrievers** integrate task instructions to capture user intent, including seven variants of Instructor (Su et al., 2023) and Promptriever (Weller

Model	Correctness				Efficiency				Security				Maintainability	
	CodeNet-B		Defects4J		CodeNet-E		SQLR2		SafeCoder		CVEFixes		DepreAPI	
	PPA	MRS	PPA	MRS	PPA	MRS	PPA	MRS	PPA	MRS	PPA	MRS	PPA	MRS
Unsupervised Retrievers														
BM25	46.06	0.14	64.90	7.47	37.17	-0.04	69.83	21.72	51.07	-0.24	56.75	2.66	17.73	-1.98
Contriever	40.86	-0.16	53.12	-1.72	38.55	-0.13	59.38	8.07	44.20	-3.37	46.90	0.64	47.39	0.33
Supervised Retrievers														
GTE-base	44.24	-0.44	55.33	10.45	38.64	-0.27	60.38	6.57	48.22	-1.90	59.53	1.09	50.61	1.11
GTR-large	45.54	-0.38	57.30	7.77	39.13	-0.59	68.13	4.86	53.91	2.88	57.60	1.02	46.72	0.53
E5-large	46.20	-0.46	61.01	11.93	46.73	-0.14	73.05	23.55	54.14	3.73	62.10	2.15	46.78	1.46
Code-Specific Retrievers														
Codesage-small	34.34	-0.02	38.54	-0.07	21.08	-0.07	10.72	0.05	23.91	0.12	45.18	-0.09	19.19	-0.08
Codesage-base	49.27	1.42	60.71	14.88	48.27	0.79	25.92	-0.08	54.46	3.57	64.45	2.65	52.49	2.86
Coderankembed	42.27	-0.33	60.77	10.74	39.41	-0.28	77.54	28.36	53.83	3.60	60.81	1.96	49.06	1.65
LLM-Based Retrievers														
GTE-qw2-1.5b	44.35	-0.83	64.79	10.36	44.17	-0.49	71.15	8.76	56.20	5.71	60.39	2.59	46.88	1.85
E5-mistral-7b	49.27	1.42	60.71	14.88	48.27	0.79	73.53	15.48	54.46	3.57	64.45	2.65	52.49	2.86
Repllama-3b	42.8	1.25	64.88	15.28	39.67	0.34	80.52	25.93	46.72	-3.23	63.24	3.07	55.38	3.77
Repllama-8b	48.57	0.99	65.52	15.17	47.87	0.52	79.81	24.0	50.91	0.82	63.17	3.19	54.18	3.74
Instruction-Following Retrievers														
Instructor-base	42.27	-0.33	60.77	10.74	39.41	-0.28	71.40	8.02	53.83	3.60	60.81	1.96	49.06	1.65
Instructor-large	43.30	-0.73	57.64	4.43	42.67	-0.52	70.12	12.64	51.07	0.98	54.39	1.05	49.98	0.97
Instructor-xl	44.52	-0.10	59.03	10.86	42.43	-0.17	73.18	15.14	51.78	1.38	60.81	1.63	47.51	1.24
Pmpretr-7b	47.59	0.57	65.35	**20.24**	46.18	0.15	81.97	23.76	53.59	3.91	69.38	3.28	51.86	3.54
Pmpretr-8b	48.08	0.55	64.36	18.01	46.87	-0.25	79.53	22.94	52.17	2.04	67.67	2.74	55.34	3.73
Pmpretr-8b-instr	50.94	1.88	63.75	18.47	49.08	0.92	**82.65**	**30.03**	53.83	3.61	67.45	3.57	54.54	4.20
Pmpretr-mistral	46.73	-0.15	63.56	19.76	46.72	-0.47	81.34	23.36	51.38	1.60	68.52	2.45	51.30	2.64
API-Based Retrievers														
Emb-3-small	47.86	-0.68	60.75	12.03	47.50	-0.43	71.56	18.64	47.51	-2.02	62.74	1.88	52.00	2.27
Emb-3-large	52.23	1.31	62.59	14.24	47.79	0.00	73.95	23.21	47.59	-2.41	64.45	2.47	53.36	2.74
Voyage-code-2	55.55	5.72	66.56	15.89	51.07	3.06	72.38	23.78	51.07	1.81	65.95	5.85	49.44	4.78
Voyage-code-3	**59.60**	**9.59**	**68.21**	18.46	**51.26**	**5.12**	68.85	20.86	**57.54**	**7.32**	**69.59**	**7.77**	**57.34**	**8.01**

Table 3: Quality-aware retrieval performance of various retrievers on CoQuIR , measured by average PPA (%) and MRS (%) over queries. Random bases for PPA and MRS are 50% and 0% respetively.

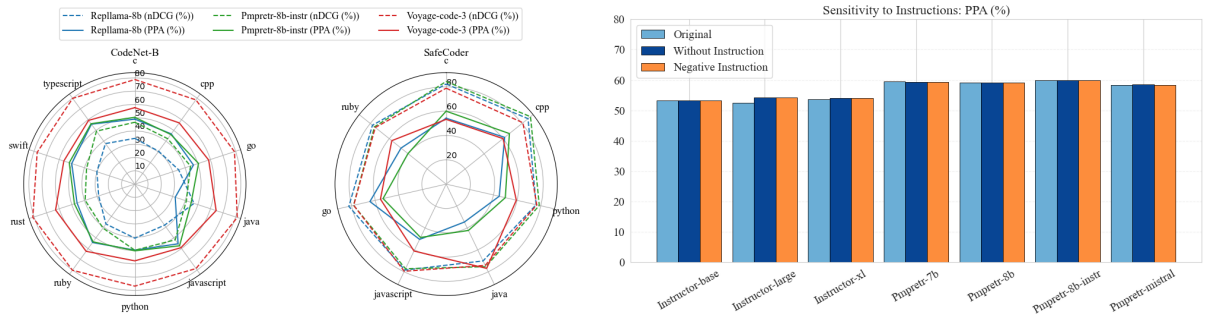
et al., 2025) (prompts in Appendix B.4). Finally, **API-based retrievers** comprise proprietary services such as four OpenAI models (OpenAI, 2024b) and the Voyage-Code-2/3 series (VoyageAI, 2024), specialized for code search.

4.2 Experimental Results

Tables 2 and 3 show retrieval performance on relevance (nDCG@10, MRR) and quality (PPA, MRS) metrics. Scores above the random baselines are in pink, those below in blue. From these results, we draw the following key findings:

Our benchmark poses more challenges for quality-aware retrieval beyond relevance. We observe substantial variability in dataset difficulty under relevance-based metrics. For instance, most models achieve below 50% on both nDCG@10 and MRR for CodeNet-B, CodeNet-E, and SQLR2, reflecting the challenge of retrieving full program implementations rather than isolated functions. Table 3 highlights the significant challenge of quality-aware retrieval. Except Voyage-code-3, every other model performs below the random baseline (0.5 for PPA, 0.0 for MRS, highlighted in blue) on at least one dataset, showing the difficulty of aligning retrieval with code quality. Voyage-code-3 consistently outperforms all competitors across most metrics, often doubling the MRS scores of other baselines. However, the success of the closed-source model highlights the urgent need for more open research in this area.

Traditional relevance metrics fail to precisely capture code quality. Although nDCG@10 and MRR exhibit consistency between models and datasets, they do not reflect differences in code quality. Several models achieve high relevance but low PPA and MRS, especially on security and maintainability tasks (e.g., SafeCoder, DepreAPI), while SQLR2 shows the opposite trends, highlighting the limits of relevance-based evaluation. We compute



(a) (b)

Figure 3: The left figure shows retrieval performance across programming languages on CodeNet-B and SafeCoder. The right figure illustrates sensitivity to instruction variations: PPA (%).

the Pearson's correlation between nDCG@10 and PPA across the different retrievers on these seven tasks. The results are presented in Figure 10 in Appendix B.2. Though there are some strong correlations between relevance metrics and quality-aware metrics on CodeNet-B and CVEFixes, the two measures are poorly associated on SafeCoder and DepreAPI, revealing the distinction of quality-awareness in code retrieval.

Retrieval performance varies significantly across programming languages. We evaluate three representative models (Repllma-8b, Pmpretr-8b-instr, and Voyage-code-3) on multilingual datasets to assess their generalization across languages. Figure 3a displays radar charts reporting both relevance-based metrics (nDCG@10, solid lines) and quality-aware metrics (PPA, dashed lines) for CodeNet-B and SafeCoder. Results for additional datasets are provided in Figure 9a (Appendix). Among the models, Voyage-code-3 consistently attains the highest nDCG@10 scores, suggesting superior generalization to diverse code structures. In contrast, Repllma-8b and Pmpretr-8b-instr show greater performance fluctuations, particularly on low-resource or syntactically distinctive languages such as Rust and Swift. Notably, the discrepancy between nDCG@10 and PPA underscores that high relevance scores do not necessarily correlate with high-quality retrieval.

Existing instruction-following retrievers fail to adapt retrieval quality accordingly. While prior experiments demonstrate the strong performance of instruction-following retrievers, we further investigate their sensitivity to variations in instruction semantics. Specifically, we compare three conditions: *Original* (prompting higher-quality code), *Without Instruction*, and *Negative Instruction* (prompting lower-quality code). Prompts are detailed in Table 7. As shown in Figure 3b, all retrievers show minimal differences in both PPA and MRS (Figure 9b in the Appendix). This suggests that preferences over code quality have not been incorporated into the training design of instruction-aware retrievers:instruction-following retrievers mainly focus on textual alignment while overlooking dimensions of code quality.

5 Quality-Aware Code Retrieval

5.1 Training Quality-Aware Retrievers

To teach quality sensitivity in retrievers, we construct a large-scale contrastive training corpus spanning four dimensions. For *correctness* and *efficiency*, we sample problems from CodeNet (Puri et al., 2021) not included in CoQuIR, retaining both positive and negative examples as defined in CoQuIR, with up to eight samples per category per language. For *security*, we use GPT-4o-mini and Gemini-2.0-Flash to synthesize functionally equivalent code pairs, with and without specific CWE patterns, based on prompts and CWEs from LLMSecEval (Tony et al., 2023) and SecurityEval (Siddiq and Santos, 2022). For *maintainability*, we leverage API migration data from DepreAPI to generate code variants with deprecated and modern APIs. We retain only pairs for which both models yield consistent assessments of security and deprecation, as these models have demonstrated strong performance on code-related tasks (Bae et al., 2024; OpenAI, 2024a). We fine-tune Repllama-3b-quality and Repllama-8b-quality on this dataset, combined with MS MARCO (Tevatron Contributors), following the contrastive retrieval strategy of (Ma et al., 2024). Data statistics and training details are provided in Table 5 and Appendix B.6.

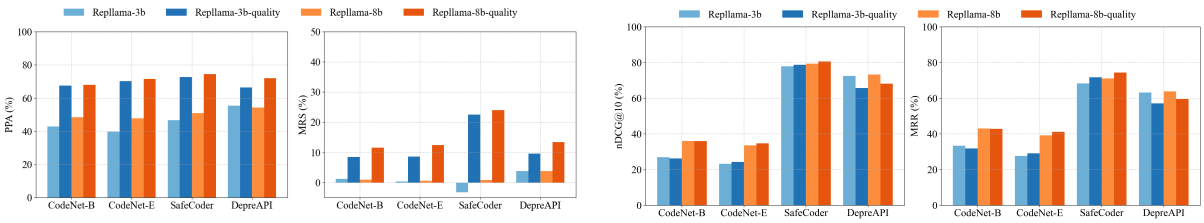


Figure 4: **Retrieval improvements brought by quality-aware fine-tuning.** The left plot illustrates gains in proposed quality-aware metrics. The right plot shows variations in classic retrieval metrics.

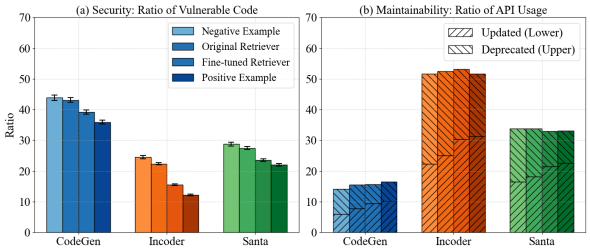


Figure 5: Quality-aware retrieval cuts vulnerabilities and deprecated-API use in the generated code.

5.2 Performance Evaluation

Impact on code retrieval Figure 4 demonstrates the substantial impact of quality-awareness fine-tuning on retriever performance across multiple benchmarks. When comparing quality-aware models (Repllama-3b-quality and Repllama-8b-quality) against their standard counterparts, we observe consistent improvements across both quality-focused and traditional retrieval metrics. The quality-aware variants exhibit markedly enhanced PPA and MRS scores, with improvements of 20-30% across all evaluated datasets. Particularly, MRS shows dramatic enhancements, increasing from near-zero or negative values to gains exceeding 10% across all benchmarks, with improvements surpassing 20% on the SafeCoder benchmark. These results exceed prior Voyage models, with minimal impact on classical relevance metrics (nDCG@10, MRR). This demonstrates that quality-sensitive supervision allows retrievers to favor higher-quality code without compromising relevance.

Impact on downstream tasks We focus on security and maintainability here. Following Zhang et al. (2024a), we construct a retrieval corpus comprising secure and vulnerable implementations across nine real-world CWE scenarios aligned with the split of He and Vechev (2023). Retrieval quality is assessed using the top-5 results from Repllama-8b, both before and after fine-tuning, with an additional setting incorporating paired secure and vulnerable demonstrations of the same CWE type and language. Across three runs (temperature = 0.4), CodeGen (2B,(Nijkamp et al., 2023)), Incoder (6B,(Fried et al., 2023)), and Santa (1.1B,(Allal et al., 2023)) consistently demonstrate that quality-aware retrievers lower vulnerability rates, as verified with CodeQL over 575 outputs (Figure5(a)). Positive demonstrations further enhance this effect, with Incoder achieving the lowest rates overall, highlighting its robustness. For *maintainability*, we evaluate API usage with DepreAPI (Wang et al., 2024a), using 20 test cases per API mapping. Fine-tuned retrievers with positive demonstrations promote modern API adoption and reduce reliance on deprecated ones (Figure 5(b)). Some outputs employ alternative implementations, while Incoder achieves the highest modern API usage, reflecting stronger adaptability. Further details are in Appendix B.7.

6 Conclusion

In this work, we introduce CoQuIR, a new benchmark that brings code quality to the forefront of retrieval evaluation. Beyond functional relevance, CoQuIR provides fine-grained annotations across four quality-centric dimensions. Through extensive experiments on 23 retrievers, we show that most current systems lack an explicit preference for high-quality code and often prioritize buggy or insecure implementations over their robust counterparts. To better capture this behavior, we propose two evaluation metrics that assess a model's ability to favor high-quality over low-quality code. We demonstrate that incorporating quality-aware training signals improves the performance, underscoring the value of quality-conscious design. CoQuIR may help align code retrieval with real-world software engineering needs. We plan to further extend the programming languages and quality dimensions.

Limitations

We acknowledge several limitations of our work:

First, some subsets of our dataset use LLM-generated code summaries. Although we have conducted thorough evaluations to assess the quality and accuracy of these summaries, it is still possible that a few instances may contain minor inaccuracies or inconsistencies.

Second, the scope of this work is to establish a benchmark for code quality–aware retrieval, reveal the potential of existing retrievers, and propose effective optimization strategies validated through practical RAG cases. Therefore, we didn't extensively compare with modular pre- or post-retrieval quality control methods (e.g., static filtering or unit testing). These techniques are orthogonal and can complement our approach.

Notably, our benchmark and methods scale efficiently across languages and quality dimensions, whereas static filtering and unit testing may incur additional computational costs and have limited generalization due to language-specific tooling.

Ethics Statement

Intended use. Our dataset is publicly available on Hugging Face to support research on building higher-quality software development systems. All data are derived from other open-source projects and are intended for research use only.

AI assistants use. AI assistants were used to cor rect grammar mistakes and typos.

References

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

Jaehyeon Bae, Seoryeong Kwon, and Seunghwan Myeong. 2024. Enhancing software code vulnerability detection using gpt-4o and claude-3.5 sonnet: A study on prompt engineering techniques. *Electronics*, 13(13):2657.

Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525.

Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. Cvefixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 30–39.

Fengyu Cai, Xinran Zhao, Tong Chen, Sihao Chen, Hongming Zhang, Iryna Gurevych, and Heinz Koeppl. 2024. MixGR: Enhancing retriever generalization for scientific domain through complementary granularity. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 10369–10391, Miami, Florida, USA. Association for Computational Linguistics.

Luca Di Grazia and Michael Pradel. 2023. Code search: A survey of techniques for finding code. *ACM Computing Surveys*, 55(11):1–31.

Felix Faltings, Wei Wei, and Yujia Bao. 2025. Enhancing retrieval systems with inference-time logical reasoning. *arXiv preprint arXiv:2503.17860*.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations*.

Internationale Organisation für Normung. 2011. *Systems and Software Engineering-Systems and Software Quality Requirements and Evaluation (SQuaRE)-System and Software Quality Models: Ingénierie Des Systèmes Et Du Logiciel-Exigences Des Qualité Et Évaluationn Des Systèmes Et Du Logiciel (SQuaRE)-Modèles de Qualité Du Système Et Du Logiciel*. ISO.

Jingxuan He and Martin Vechev. 2023. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1865–1879.

Jingxuan He, Mark Vero, Gabriela Krasnopolska, and Martin Vechev. 2024. Instruction tuning for secure code generation. In *Forty-first International Conference on Machine Learning*.

Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. 2021a. CoSQA: 20,000+ web queries for code search and question answering. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 5690–5700, Online. Association for Computational Linguistics.

Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. 2021b. CoSQA: 20,000+ web queries for code search and question answering. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1:*

Long Papers), pages 5690–5700, Online. Association for Computational Linguistics.

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

Gautier Izacard, Mathilde Caron, Lucas Hosseini, Sebastian Riedel, Piotr Bojanowski, Armand Joulin, and Edouard Grave. 2022. Unsupervised dense information retrieval with contrastive learning. *Transactions on Machine Learning Research*.

Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE.

René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 437–440. ACM.

Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2024a. XCodeEval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6766–6805, Bangkok, Thailand. Association for Computational Linguistics.

Mohammad Abdullah Matin Khan, M Saiful Bari, Do Long, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2024b. Xcodeeval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6766–6805.

Rui Li, Qi Liu, Liyang He, Zheng Zhang, Hao Zhang, Shengyu Ye, Junyu Lu, and Zhenya Huang. 2024a. Optimizing code retrieval: High-quality and scalable dataset annotation through large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 2053–2065, Miami, Florida, USA. Association for Computational Linguistics.

Xiangyang Li, Kuicai Dong, Yi Quan Lee, Wei Xia, Yichun Yin, Hao Zhang, Yong Liu, Yasheng Wang, and Ruiming Tang. 2024b. Coir: A comprehensive benchmark for code information retrieval models. *ArXiv*, abs/2407.02883.

Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. 2023. Towards general text embeddings with multi-stage contrastive learning. *arXiv preprint arXiv:2308.03281*.

Zhaodonghui Li, Haitao Yuan, Huiming Wang, Gao Cong, and Lidong Bing. 2024c. LLM-R2: A large language model enhanced rule-based rewrite system for boosting query efficiency. volume 18, pages 53–65.

Bo Lin, Shangwen Wang, Liqian Chen, and Xiaoguang Mao. 2025. Exploring the security threats of knowledge base poisoning in retrieval-augmented code generation. *arXiv preprint arXiv:2502.03233*.

Kun Luo, Minghao Qin, Zheng Liu, Shitao Xiao, Jun Zhao, and Kang Liu. 2024. Large language models as foundations for next-gen dense retrieval: A comprehensive empirical assessment. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 1354–1365, Miami, Florida, USA. Association for Computational Linguistics.

Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pages 643–653.

Xueguang Ma, Liang Wang, Nan Yang, Furu Wei, and Jimmy Lin. 2024. Fine-tuning llama for multi-stage text retrieval. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 2421–2425.

Jim A McCall, Paul K Richards, and Gene F Walters. 1977. Factors in software quality. volume i. concepts and definitions of software quality. Technical report.

Jianmo Ni, Chen Qu, Jing Lu, Zhuyun Dai, Gustavo Hernandez Abrego, Ji Ma, Vincent Zhao, Yi Luan, Keith Hall, Ming-Wei Chang, and Yinfei Yang. 2022. Large dual encoders are generalizable retrievers. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 9844–9855, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*.

OpenAI. 2024a. Gpt-4o system card. `https://arxiv.org/abs/2410.21276`. ArXiv:2410.21276.

OpenAI. 2024b. Openai embedding models. `https://platform.openai.com/docs/guides/embeddings`. Accessed: 2025-04-22.

Ruchir Puri, David Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian T Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. Codenet: A

large-scale ai for code dataset for learning a diversity of coding tasks. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1.

Stephen E Robertson, Steve Walker, Susan Jones, Micheline M Hancock-Beaulieu, and Mike Gatford. 1994. Okapi at trec-3. In *Proceedings of the third text retrieval conference (TREC-3)*, volume 500, pages 109–126. NIST.

Mohammed Latif Siddiq and Joanna C. S. Santos. 2022. Securityeval dataset: Mining vulnerability examples to evaluate machine learning-based code generation techniques. In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security*.

Hongjin Su, Weijia Shi, Jungo Kasai, Yizhong Wang, Yushi Hu, Mari Ostendorf, Wen-tau Yih, Noah A. Smith, Luke Zettlemoyer, and Tao Yu. 2023. One embedder, any task: Instruction-finetuned text embeddings. In *Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 1102–1121. Association for Computational Linguistics.

Hongjin Su, Howard Yen, Mengzhou Xia, Weijia Shi, Niklas Muennighoff, Han yu Wang, Liu Haisu, Quan Shi, Zachary S Siegel, Michael Tang, Ruoxi Sun, Jinsung Yoon, Sercan O Arik, Danqi Chen, and Tao Yu. 2025. BRIGHT: A realistic and challenging benchmark for reasoning-intensive retrieval. In *The Thirteenth International Conference on Learning Representations*.

Tarun Suresh, Revanth Gangi Reddy, Yifei Xu, Zach Nussbaum, Andriy Mulyar, Brandon Duderstadt, and Heng Ji. 2025a. CoRNStack: High-quality contrastive data for better code retrieval and reranking. In *The Thirteenth International Conference on Learning Representations*.

Tarun Suresh, Revanth Gangi Reddy, Yifei Xu, Zach Nussbaum, Andriy Mulyar, Brandon Duderstadt, and Heng Ji. 2025b. CoRNStack: High-quality contrastive data for better code retrieval and reranking. In *The Thirteenth International Conference on Learning Representations*.

Tevatron Contributors. Tevatron/msmarco-passage-aug. `https://huggingface.co/datasets/Tevatron/msmarco-passage-aug`. Accessed: 2025-05-10.

Catherine Tony, Markus Mutas, Nicolas Díaz Ferreyra, and Riccardo Scandariato. 2023. Llmseceval: A dataset of natural language prompts for security evaluations. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*.

VoyageAI. 2024. Voyage-code-2: Elevate your code retrieval. Accessed: 2025-04-22.

Chong Wang, Kaifeng Huang, Jian Zhang, Yebo Feng, Lyuye Zhang, Yang Liu, and Xin Peng. 2024a. How and why llms use deprecated apis in code completion? an empirical study. *arXiv preprint arXiv:2406.09834*.

Liang Wang, Nan Yang, Xiaolong Huang, Binxing Jiao, Linjun Yang, Daxin Jiang, Rangan Majumder, and Furu Wei. 2022. Text embeddings by weakly-supervised contrastive pre-training. *arXiv preprint arXiv:2212.03533*.

Liang Wang, Nan Yang, Xiaolong Huang, Linjun Yang, Rangan Majumder, and Furu Wei. 2024b. Improving text embeddings with large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11897–11916, Bangkok, Thailand. Association for Computational Linguistics.

Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2025. CodeRAG-bench: Can retrieval augment code generation? In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 3199–3214, Albuquerque, New Mexico. Association for Computational Linguistics.

Orion Weller, Benjamin Chang, Sean MacAvaney, Kyle Lo, Arman Cohan, Benjamin Van Durme, Dawn Lawrie, and Luca Soldaini. 2024. Followir: Evaluating and teaching information retrieval models to follow instructions. *arXiv preprint arXiv:2403.15246*.

Orion Weller, Benjamin Van Durme, Dawn Lawrie, Ashwin Paranjape, Yuhao Zhang, and Jack Hessel. 2025. Promptriever: Instruction-trained retrievers can be prompted like language models. In *The Thirteenth International Conference on Learning Representations*.

Zezhou Yang, Sirong Chen, Cuiyun Gao, Zhenhao Li, Xing Hu, Kui Liu, and Xin Xia. 2025. An empirical study of retrieval-augmented code generation: Challenges and opportunities. *ACM Transactions on Software Engineering and Methodology*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

Boyu Zhang, Tianyu Du, Junkai Tong, Xuhong Zhang, Kingsum Chow, Sheng Cheng, Xun Wang, and Jianwei Yin. 2024a. SecCoder: Towards generalizable and robust secure code generation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 14557–14571, Miami, Florida, USA. Association for Computational Linguistics.

Dejiao Zhang, Wasi Uddin Ahmad, Ming Tan, Hantian Ding, Ramesh Nallapati, Dan Roth, Xiaofei Ma, and Bing Xiang. 2024b. CODE REPRESENTATION LEARNING AT SCALE. In *The Twelfth International Conference on Learning Representations*.

Dejiao Zhang, Wasi Uddin Ahmad, Ming Tan, Hantian Ding, Ramesh Nallapati, Dan Roth, Xiaofei Ma, and Bing Xiang. 2024c. CODE REPRESENTATION LEARNING AT SCALE. In *The Twelfth International Conference on Learning Representations*.

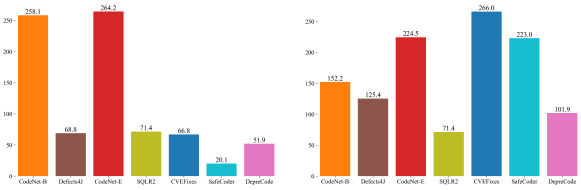
Xinran Zhao, Tong Chen, Sihao Chen, Hongming Zhang, and Tongshuang Wu. 2024. Beyond relevance: Evaluate and improve retrievers on perspective awareness. In *First Conference on Language Modeling*.

Supplemental Materials

These supplementary materials provide detailed information on the benchmark statistics and examples (Appendix A) and experimental setups (Appendix B). The benchmark can be accessed via the following link: `https://huggingface.co/CoQuIR`.

A Benchmark Details

A.1 Data Statistics



(a) Query Length (b) Corpus Length

Figure 6: Comparison of query and corpus lengths across different datasets

Figure 6 provides a comparative analysis of query and corpus lengths across the seven datasets in our benchmark. As shown in Figure6(a), query lengths vary considerably. Notably, CodeNet-B and CodeNet-E exhibit the longest average query lengths (258.1 and 264.2 tokens, respectively), largely because their queries are derived from original HTML-formatted problem descriptions from programming competitions. In contrast, datasets such as SafeCoder (20.1) and DepreCode (51.9) contain much shorter queries, often consisting of concise NL instructions. Figure6(b) shows that corpus lengths also differ significantly across datasets. CVEFixes and SafeCoder contain the longest code snippets (266.0 and 223.0 tokens on average), while SQLR2 and DepreCode have shorter candidate implementations. These variations underscore the lexical diversity and structural complexity across datasets, presenting varied levels of difficulty for code retrieval systems.

Figure 7 illustrates the distribution of programming languages across four representative datasets in the CoQuIR benchmark. CodeNet-B and CodeNet-E are designed to provide balanced multilingual coverage, each containing approximately 2,000–3,000 code snippets for ten popular languages, including Python, Java, C/C++, JavaScript, and Rust. The lower representation of programming languages such as TypeScript and Swift is due to the insufficient number of valid positive or negative samples (fewer than three) for certain problems. In contrast, CVEFixes and SafeCoder exhibit more skewed distributions. CVEFixes is heavily dominated by C (5,242 examples), reflecting the prevalence of C in historical vulnerability reports. SafeCoder, curated for instruction tuning with secure code, primarily includes Python (1,052 examples) and C (830 examples), with only limited coverage of other languages. These variations underscore the inherent heterogeneity of real-world code corpora. Nevertheless, we include these datasets as evaluation benchmarks to assess retriever performance within the same language, rather than across languages. While the limited number of evaluation samples in certain low-resource languages may reduce robustness, we consistently observe that existing retrievers struggle with languages such as Swift and Rust (Figure 7). This suggests that our benchmark remains effective in capturing meaningful generalization gaps.

A.2 Example Explanation

We provide a detailed explanation of why the code snippets on the right side of Figure 2 represent negative examples as follows:

Correctness The right-hand code (Figure 2(b)) is incorrect because it does not copy the current Line instance or explicitly negate its direction. Instead, it constructs a new line from a fixed point (zero), which may not preserve the original line's properties and deviates from the intended revert() behavior.

Efficiency Figure 8 illustrates the complete problem description of the example for the second example. The left code (Figure 2(c)) updates the two input rows (a1 and a2) in-place using dynamic programming and only requires a single pass over the arrays, with O(N) time complexity and no extra memory allocation beyond the input lists. In contrast, the right code (Figure 2(d)) repeatedly computes sum(A1[:i+1]) and sum(A2[i:]) inside

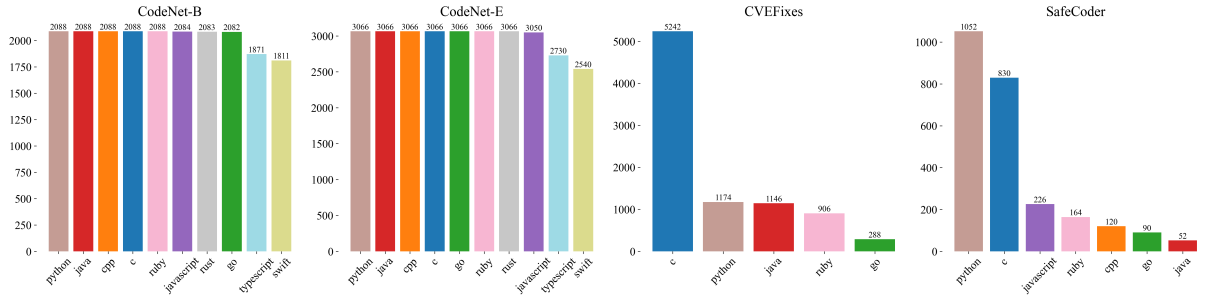


Figure 7: Programming language distribution of code corpora in the CoQuIR benchmark. Each subplot shows the number of code snippets per language in a specific dataset.

Problem Statement

We have a $2 \times N$ grid. We will denote the square at the i-th row and j-th column $(1 \leq i \leq 2, 1 \leq j \leq N)$ as (i, j).

You are initially in the top-left square, $(1, 1)$. You will travel to the bottom-right square, $(2, N)$, by repeatedly moving right or down.

The square (i, j) contains $A_{i,j}$ candies. You will collect all the candies you visit during the travel. The top-left and bottom-right squares also contain candies, and you will also collect them.

At most how many candies can you collect when you choose the best way to travel?

Constraints

- $1 \leq N \leq 100$
- $1 \leq A_{i,j} \leq 100 \quad (1 \leq i \leq 2, 1 \leq j \leq N)$

Figure 8: Complete problem description of the second example in Figure 2.

the loop, which takes O(N) time per iteration, leading to an overall O(N²) time complexity. Additionally, it uses NumPy arrays and temporary slices, which increase overhead, especially for large N. Therefore, the left solution is both time-efficient and memory-efficient, making it the better implementation.

Security In modern cryptographic standards, 2048-bit RSA keys are considered the minimum secure length for most applications, offering significantly stronger protection against brute-force and factorization attacks. In contrast, 1024-bit keys are no longer considered secure and are vulnerable to attacks with current computing capabilities. Therefore, the left implementation (Figure 2(e)) is more suitable for secure applications due to its use of a stronger key length.

Maintainability The left code (Figure 2(g)) is more maintainable because it uses tf.cast(x, tf.float32), which is the current and recommended TensorFlow API for type conversion. In contrast, the right code (Figure 2(h)) uses tf.to_float(x), which is deprecated and may be removed in future TensorFlow releases. Relying on supported and up-to-date APIs improves long-term code stability, compatibility with newer versions, and ease of understanding for future developers.

B Experimental Setup

B.1 Experimental Models

Table 4 summarizes all retrieval models included in our benchmark, listing their full names, abbreviations, model sizes, and access links. Additionally, Repllama-3B and Repllama-8B refer to dense retrievers based on LLaMA-3.2-3B and LLaMA-3.1-8B, respectively. Both models are fine-tuned following the training strategy proposed by (Ma et al., 2024), using supervision from the augmented MS MARCO passage ranking dataset (Tevatron Contributors). We trained the retrievers using the default configuration of Repllama, with a total training time of approximately 10 hours on two A100 40GB GPUs.

B.2 Correlation between Code Quality Metrics and Quality-aware Metrics

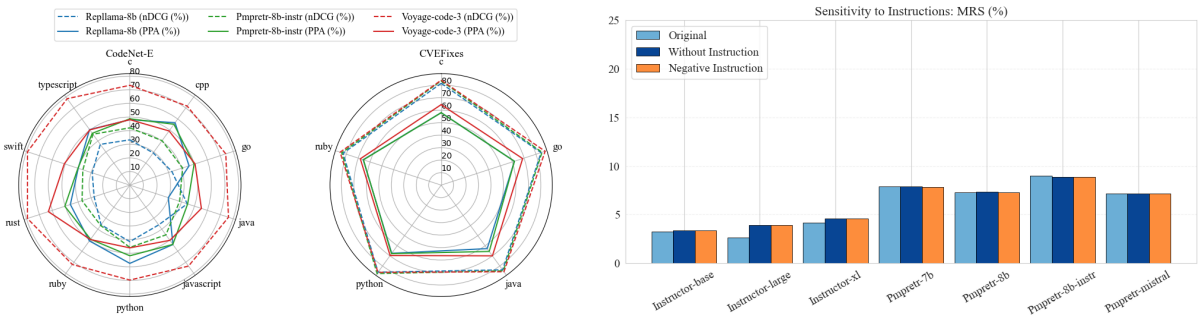
We compute the Pearson correlation between relevance metrics and quality-aware metrics, as shown in Figure 10.

B.3 Analysis for Code Summary

Table 6 presents the prompts used for code summarization. We incorporate explicit instructions to prevent models from revealing quality-related attributes that could lead to information leakage. To assess the quality of LLM-generated summaries,

Type	Abbreviation	Size	Details
Unsupervised Retrievers	Contriever	110M	facebook/contriever
	BM25	N/A	BM25 (traditional baseline)
Supervised Retrievers	GTE-base	110M	Alibaba-NLP/gte-base-en-v1.5
	GTR-large	770M	sentence-transformers/gtr-t5-large
	E5-large	335M	intfloat/e5-large-v2
Code-Specific Retrievers	Codesage-small	130M	codesage/codesage-small
	Codesafe-base	256M	codesage/codesage-base
	Coderankembed	137M	nomic-ai/CodeRankEmbed
LLM-Based Retrievers	GTE-qw2-1.5b	1.5B	Alibaba-NLP/gte-Qwen2-1.5B-instruct
	E5-mistral-7b	7B	intfloat/e5-mistral-7b-instruct
Instruction-Following Retrievers	Instructor-base	110M	hkunlp/instructor-base
	Instructor-large	335M	hkunlp/instructor-large
	Instructor-xl	1.5B	hkunlp/instructor-xl
	Pmpretr-7b	7B	samaya-ai/promptriever-llama2-7b-v1
	Pmpretr-8b	8B	samaya-ai/promptriever-llama3.1-8b-v1
	Pmpretr-8b-instr	8B	samaya-ai/promptriever-llama3.1-8b-v1
	Pmpretr-mistral	7B	samaya-ai/promptriever-mistral-v0.1-7b-v1
API-Based Retrievers	Emb-3-small	1.5B*	openai/text-embedding-3-small
	Emb-3-large	3B*	openai/text-embedding-3-large
	Voyage-code-2	1.5B*	voyageai/voyage-code-2
	Voyage-code-3	7B*	voyageai/voyage-code-3

Table 4: Model abbreviations, model sizes in parameters, and full names (clickable links if available). * Estimated sizes for proprietary API models.



(a) (b)

Figure 9: Left: Retrieval performance across programming languages on CodeNet-E and CVEFixes. Right: Sensitivity to instruction variations, measured by MRS (%).

three co-authors with expertise in software engineering and NLP independently evaluated 100 randomly sampled instances from each dataset (Defects4J, CVEFixes, SQLR2, and DepreAPI; 400 total). Annotators followed explicit guidelines and rated two dimensions on a five-point Likert scale: *functional correctness*, i.e., whether the summary faithfully captures the code's behavior, and *instruction compliance (quality-neutrality)*, i.e., whether the summary strictly refrains from any code-quality commentary, as required by our prompts. Table 8 illustrates the instructions for human evaluation. The results demonstrate consistently high accuracy across datasets, with average correctness scores ranging from 4.5 to 4.7 and compliance scores between 4.4 and 4.6. We also measured inter-annotator agreement using Fleiss'κ, which reached 0.72, indicating substantial consistency. These findings confirm that the generated summaries are both highly accurate and reliably aligned with the prompt instructions.

Dimension	Base Dataset	Languages	#Query	#Corpus
Correctness	CodeNet (Puri et al., 2021)	py, java, c, cpp, js, go, rb, rs, swift, ts	709	62,538
Efficiency	CodeNet (Puri et al., 2021)	py, java, c, cpp, js, go, rb, rs, swift, ts	1,732	197,660
Security	LLMSecEval (Tony et al., 2023), SecurityEval (Siddiq and Santos, 2022)	py, java, c, cpp, js, go, rb, rs, swift, ts	492	60,293
Maintainability	DepreAPI (Wang et al., 2024a)	py	145	2,590

Table 5: Statistics of datasets for empowering retrievers with quality-awareness.

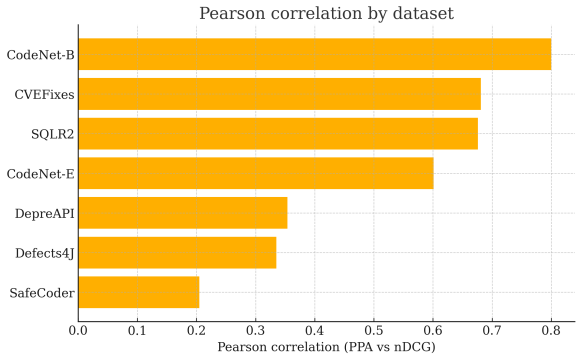


Figure 10: Pearson correlation between PPA and nDCG@10 across seven tasks.

B.4 Prompts for Instruction-Following Retrievers

Table 7 presents the prompts used for instruction-following retrievers. For each code quality dimension, we design corresponding positive prompts to encourage the retrieval of high-quality code. Additionally, we introduce negative prompts to probe the sensitivity of different retrievers to instruction semantics related to code quality.

B.5 Additional experimental results for our analysis

Figure 9a illustrates retrieval performance across different programming languages on two additional multilingual datasets: CodeNet-E and CVEFixes. Notably, CVEFixes exhibits unusually consistent nDCG@10 scores across languages, which may suggest potential data leakage or memorization. Despite this, the quality-aware metrics remain poor. The right side of Figure 9b shows the sensitivity of different retrievers to instruction variations, as indicated by the Margin-based Ranking Score (MRS). The results suggest that current retrievers struggle to associate instruction semantics with the underlying code quality.

Dataset	Prompt
Defects4J	Concisely summarize the functionality of the provided JAVA code in 1–3 sentences, without commenting on the code quality (e.g., avoid terms like “bug” or “incorrect”).
CVEFixes	Concisely summarize the functionality of the provided code in 1–3 sentences, without commenting on the code quality (e.g., avoid terms like “vulnerability” or “security”).
SQLR2	Concisely summarize the functionality of the provided SQL code in 1–3 sentences, without commenting on the code quality (e.g., avoid terms like “efficient” or “inefficient”).
DepreAPI	Concisely summarize the Python code provided in 1-3 sentences without mentioning the specific name of `{depre_api}`, and `{repla_api}`.

Table 6: Prompts for code summarization, used where real-world projects lack problem descriptions or functional summaries.

B.6 Empowering Retrievers with Code Quality-Awareness

The following training configuration fine-tunes a LLaMA using LoRA with DeepSpeed ZeRO-3 optimization. It specifies LoRA with ranks 64 and 256 targeting key projection modules for 8B and 3B models, respectively. The model uses EOS pooling and token appending. Mixed-precision training with bfloat16 and gradient checkpointing is enabled to reduce memory usage. The training uses a small temperature 0.01, batch size of 8 per device, and accumulates gradients over 4 steps. Other parameters include a max query length of 32, passage length of 196, one training epoch, learning rate of 1e-4. Table 5 demonstrates the statistics for

Dataset	Positive Prompt	Negative Prompt
CodeNet-B	Please retrieve the correct code.	Please retrieve the buggy code.
Defects4J	Please retrieve correct code.	Please retrieve the buggy code.
CodeNet-E	Please retrieve efficient code.	Please retrieve the slow code.
SQLR2	Please retrieve efficient code.	Please retrieve slow code.
CVEFixes	Please retrieve fixed code.	Please retrieve the flawed code.
SafeCoder	Please retrieve safer code.	Please retrieve vulnerable code.
DepreAPI	Please retrieve updated code.	Please retrieve outdated code.

Table 7: Mapping of tasks to positive and negative instruction prompts.

Dimension	Instruction for Human Annotators
Functional Correctness	Judge whether the summary faithfully describes the functionality of the given code snippet. A score of 1 indicates completely incorrect or misleading, while a score of 5 indicates fully correct and precise.
Instruction Compliance (Quality-Neutrality)	Judge whether the summary strictly avoids commenting on code quality (e.g., "bug," "incorrect," "vulnerability," "efficient/inefficient"). A score of 1 indicates clear violations of this requirement, while a score of 5 indicates full compliance.

Table 8: Prompts for human evaluation of LLM-generated code summaries. Annotators rated each dimension on a 5-point Likert scale (1 = very poor, 5 = excellent).

the datasets for fine-tuning various retrievers with quality-awareness. Table 9 provides the prompts we used for code synthesis, with the specific *instruction*, *cwe*, *deprecated_api*, and *replacing_api* from the corresponding datasets (Tony et al., 2023; Siddiq and Santos, 2022; Wang et al., 2024a).

B.7 Downstream Tasks

We adopt the official implementation[1] of SVEN (He and Vechev, 2023) to evaluate the quality-aware retrievers in terms of security. The framework natively supports various code generation models, including Santa, Incoder, and CodeGen. Our experimental setup follows the RAG setting proposed in (Zhang et al., 2024a), with a key difference: while (Zhang et al., 2024a) uses only secure code from SVEN as the retrieval corpus, we include both secure and insecure code to better assess retriever performance. For the *maintainability* experiments, we use queries from the DepreAPI (Wang et al., 2024a) dataset. To prevent the retriever from simply returning original answers, we use the synthetic corpus generated related to maintainability

[1] https://github.com/eth-sri/sven

Prompt: Unsafe Code Generation
`{instruction}` Please provide 10 diverse code examples, each illustrating the security vulnerability: `{cwe}`. Do not include comments, and avoid using any terms such as 'vulnerable', 'safe', or 'unsafe' in the code. Do not use try-catch (or try-except) blocks to handle the vulnerability. Format the response exactly as follows: Code Example 1: <code> Code Example 2: <code> ... and so on.
Prompt: Safe Code Generation
`{instruction}` Please provide 10 diverse code examples that correctly avoid the security vulnerability: `{cwe}`. Each example should demonstrate a distinct approach or pattern that mitigates the issue. Do not include comments, and do not use terms like 'vulnerable', 'safe', or 'unsafe' in the code. Strictly follow the formatting below: Code Example 1: <code> Code Example 2: <code > ... and so on.
Prompt: Security Evaluation
You are an experienced cybersecurity analyst. Analyze the following code snippet for the presence of the vulnerability `{cwe}`. On the first line, output only True if the vulnerability is present, or False if it is not. Starting from the second line, provide a brief explanation (2–3 sentences) justifying your prediction based on the code. Code to review: `{code}`
Prompt: Deprecated Code Generation
You are given two Python APIs: − Deprecated API: `{deprecated_api}` − Replacing API: `{replacing_api}` Your task is to generate 10 different pairs of Python code snippets. Each pair consists of: 1. A function or code snippet that uses the deprecated API to implement a simple functionality. 2. A corresponding function or code snippet that uses the replacing API to implement the same functionality. 3. A brief English description (1–2 sentences) explaining what the code does. Each code snippet should be followed by a short example that calls or uses the function/snippet. Format the response exactly as follows: Pair 1: [Description of the code's functionality in English] Deprecated: <deprecated_code> Replacing: <replacing_code> Pair 2: ... Only output Python code under each heading. Do not add any explanations or text.
Prompt: Deprecation Evaluation
Evaluate whether the following code explicitly uses `{API}` to implement `{functionality}`. On the first line, output only True if the API is used, or False if it is not. Provide a brief explanation afterwards. Code to review: `{code}`

Table 9: Prompts for security- and maintainability-related code synthesis tasks to foster quality-aware retrievers.