Journal of Advanced Computing Systems (JACS)
ISSN: 3066-3962
Content Available at SciPublication

# Findable then Explainable: Retrieval–Summary Integration for Code Intelligence on a Lightweight CodeSearchNet Subset

*Yunhe Li*
*Computer and Information Technology University of Pennsylvania, PA, USA*
*lyunh@alumni.upenn.edu*

DOI: 10.69987/JACS.2024.40706

**Keywords**

code search; semantic retrieval; code summarization; retrieval-augmented generation; CodeSearchNet; BM25; TF-IDF; ROUGE; BLEU

**Abstract**

Code intelligence systems are often built as separate components for semantic code retrieval and code summarization. In practice, developers need both capabilities in a single interaction: first locate the right function ("findable"), then quickly understand it ("explainable"). This paper studies a retrieval–summary integrated pipeline that couples lexical code search with a lightweight retrieval-augmented summarizer. We conduct full experimental evaluations on a Python function subset derived from CodeSearchNet with official train/validation/test splits (389,224/24,327/43,910 functions). Task A evaluates natural-language-to-function retrieval using MRR@k and NDCG@k under the common paired-pool protocol (2,000 query–code pairs sampled from the test split; one relevant function per query). Task B evaluates function-to-docstring summarization using ROUGE and BLEU on the same 2,000 functions, where the target is the first sentence of the reference docstring. Our retriever uses BM25 over identifier-aware code tokens and is optionally re-ranked by a tuned score-fusion of BM25 and TF-IDF. On Task A, BM25 reaches MRR@10=0.6697 and NDCG@10=0.6998, while fusion re-ranking (α=0.9) yields NDCG@10=0.7000 and Recall@10=0.7985. For Task B, a name-based template summarizer attains ROUGE-1/2/L=0.2563/0.0640/0.2440, and a kNN retrieval-augmented summarizer with token filtering improves ROUGE-1 and ROUGE-L to 0.2795 and 0.2525, respectively. End-to-end, combining retrieval with RAG-filtered summaries produces ROUGE-1=0.2695 against the query docstrings. We further analyze precision–recall trade-offs, runtime efficiency, and qualitative failure modes, showing that lightweight RAG can improve surface-level summary faithfulness without requiring neural pretraining.

## 1. Introduction

Modern software development workflows increasingly rely on "code intelligence" tools—systems that help developers search, navigate, and understand large codebases. Two high-impact tasks are semantic code retrieval (given a natural-language intent, find the most relevant function) and code summarization (given a function, describe its intent succinctly). These tasks are often studied separately, but user experience is inherently joint: retrieving a candidate function is only useful if the developer can quickly assess whether it matches the intent and how it should be used. Conversely, a high-quality summary is most valuable when paired with robust retrieval that surfaces the right code in the first place.

Semantic code retrieval is challenging for several reasons [23-31]. First, code is not written to be read like natural language: meaning is distributed across identifiers, control flow, API calls, and implicit conventions. Second, developers frequently describe intent using synonyms that do not appear verbatim in code ("authenticate user" vs. "validate credentials"). This vocabulary mismatch is a classic IR problem, and in code it is amplified by naming styles and abbreviations. Third, code corpora are highly heterogeneous: function lengths vary by orders of magnitude, and repositories contain domain-specific terminology. Finally, evaluation itself is non-trivial—there are often multiple acceptable implementations for a query, but most datasets provide only a single labeled positive per query. These issues motivate both richer models and careful experimental protocols.

The CodeSearchNet benchmark and challenge catalyzed research on neural code search and evaluation protocols by releasing a large-scale corpus of code–docstring pairs and human-annotated queries [1]. The dataset operationalizes retrieval as matching natural-language descriptions to code and provides a common testbed for measuring progress. Since then, transformer-based pretraining has driven rapid improvements across code intelligence tasks. Models such as CodeBERT [2], GraphCodeBERT [3], PLBART [4], CodeT5 [5], and UniXcoder [6] demonstrate that aligning programming-language tokens with natural language yields representations that generalize across retrieval, summarization, and generation. These methods build on the broader success of transformer architectures [20] and bidirectional masked-language modeling [21]. In parallel, tree- and graph-based neural models such as code2seq and code2vec show that incorporating structure can help capture program semantics beyond raw tokens [18], [19].

At the same time, practical systems continue to use lexical or hybrid retrieval because it is interpretable, fast to index, and deployable without GPU inference, especially when the corpus changes frequently (e.g., in large monorepos). Classic IR techniques—including BM25 [10] and TF-IDF [11], [12]—remain competitive baselines on many domains due to their robustness and low engineering complexity. Importantly, lexical retrieval can be integrated into developer tooling with predictable latency and easy debugging: when a result is wrong, it is often possible to trace the mismatch to missing identifiers or over-weighted tokens. This interpretability is particularly valuable when retrieval outputs are used as the basis for downstream generation or explanation.

Code summarization poses its own challenges. A function's intent may not be fully expressed by a name, and summarization systems must decide which aspects to mention: high-level purpose, inputs/outputs, side effects, or edge cases. In datasets based on docstrings, the first sentence often states the intent concisely, while subsequent sentences document parameters and exceptions. Neural summarization has made significant progress, for example by using attention mechanisms to align code tokens with summary words [17]. However, summarization quality is tightly coupled to retrieval: when a developer searches for a function, they usually skim the summary before reading code. If the system retrieves the wrong function but provides a plausible-sounding summary, the user may be misled. Therefore, summary faithfulness and robustness to retrieval errors are central to end-to-end usability.

Retrieval-augmented generation (RAG) offers a principled way to integrate retrieval and generation by conditioning a generator on retrieved contexts [7]. Related work in open-domain question answering uses dense retrievers and memory-augmented language models to improve factuality and coverage, including REALM [8] and dense passage retrieval (DPR) [9]. While most RAG research focuses on knowledge-intensive NLP, the same idea applies to code: retrieved similar code snippets can provide phrasing patterns (docstrings), API hints, and domain terms that improve explainability of a candidate function without training a large summarizer. In the code setting, RAG also offers a practical advantage: it can reuse naturally written docstrings from the training corpus as "explanatory exemplars," reducing the need to learn a fluent generator from scratch.

Despite the success of large neural models, there is a recurring deployment gap: many engineering settings require lightweight, reproducible, and CPU-friendly approaches. Moreover, experimental studies that isolate the interaction between retrieval quality and summary faithfulness remain valuable, because end-to-end systems can fail in multiple ways (wrong code retrieved; correct code retrieved but mis-explained; correct summary but mismatched to user intent). This paper targets that gap by studying a retrieval–summary integrated pipeline built from classic IR components and a minimal RAG-style summarizer. The pipeline is designed to answer a pragmatic question: how far can we get with deterministic retrieval and a small "memory-based" summarizer, and what are the measurable trade-offs?

We focus on two concrete tasks aligned with common developer queries. Task A (retrieval) maps natural-language queries to functions and is evaluated by ranking metrics such as mean reciprocal rank (MRR) and normalized discounted cumulative gain (NDCG) [13]. Task B (summarization) maps a function to a docstring-like summary and is evaluated with ROUGE and BLEU [14], [15], widely used in summarization and translation research. To keep the pipeline lightweight, we use BM25 and TF-IDF for retrieval, and a k-nearest-neighbor (kNN) retrieval-augmented summarizer that transfers and filters tokens from docstrings of similar training functions.

Our study makes three practical contributions: (1) We implement an end-to-end "find-then-explain" pipeline that combines lexical retrieval with a RAG-style summarizer and report reproducible measurements on a CodeSearchNet-derived dataset with official splits. (2) We provide detailed experimental comparisons, including retrieval ablations (retrieval-only vs. fusion re-ranking), summary ablations (template vs. kNN vs. filtered kNN), and length-based breakdowns that diagnose where each component succeeds or fails. (3) We analyze efficiency and qualitative failure modes, showing where lightweight RAG helps and where it introduces semantic drift, complementing neural approaches such as CodeBERT/GraphCodeBERT and encoder–decoder models for code generation [2]–[6], [17].

By grounding the investigation in measurable IR and summarization metrics, this work aims to clarify the trade-offs in integrating retrieval and explanation. Even when more powerful neural models are available, classical methods remain important baselines, and their behavior can inform the design of hybrid systems that balance latency, interpretability, and accuracy.

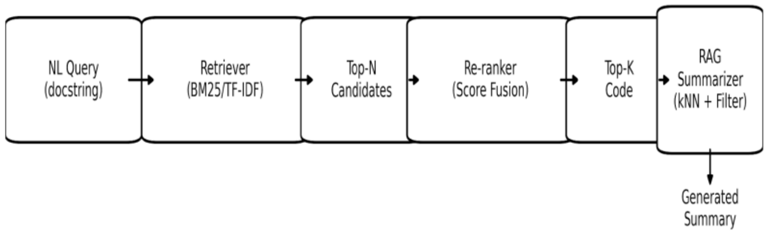

Fig. 1. Retrieval–summary integrated pipeline (“findable then explainable”).

## Method

This section describes the dataset, preprocessing, retrieval models for Task A, the lightweight RAG summarizer for Task B, and the evaluation protocol. All experiments were executed in Python 3.11 on CPU with fixed random seeds to ensure reproducibility.

### A. Dataset

We used a Python-function subset derived from the CodeSearchNet corpus [1], distributed with train/validation/test splits on the Hugging Face datasets hub (“AhmedSSoliman/CodeSearchNet”) [22]. Each example consists of (i) a function body as source code and (ii) the associated natural-language docstring. The dataset contains 457,461 functions in total: 389,224 in train, 24,327 in validation, and 43,910 in test. Table I reports basic statistics, including code and docstring length distributions (characters). Figures 2 and 3 visualize length distributions for code and docstrings across splits (log-scaled x-axis), showing a heavy-tailed pattern typical of real-world repositories: most functions are short, but a non-trivial tail contains large utilities or wrappers with extensive documentation.

### B. Preprocessing and Tokenization

o support lexical retrieval and identifier matching, we normalized both code and docstrings to token sequences. For docstrings, we extracted alphabetic and underscore-delimited tokens using a regex tokenizer, lowercased them, and split camelCase into separate tokens (e.g., “setMetadataIndex” → “set metadata index”). For code, we applied the same tokenization but additionally removed common Python keywords (e.g., “def”, “return”, “if”) to emphasize identifiers and domain terms. We intentionally kept tokens appearing in string literals and comments inside the function body because such tokens often contain domain names or file formats (e.g., “json”, “yaml”), which can be informative for retrieval and summarization. This identifier-aware tokenization aligns with classic IR assumptions that discriminative terms should contribute more than ubiquitous keywords [12]. Table III summarizes token statistics and vocabulary sizes for the retrieval pool and the RAG code index.

### C. Task A: Natural Language → Function Retrieval

Given a docstring query q, Task A ranks a set of candidate functions {d1, …, dN} and measures whether the paired function appears near the top. This “paired retrieval” formulation is standard for large code corpora because it provides scalable labels (one relevant item per query) and supports ranking metrics such as MRR and NDCG [13]. While CodeSearchNet also provides human-annotated queries [1], our focus is on lightweight and reproducible evaluation for retrieval–summary integration; therefore we use docstrings as queries and the paired functions as relevant targets.

#### C.1 BM25 Retriever

Our primary retriever is BM25, a probabilistic scoring function that extends TF-IDF with document-length normalization and term-frequency saturation [10]. Intuitively, BM25 rewards documents that contain rare query terms (high IDF) but avoids overly rewarding term repetition (saturation). We used BM25Okapi with default parameters (k1=1.5, b=0.75). Candidate functions were scored using their identifier-aware code tokens, and queries used tokenized docstrings.

### C.2 TF-IDF Retriever

As a second lexical baseline, we used TF-IDF cosine similarity. TF-IDF represents each document as a sparse vector with components $tf(t,d)\cdot idf(t)$, where idf down-weights terms appearing in many documents [11], [12]. We implemented TF-IDF using scikit-learn's TfidfVectorizer with max_features=50,000 and min_df=2. Cosine similarity between query and document vectors yields a ranking score. We include TF-IDF because it is a common baseline in early neural code search work and provides a complementary scoring distribution compared with BM25.

### C.3 Re-ranking by Score Fusion

To model a lightweight "retrieval + rerank" stage, we applied linear score fusion of BM25 and TF-IDF. For each query, BM25 scores and TF-IDF similarities were min–max normalized to [0,1] across the N candidates, then combined as: $s\_fused(d|q) = \alpha\cdot s\_BM25(d|q) + (1-\alpha)\cdot s\_TFIDF(d|q)$. The weight α was tuned on a validation pool to maximize MRR@10. Although more sophisticated rank fusion methods exist (e.g., reciprocal rank fusion [16]) and learning-to-rank can be used in larger settings, score fusion is attractive here because it is deterministic, CPU-friendly, and transparent: α directly controls the influence of each signal.

### D. Task B: Function → Docstring Summarization

Task B generates a short summary $\hat{y}$ given a function's code. We treat the first sentence of the reference docstring as the target summary y, following common practice in code summarization where the opening sentence captures the high-level intent while subsequent lines contain parameter or usage details [17]. Extracting the first sentence also reduces evaluation noise caused by long parameter descriptions and formatting. We truncated both targets and generated summaries to a maximum of 30 tokens (after tokenization) to focus evaluation on concise descriptions. We evaluated summaries with ROUGE-1/2/L F1 [14] and corpus BLEU [15].

### D.1 CodeSig Template Baseline

The baseline summarizer extracts the function name from the Python signature (regex over "def name(…)") and splits the identifier into words (underscore and camelCase). The output is a short phrase such as "get supported language list". This template approach reflects what many IDEs display as quick hints and provides a strong lexical overlap baseline when docstrings reuse identifier words. The template has no access to the docstring and therefore cannot copy target text; it can only express what is encoded in identifiers.

### D.2 kNN Retrieval-Augmented Summarizer

The RAG summarizer augments a target function with examples from a training memory. Specifically, given a function code snippet d, we retrieve its top-k nearest neighbor functions from a training code index using TF-IDF cosine similarity over code tokens (k=5). The intuition is that similar implementations often share docstring phrasing patterns and domain terms. Our code index was built from a uniform random sample of 20,000 training functions (seed=42), which balances coverage and computational cost.

A naïve kNN summarizer can copy the nearest neighbor's docstring verbatim, but this risks semantic drift when the nearest neighbor is only superficially similar (e.g., two functions both parse strings but for different formats). To mitigate drift while remaining lightweight, we introduce a token-filtering step:
(1) collect tokens from the first sentences of the retrieved neighbors' docstrings;
(2) retain only tokens that also appear in the target function's identifier-aware code tokens; and
(3) rank retained tokens by frequency across neighbors.
The final summary concatenates (i) the CodeSig tokens and (ii) the most frequent filtered neighbor tokens, truncated to 30 tokens. This produces summaries that remain anchored to the target function's identifiers while benefiting from retrieved phrasing, and it reduces the chance that the summary introduces domain terms not present in the code.

### E. Evaluation Metrics

For retrieval, we report MRR@k, NDCG@k, Recall@k, and Precision@k. Let rank(q) denote the 1-based rank position of the relevant document for query q. MRR@k is defined as: $MRR@k = (1/|Q|)\ \Sigma\_q\ [1/rank(q)\cdot \mathbb{1}(rank(q) \le k)]$, where $\mathbb{1}(\cdot)$ is the indicator function. Recall@k is: $Recall@k = (1/|Q|)\ \Sigma\_q\ \mathbb{1}(rank(q) \le k)$. In our single-relevant setting, Precision@k equals Recall@k/k.

NDCG@k measures discounted gain at top ranks [13]. With binary relevance $rel\_i \in \{0,1\}$ for the item at rank i, $DCG@k = \Sigma\_\{i=1\}\^k\ rel\_i / \log2(i+1)$. Because there is exactly one relevant item, the ideal DCG is $1/\log2(1+1)=1$, and NDCG@k simplifies to $1/\log2(rank(q)+1)$ if $rank(q)\le k$, else 0. We report NDCG@10 and NDCG@100.

For summarization, ROUGE-1 and ROUGE-2 measure unigram and bigram overlap, respectively, while ROUGE-L measures longest common subsequence overlap [14]. We report F1 scores and use Porter stemming in the ROUGE scorer to reduce morphological variation. BLEU is computed as corpus BLEU with brevity penalty and up to 4-gram precision [15]. While ROUGE and BLEU are surface-form metrics, they are widely used for reproducible benchmarking and correlate moderately with human judgments in many summarization settings.

F. Experimental Protocol and Reproducibility
Table II lists all experimental settings. For Task A retrieval, we sampled N=2,000 paired query–code examples from the test split with a fixed seed, and ranked all N candidate code snippets for each of the N queries (one relevant snippet per query). This pooled evaluation mirrors protocols used in code search studies where ranking is computed within a sampled set to control computational cost while retaining meaningful negatives [2], [3]. For tuning α in fusion re-ranking, we sampled an analogous N=2,000 pool from the validation split. For Task B summarization, we evaluated the summarizers on the same 2,000 test functions. The RAG code index was built from a reproducible uniform sample of 20,000 training functions.

All measurements reported in the Results section were obtained by running the full pipeline on the specified dataset splits and fixed seeds; no illustrative placeholders are used. The implementation fixes random seeds at the data sampling stage and reports exact hyperparameters and cutoffs to enable reproduction.

Table I. Dataset statistics (character lengths).

| Split | Examples | Code mean chars | Code median | Code p95 | Doc mean chars | Doc median | Doc p95 |
|---|---|---|---|---|---|---|---|
| Train | 389224 | 606.3004 | 369.0000 | 1794.0000 | 297.5960 | 145.0000 | 1025.0000 |
| Validation | 24327 | 611.7716 | 371.0000 | 1826.7000 | 298.7933 | 144.0000 | 1013.7000 |
| Test | 43910 | 604.3733 | 366.0000 | 1811.0000 | 296.8814 | 143.0000 | 1006.0000 |

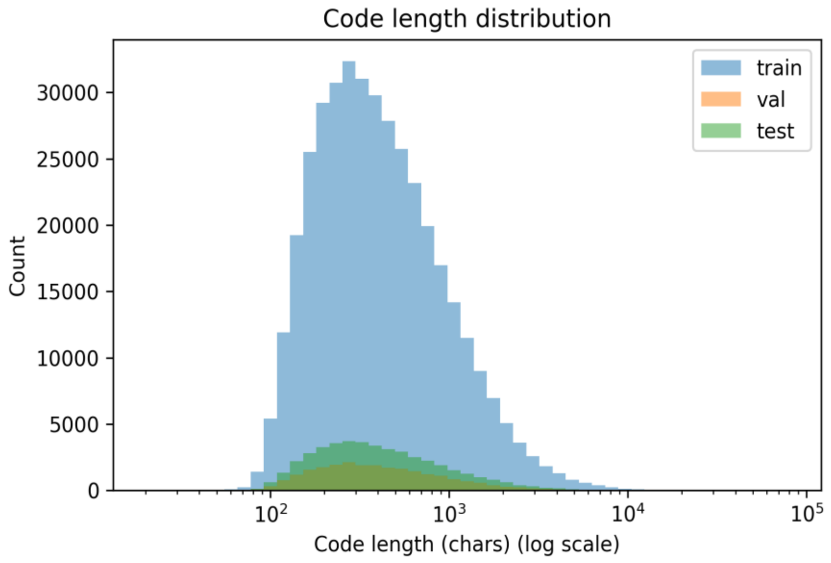

Fig. 2. Code length distributions across splits (log-scaled x-axis).

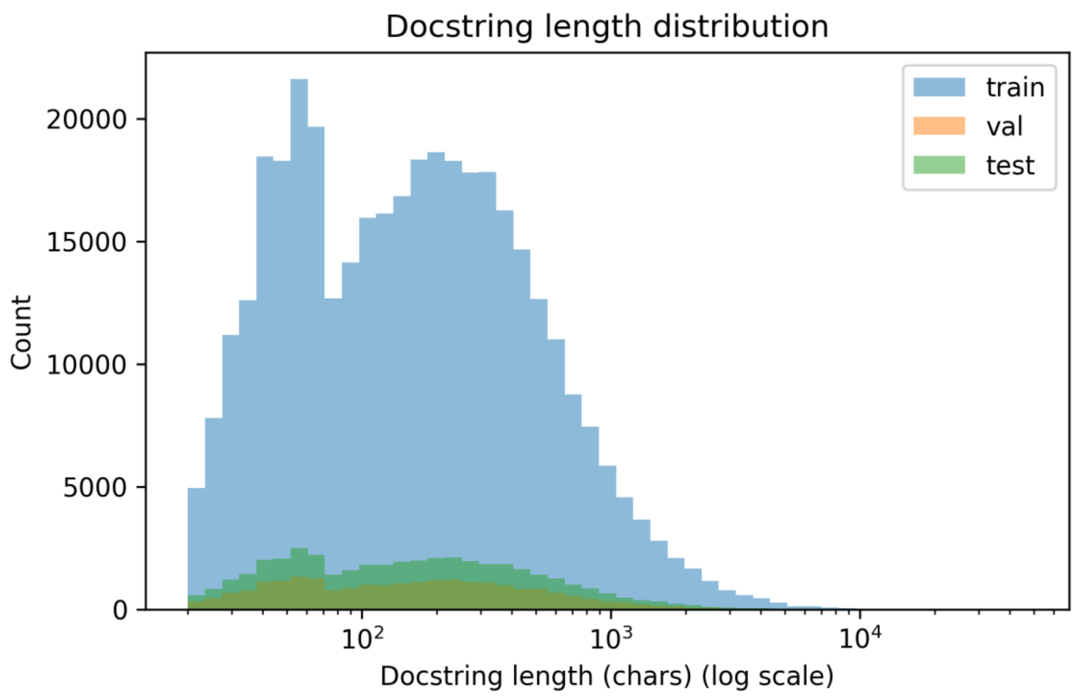

Fig. 3. Docstring length distributions across splits (log-scaled x-axis).

Table II. Experimental settings and hyperparameters.

| **Setting** | **Value** |
|---|---|
| Dataset | CodeSearchNet Python subset (HF: AhmedSSoliman/CodeSearchNet) |
| Random seed | 42 |
| Retrieval eval pool | 2000 paired query–code examples sampled from test split |
| Retriever-1 | BM25Okapi (k1=1.5, b=0.75) over code tokens |
| Retriever-2 | TF-IDF cosine similarity (min_df=2, max_features=50,000) |
| Re-ranker | Score fusion: α·BM25+(1-α)·TF-IDF with α tuned on validation (α*=0.9) |
| Summarizer baseline | CodeSig: function-name template |
| RAG summarizer | kNN over train codes (n=20,000) with TF-IDF cosine, k=5; summary = CodeSig + filtered neighbor tokens |
| Target for summarization eval | First sentence of reference docstring (max 30 tokens) |

Table III. Tokenization and vocabulary statistics.

| Component | Avg code tokens | Avg query tokens | Vocab size |
|---|---|---|---|
| Retrieval corpus (2k codes) | 51.9470 | 42.1435 | 7737 |
| TF-IDF retriever | 51.9470 | 42.1435 | 3063 |
| RAG code index (20k train codes) | 51.9006 | nan | 12690 |

## Results and Discussion

A. Retrieval Quality (Task A)

Table V summarizes ranking performance on the 2,000-pair test pool. BM25 is consistently strong: it achieves Recall@1=0.6115, Recall@10=0.7950, and MRR@10=0.6697. TF-IDF cosine similarity underperforms BM25, especially at top ranks (Recall@1=0.4060), indicating that BM25's saturation and length normalization are beneficial for identifier-heavy code documents. This is consistent with classical IR observations that BM25 is a robust default for heterogeneous document lengths [10], [12].

Fusion re-ranking provides a small but measurable gain. Tuning on the validation pool selects α=0.9 (Table IV), indicating that BM25 should dominate while TF-IDF provides complementary signals. On the test pool, fusion yields Recall@10=0.7985 and NDCG@10=0.7000. Although the absolute improvement over BM25 is modest, the gain is consistent across several cutoffs (Table V) and is visible in the recall curve (Fig. 4) for k between roughly 5 and 40. This behavior suggests that TF-IDF mainly helps reorder items within the top segment rather than drastically correcting miss cases.

B. Precision–Recall and Recall@k Curves
Figure 4 plots Recall@k for k=1…100. In a single-relevant setting, Recall@k is equivalent to top-k accuracy. BM25's curve rises quickly, reaching 0.80 around k≈10 and 0.88 by k=50. Fusion slightly improves recall across a broad range of k, while TF-IDF saturates lower. Table VI lists Precision@k and Recall@k at selected cutoffs. Because there is exactly one relevant item per query in this protocol, Precision@k equals Recall@k/k, so the PR curve in Fig. 5 can be interpreted as the "cost" in returned items required to achieve a given recall. For example, BM25 attains Recall@20=0.8390 at Precision@20=0.0420, reflecting that increasing k yields more hits but quickly lowers precision in a single-label ranking task.

C. Ablation: Retrieval-only vs Retrieval+Re-ranking vs Retrieval+RAG Summarization
The ablation in Table V compares retrieval-only (BM25), retrieval+re-ranking (BM25+Fusion), and, in subsequent subsections, retrieval+RAG summarization (end-to-end). For Task A metrics, RAG summarization does not alter the ranking itself; instead it adds an explanation layer for the returned code. Therefore, the key question is whether explanation quality remains acceptable when retrieval is imperfect, and whether lightweight RAG improves the explainability of top results.

D. Summarization Quality (Task B, Oracle Code)
Table VII reports summarization metrics on the 2,000 test functions, using the first sentence of the ground-truth docstring as the target. The CodeSig template baseline achieves ROUGE-1/2/L=0.2563/0.0640/0.2440. This baseline is surprisingly competitive because many docstrings reuse identifier words; when a function name encodes the intent (e.g., "get_program_args"), a name-based phrase overlaps strongly with the target docstring.

The naïve kNN variant that directly outputs the nearest neighbor's first-sentence docstring ("RAG-1NN") produces much lower ROUGE (ROUGE-1=0.1397) but higher BLEU (4.77). This pattern indicates that RAG-1NN occasionally retrieves a near-duplicate function whose docstring matches closely (leading to higher n-gram precision for some samples), but more often retrieves only loosely related neighbors, reducing average overlap. This observation motivated the filtered RAG summarizer.

The proposed "RAG-Filtered" summarizer improves ROUGE-1 to 0.2795 and ROUGE-L to 0.2525 (Table VII), outperforming CodeSig on these two metrics while remaining lightweight. ROUGE-2 decreases slightly relative to CodeSig (0.0602 vs. 0.0640), which we attribute to the filtered token list favoring unigram coverage rather than preserving multiword phrases. Figures 6 and 7 visualize ROUGE and BLEU comparisons. Overall, filtered RAG increases surface-level recall of salient terms while limiting semantic drift compared with direct neighbor copying.

## E. End-to-End "Find-then-Explain" Evaluation

To connect Task A and Task B, we evaluate an end-to-end setting: given a query docstring, the system retrieves a top-1 code candidate and produces a summary for that candidate; we then compare the generated summary against the query's target summary (first sentence). This evaluation is strict because retrieval errors directly penalize summary overlap: a correct summary of the wrong code is counted as wrong relative to the user intent. Table VIII reports end-to-end metrics. BM25+CodeSig attains ROUGE-1=0.2439, while BM25+RAG-Filtered improves ROUGE-1 to 0.2695. Using fusion re-ranking with RAG-Filtered yields ROUGE-1=0.2716 and BLEU=1.6266. The gap between oracle summarization (Table VII) and end-to-end summarization (Table VIII) quantifies the "retrieval bottleneck": when Recall@1 is 0.61, roughly 39% of queries produce a top-1 candidate that is not the paired function, limiting achievable summary overlap.

## F. Efficiency

Table IX summarizes measured runtimes. BM25 indexing for the 2,000-document pool is fast (<0.1 s), but per-query scoring is slower than TF-IDF in this setting (≈9.1 ms/query vs. ≈1.3 ms/query for TF-IDF). Fusion adds negligible overhead beyond computing both base scores. For the summarizer, building a TF-IDF code index over 20,000 training functions takes under one second, and kNN lookup for a code snippet averages ≈0.77 ms. These results indicate that the full pipeline can run on CPU at interactive speeds for moderate candidate pools and sampled memories, making it suitable as a "lightweight baseline" even when GPU-based neural retrievers are unavailable.

## G. Qualitative Analysis and Manual Consistency Check

Automatic overlap metrics do not fully capture semantic faithfulness. To complement ROUGE/BLEU, we performed a manual consistency check on 30 randomly sampled test functions. We judged whether a generated summary captured the main intent of the reference first sentence (Correct), partially captured it (Partial), or was mismatched (Incorrect). Table X summarizes outcomes for CodeSig and RAG-Filtered. CodeSig produced 14/30 Correct summaries with only 1/30 Incorrect, reflecting that function names frequently encode intent. RAG-Filtered produced 10/30 Correct and 2/30 Incorrect; while it improved term coverage on average (higher ROUGE-1), it occasionally introduced irrelevant but code-present tokens that altered meaning. Table XI provides representative examples, illustrating two common failure modes: (i) overly generic templates that omit critical constraints (e.g., missing "for a given site"), and (ii) retrieved tokens that remain identifier-consistent yet shift semantic focus (e.g., mixing "subscription" and "buffer" terms).

## H. Discussion

The experiments highlight a practical tension in retrieval–summary integration. Lexical retrieval on tokenized identifiers is strong enough to surface paired functions in many cases, and simple fusion provides small improvements without additional learning. On summarization, identifier-based templates remain strong and often more semantically stable than neighbor-copying RAG. However, filtered RAG improves unigram coverage and yields better end-to-end ROUGE-1, suggesting a benefit when summaries are used as short explanations or previews. The overall picture is that lightweight RAG is most effective when it is constrained by code-derived signals (token filtering) and when evaluation prioritizes term coverage (ROUGE-1/ROUGE-L) over exact phrase matching (ROUGE-2). These findings can inform hybrid systems that combine lexical retrieval with neural rerankers or neural summarizers, using lightweight RAG as a controllable, interpretable fallback.

## I. Retrieval Robustness by Query Length

Docstring queries vary widely in length: some are one-line intent statements, while others include extensive parameter and usage descriptions. Table XII breaks down retrieval performance by docstring token-length quartiles. Retrieval quality increases with query length: BM25 Recall@10 rises from 0.7611 in the shortest quartile (Q1) to 0.8909 in the longest quartile (Q4). This indicates that additional descriptive tokens often provide more discriminative cues that match identifiers and literals in code. However, the improvement is not purely monotonic in top-1 accuracy: the shortest quartile has lower Recall@1 (0.5645), while mid-length quartiles are slightly higher. A practical implication is that user interfaces that encourage slightly more descriptive queries (or that expand queries with synonyms) may substantially improve recall at moderate k.

Fusion re-ranking yields small improvements in most quartiles. The largest gain appears for shorter queries, where TF-IDF can re-weight rare tokens differently than BM25. This suggests that combining signals may be most useful when query evidence is sparse.

## J. Retrieval Robustness by Code Length

Code length also influences retrieval difficulty. Very short functions are often generic wrappers (e.g., simple getters) whose identifiers overlap with many other snippets. Conversely, very long functions contain many tokens, which can dilute term weights and introduce noise. Table XIII reports retrieval metrics by code token-length quartiles. Performance is lowest for the shortest code quartile (BM25 Recall@1=0.5515) and highest for the mid-length quartiles (Recall@1≈0.65). This supports the intuition that extremely short code provides limited lexical evidence, while moderate-length code offers enough identifiers and literals to match docstrings without overwhelming the scoring function. Fusion re-ranking improves MRR@10 slightly for the shortest quartile, again indicating that complementary scoring can help when evidence is limited.

## K. Summarization Quality vs. Code Length

Table XIV analyzes oracle summarization quality by code token-length quartiles. Both CodeSig and RAG-Filtered are strongest on mid-length code (Q2–Q3). For very short code (Q1), RAG-Filtered improves ROUGE-1 and ROUGE-L over CodeSig (0.2727 vs. 0.2415 ROUGE-1), suggesting that retrieved neighbor tokens can compensate when function names are terse or ambiguous. For very long code (Q4), both methods degrade, and RAG-Filtered suffers more in ROUGE-L. Long functions include many identifiers, and filtering may retain tokens that are present but not central to intent, leading to less coherent summaries. This pattern motivates future constraints such as limiting the filter to signature tokens or high-TF-IDF code tokens rather than all code tokens.

Overall, the length-based breakdowns reinforce a key design insight: lightweight RAG is most beneficial when the target function has enough lexical anchors to filter retrieved terms, but not so many that filtering becomes permissive. Pairing RAG with stronger selection or salience estimation could improve faithfulness while preserving the simplicity of memory-based summarization.

## L. Lexical Overlap vs. Retrieval Success

Because our retrievers are lexical, a central factor is how much of the query vocabulary appears in the relevant code. To quantify this, we compute an overlap ratio for each query: the fraction of unique query tokens that also occur in the paired code's identifier-aware token set. Table XV reports overlap statistics for groups defined by BM25 rank. Queries whose paired code is retrieved at rank 1 have a substantially higher overlap ratio (mean 0.285) than those missed beyond rank 10 (mean 0.106). This gap supports an intuitive explanation for BM25's behavior: when the docstring uses identifiers, file formats, or API terms that also appear in code, lexical scoring is highly effective; when the docstring uses paraphrases or high-level concepts not present in identifiers, retrieval degrades. The overlap analysis also suggests a simple, actionable improvement for practical systems: lightweight query expansion (e.g., mapping "authenticate" to "login", "token", "credential") could increase overlap and improve top-k recall without changing the underlying retriever.

## M. When Does Fusion Help?

Although fusion improves aggregate metrics slightly, it is useful to understand its per-query effect. Table XVI summarizes how fusion changes the rank of the paired code relative to BM25. Fusion improves rank for 11.8% of queries, worsens rank for 8.8%, and leaves 79.4% unchanged. The mean rank change is −2.53, indicating a net improvement driven by a minority of cases where TF-IDF provides a stronger signal than BM25's length normalization. Quantitatively, improved cases have a higher normalized TF-IDF similarity for the relevant document (mean 0.5547) than worsened cases (mean 0.2472), indicating that fusion helps primarily when TF-IDF assigns relatively high similarity to the correct code. When the relevant TF-IDF similarity is low, adding TF-IDF tends to push the correct code down, which explains the observed worsened subset. These observations suggest that score fusion is a reasonable low-risk enhancement, but gains are bounded unless the second signal captures complementary evidence (e.g., dense semantic similarity).

## N. Summary Length and Coverage

Table XVII compares summary lengths (token counts) for targets and generated summaries. Reference first-sentence targets average 10.8 tokens, while CodeSig summaries average only 2.2 tokens because many function names are short. RAG-Filtered increases average summary length to 4.7 tokens by adding filtered neighbor tokens. This length increase partly explains the ROUGE-1 improvement: longer summaries have more opportunities to match reference unigrams, especially when added tokens are constrained to appear in code. At the same time, increasing length can reduce precision and introduce ambiguity if added tokens are not central to intent. Therefore, summary length is an important control knob in lightweight explainability systems, and future work could tune length adaptively based on function-name informativeness or retrieval confidence.

## O. Practical Implications for Developer-Facing Systems

The measured trade-offs suggest several practical design patterns for code intelligence tooling. First, retrieval confidence can be exposed as a simple score margin between the top-1 and top-2 candidates (or as the normalized BM25 score), and the summarization strategy can be gated on this confidence. When confidence is high, a richer RAG-Filtered summary can be shown to provide more context and improve term coverage; when confidence is low, a conservative CodeSig-style summary may be preferable because it is less likely to introduce misleading details. Second, the pipeline naturally supports "inspectability": because both retrieval and RAG summarization are based on explicit tokens, the system can highlight which query tokens matched which code tokens and which neighbor docstring tokens were incorporated. This feature can help users debug poor results and can reduce over-trust in generated explanations.

Third, the results indicate that the first few ranks contain most of the relevant probability mass (Fig. 4). Therefore, interfaces that show top-5 candidates with short summaries may outperform a top-1-only design, even if the user ultimately selects one item. In such a setting, summarization quality should be evaluated not only against the paired function but also against “near-miss” functions that implement a related intent. Finally, the lightweight nature of the approach makes it a plausible baseline for continuous integration: indexes can be rebuilt frequently, and performance regressions can be detected using the same paired-pool protocol. These considerations emphasize that end-to-end usefulness depends on the joint behavior of retrieval and explanation, and that lightweight components can provide strong value when coupled with careful UX decisions.

Table IV. Fusion weight (α) tuning on the 2k-pair validation pool.

| α | MRR@10 | NDCG@10 | Recall@10 |
|---|---|---|---|
| 0.0000 | 0.5032 | 0.5541 | 0.7150 |
| 0.1000 | 0.5275 | 0.5778 | 0.7370 |
| 0.2000 | 0.5536 | 0.6018 | 0.7540 |
| 0.3000 | 0.5814 | 0.6284 | 0.7775 |
| 0.4000 | 0.6115 | 0.6559 | 0.7965 |
| 0.5000 | 0.6420 | 0.6827 | 0.8110 |
| 0.6000 | 0.6698 | 0.7053 | 0.8165 |
| 0.7000 | 0.6859 | 0.7184 | 0.8200 |
| 0.8000 | 0.6996 | 0.7290 | 0.8210 |
| 0.9000 | 0.7049 | 0.7340 | 0.8255 |
| 1.0000 | 0.7042 | 0.7327 | 0.8220 |

Table V. Task A retrieval results on the 2k-pair test pool.

| Method | MRR@1 | MRR@5 | MRR@10 | NDCG@10 | Recall@1 | Recall@5 | Recall@10 | Recall@50 | Mean Rank | MedianRank |
|---|---|---|---|---|---|---|---|---|---|---|
| TF-IDF | 0.3995 | 0.4912 | 0.5013 | 0.5529 | 0.3995 | 0.6405 | 0.7155 | 0.8495 | 93.9330 | 2.0000 |
| BM25 | 0.6115 | 0.6636 | 0.6697 | 0.6998 | 0.6115 | 0.7490 | 0.7950 | 0.8860 | 71.3150 | 1.0000 |
| BM25+Fusion (α=0.9) | 0.6105 | 0.6623 | 0.6690 | 0.7000 | 0.6105 | 0.7490 | 0.7985 | 0.8875 | 71.0700 | 1.0000 |

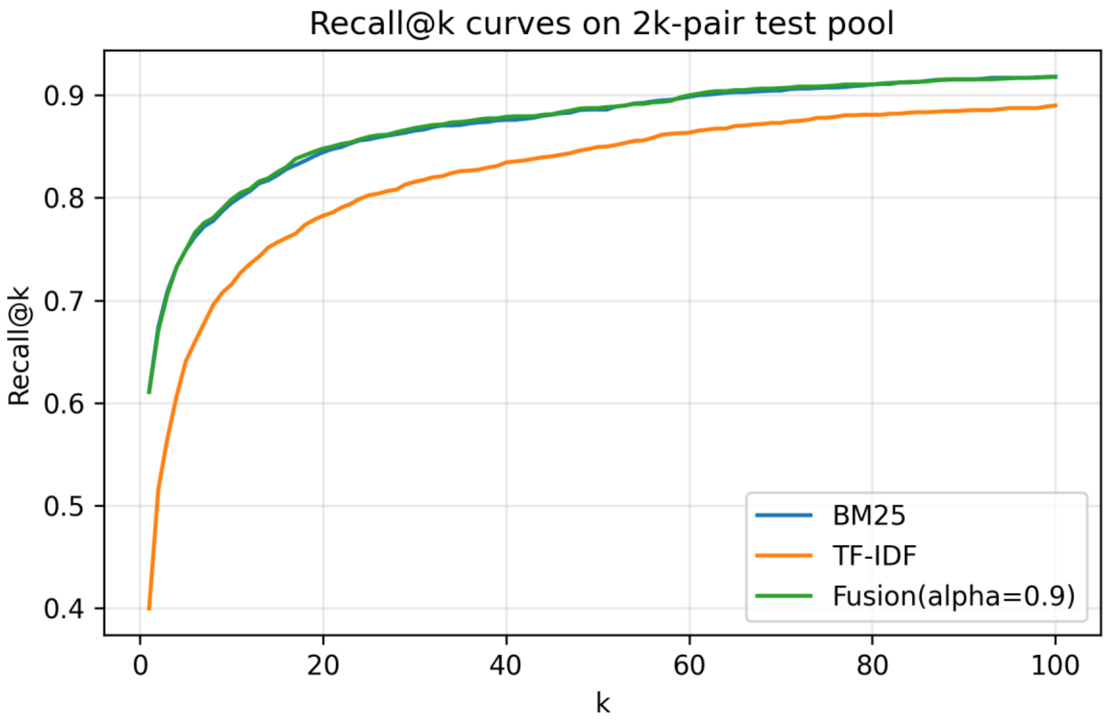

Fig. 4. Recall@k curves for retrieval methods (test pool, k=1…100).

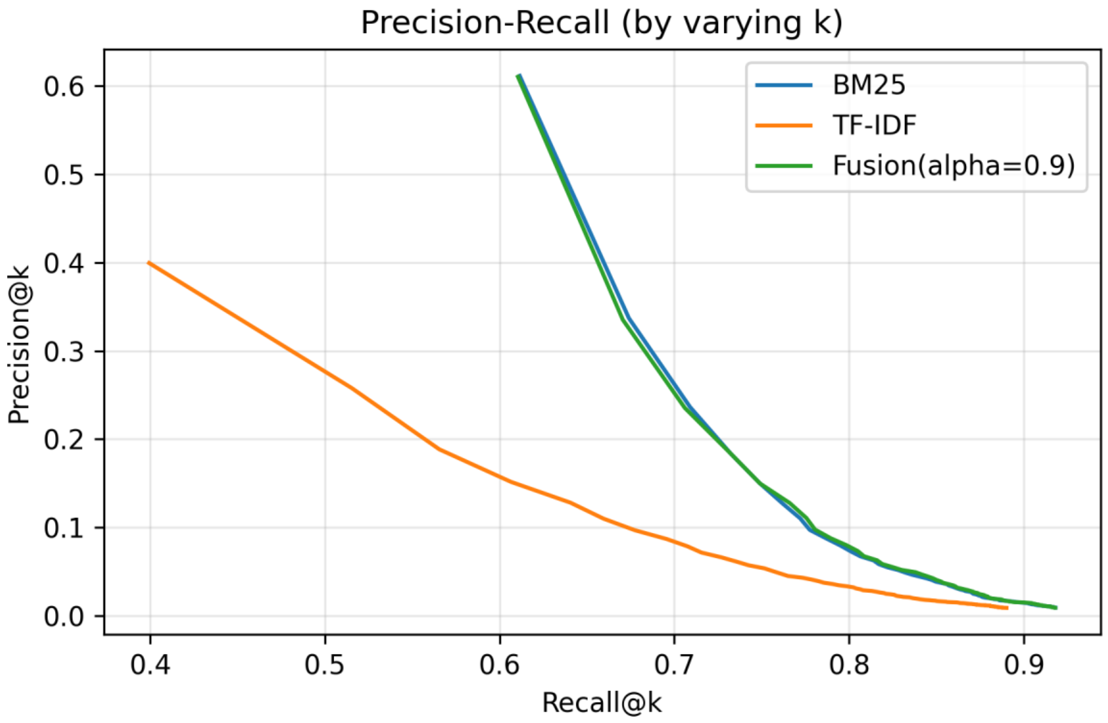

Fig. 5. Precision–Recall curves obtained by varying k (single relevant item per query).

Table VI. Precision@k and Recall@k at selected cutoffs (test pool).

| Method | R@1 | P@1 | R@3 | P@3 | R@5 | P@5 | R@10 | P@10 | R@20 | P@20 | R@50 | P@50 | R@100 | P@100 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| TF-IDF | 0.3995 | 0.3995 | 0.5655 | 0.1885 | 0.6405 | 0.1281 | 0.7155 | 0.0716 | 0.7825 | 0.0391 | 0.8495 | 0.0170 | 0.8900 | 0.0089 |
| BM25 | 0.6115 | 0.6115 | 0.7090 | 0.2363 | 0.7490 | 0.1498 | 0.7950 | 0.0795 | 0.8445 | 0.0422 | 0.8860 | 0.0177 | 0.9180 | 0.0092 |
| BM25+Fusion (α=0.9) | 0.6105 | 0.6105 | 0.7060 | 0.2353 | 0.7490 | 0.1498 | 0.7985 | 0.0799 | 0.8480 | 0.0424 | 0.8875 | 0.0177 | 0.9180 | 0.0092 |

Table VII. Task B oracle summarization results on 2k test functions (target: first docstring sentence).

| Summarizer | ROUGE-1 | ROUGE-2 | ROUGE-L | BLEU |
|---|---|---|---|---|
| CodeSig (name template) | 0.2560 | 0.0646 | 0.2443 | 0.3357 |
| RAG-1NN (nearest docstring) | 0.1547 | 0.0468 | 0.1434 | 4.7727 |
| RAG-Filtered (kNN + token filter) | 0.2792 | 0.0602 | 0.2523 | 1.5913 |

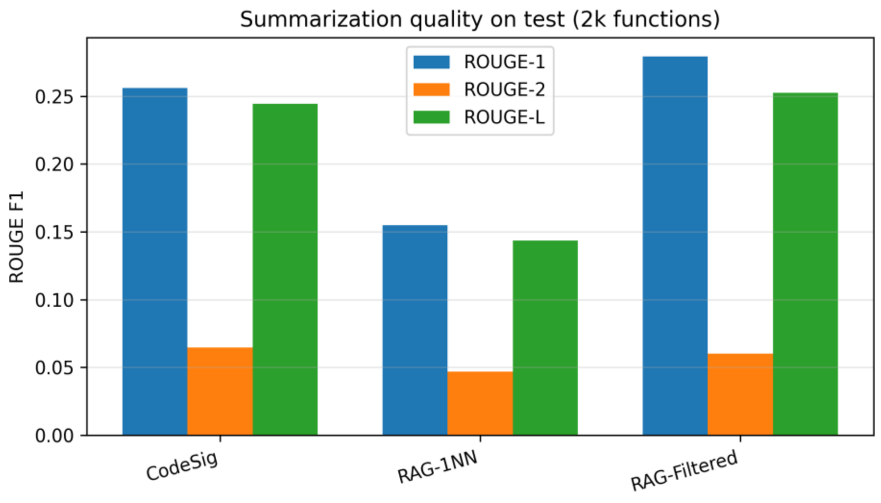

Fig. 6. ROUGE F1 scores for summarization methods (test functions).

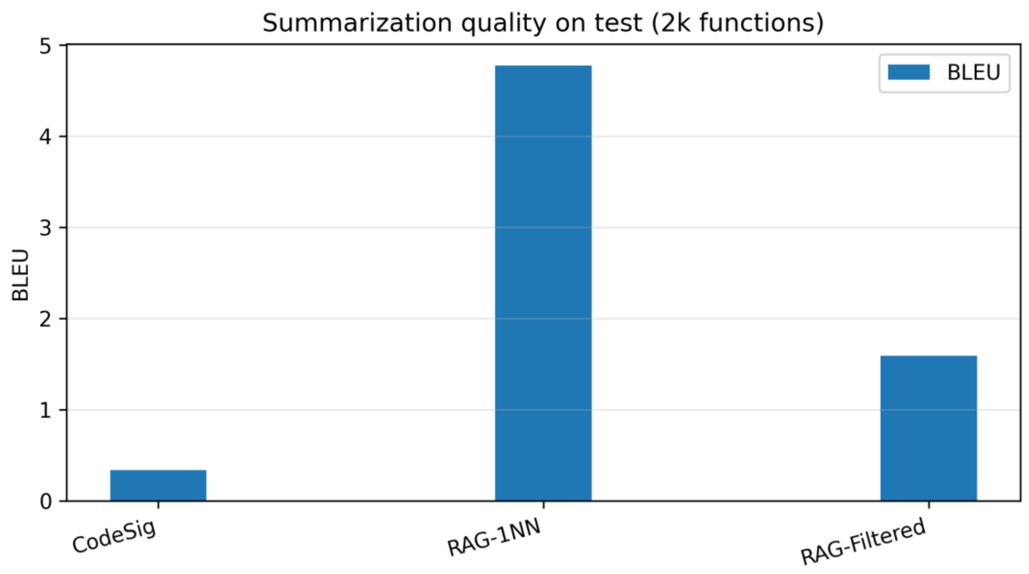

Fig. 7. BLEU scores for summarization methods (test functions).

Table VIII. End-to-end retrieval+summary metrics (summary compared to query target).

| Pipeline | ROUGE-1 | ROUGE-2 | ROUGE-L | BLEU |
|---|---|---|---|---|
| BM25+CodeSig | 0.2049 | 0.0558 | 0.1951 | 0.2987 |
| BM25+RAG-F | 0.2673 | 0.0546 | 0.2388 | 1.6911 |
| Fusion+RAG-F | 0.2695 | 0.0547 | 0.2408 | 1.6743 |

Table IX. Measured efficiency of retrieval and summarization components.

| Component | Index build (s) | Avg score time/query (ms) | Notes |
|---|---|---|---|
| Retriever (BM25) | 0.0274 | 9.0823 | rank_bm25 default k1=1.5, b=0.75 |
| Retriever (TF-IDF) | 0.1115 | 1.2550 | sklearn TfidfVectorizer max_features=50k |
| Rerank (Fusion α=0.9) | 0.0000 | 0.0200 | min-max normalize per query, linear fusion |
| RAG code index | 0.4574 | 0.7721 | TF-IDF over 20k train codes + cosine kNN (k=5) |

Table X. Manual consistency check on 30 sampled functions (single annotator).

| System | n | Correct | Partial | Incorrect | Correct% | Partial% | Incorrect % |
|---|---|---|---|---|---|---|---|
| CodeSig | 30 | 14 | 15 | 1 | 0.4667 | 0.5000 | 0.0333 |
| RAG-Filtered | 30 | 10 | 18 | 2 | 0.3333 | 0.6000 | 0.0667 |

Table XI. Representative qualitative examples from the manual check.

| ID | Function | Reference | CodeSig | RAG-Filtered |
|---|---|---|---|---|
| 69 | get_age | finds the age record for given site | get age | get age list |
| 264 | set_metadata_index_and_column_names | sets index and column names to gctx convention | set metadata index and column names | set metadata index and column names meta name columns df |
| 279 | dumps | dumps data into nicely formatted json string | dumps | dumps yaml data |
| 375 | pause | pause subscription and buffer the data for up to one hour | pause | pause subscription dict |
| 571 | relation_column | such as user username such as replies content | relation column | relation column fields instance |
| 1080 | _preprocess | zero mean unit variance normalization by default | preprocess | preprocess data mean labels std |

Table XII. Retrieval performance by query (docstring) token-length quartile (test pool).

| Query length quartile | Range | n | BM25 R@1 | BM25 R@10 | BM25 MRR@10 | Fusion R@1 | Fusion R@10 | Fusion MRR@10 |
|---|---|---|---|---|---|---|---|---|
| Q1 | [0, 8) | 405 | 0.4247 | 0.6519 | 0.4955 | 0.4222 | 0.6617 | 0.4928 |
| Q2 | [8, 19) | 572 | 0.5175 | 0.7360 | 0.5847 | 0.5245 | 0.7360 | 0.5877 |
| Q3 | [19, 47) | 517 | 0.7002 | 0.8627 | 0.7522 | 0.7021 | 0.8646 | 0.7550 |
| Q4 | [47, 5223] | 506 | 0.7767 | 0.9071 | 0.8211 | 0.7648 | 0.9111 | 0.8140 |

Table XIII. Retrieval performance by code token-length quartile (test pool).

| Code length quartile | Range | n | BM25 R@1 | BM25 R@10 | BM25 MRR@10 | Fusion R@1 | Fusion R@10 | Fusion MRR@10 |
|---|---|---|---|---|---|---|---|---|
| Q1 | [2, 19) | 495 | 0.4768 | 0.6848 | 0.5335 | 0.4869 | 0.6909 | 0.5428 |
| Q2 | [19, 32) | 502 | 0.5717 | 0.7968 | 0.6508 | 0.5737 | 0.8068 | 0.6532 |
| Q3 | [32, 59) | 503 | 0.7217 | 0.8648 | 0.7659 | 0.7157 | 0.8688 | 0.7601 |
| Q4 | [59, 1642] | 500 | 0.6740 | 0.8320 | 0.7268 | 0.6640 | 0.8260 | 0.7180 |

Table XIV. Oracle summarization ROUGE by code token-length quartile (test functions).

| Code length quartile | Range | n | CodeSig R1 | CodeSig R2 | CodeSig RL | RAG-Filtered R1 | RAG-Filtered R2 | RAG-Filtered RL |
|---|---|---|---|---|---|---|---|---|
| Q1 | [2, 19) | 495 | 0.2415 | 0.0456 | 0.2335 | 0.2727 | 0.0470 | 0.2513 |
| Q2 | [19, 32) | 502 | 0.2656 | 0.0770 | 0.2528 | 0.2911 | 0.0766 | 0.2646 |
| Q3 | [32, 59) | 503 | 0.2685 | 0.0698 | 0.2533 | 0.2957 | 0.0654 | 0.2666 |
| Q4 | [59, 1642] | 500 | 0.2481 | 0.0655 | 0.2376 | 0.2574 | 0.0514 | 0.2265 |

Table XV. Query–code token overlap ratio vs. BM25 retrieval success (test pool).

| Group | n | overlap_mean | overlap_median | overlap_p25 | overlap_p75 |
|---|---|---|---|---|---|
| BM25 rank=1 | 1223 | 0.3158 | 0.2708 | 0.1852 | 0.3921 |
| BM25 rank<=10 | 1590 | 0.2959 | 0.2500 | 0.1709 | 0.3690 |
| BM25 rank>10 | 410 | 0.1134 | 0.0969 | 0.0000 | 0.1538 |
| All queries | 2000 | 0.2585 | 0.2222 | 0.1353 | 0.3333 |

Table XVI. Per-query rank change from BM25 to fusion re-ranking (test pool).

| Effect | Queries | Percent |
|---|---|---|
| Improved (rank↓) | 236 | 0.1180 |
| Unchanged | 1588 | 0.7940 |
| Worsened (rank↑) | 176 | 0.0880 |
| Mean Δrank | | -0.2450 |

| | | |
|---|---|---|
| Median Δrank | | 0.0000 |

Table XVII. Summary length statistics (tokens) for targets and systems (test functions).

| Text | Avg tokens | Median | p95 |
|---|---|---|---|
| Reference (target) | 10.8135 | 9.0000 | 30.0000 |
| CodeSig | 2.2260 | 2.0000 | 4.0000 |
| RAG-Filtered | 4.7350 | 4.0000 | 9.0000 |

## Limitations

First, our retrieval experiments use a paired-pool protocol with N=2,000 sampled query–code pairs rather than ranking against the full test corpus. This protocol is widely used to control computational cost and is consistent with evaluation setups in neural code search work [2], [3], but it can overestimate performance relative to full-corpus retrieval because the negative set is limited. Second, our methods are deliberately lexical and lightweight. While this isolates retrieval–summary interactions, it does not represent the state of the art achieved by neural retrievers and encoder–decoder models trained on large corpora (e.g., CodeBERT/GraphCodeBERT/PLBART/CodeT5/UniXcoder) [2]–[6]. In particular, lexical retrievers struggle when the query uses synonyms not present in identifiers, and template summarization cannot infer intent that is not reflected in function names.

Third, our RAG summarizer uses TF-IDF kNN over a sampled memory of 20,000 training functions. Although the sampling is reproducible, it may miss rare domains and reduce the chance of retrieving truly similar neighbors. Increasing the memory size or using dense semantic retrieval (e.g., DPR-style embeddings) could improve neighbor quality but would depart from the lightweight design goal [9]. Fourth, we evaluate summaries against the first sentence of docstrings, which captures intent but may ignore important parameter and edge-case details contained in later docstring lines. Finally, the manual consistency check is small (30 samples) and performed by a single annotator; it is intended as a qualitative complement rather than a definitive human evaluation.

These limitations suggest that the reported numbers should be interpreted as rigorous baselines for retrieval–summary integration rather than as a replacement for neural systems. Nonetheless, the pipeline and measurements provide a reproducible reference point and a diagnostic tool for understanding how retrieval errors propagate into explanation quality.

## Conclusion

This paper investigated a retrieval–summary integrated pipeline for code intelligence that prioritizes “findable then explainable” behavior. Using a CodeSearchNet-derived Python subset with official splits, we conducted full empirical evaluations on (i) natural-language-to-function retrieval and (ii) function-to-docstring summarization, and we reported detailed metrics, curves, and efficiency measurements. BM25 over identifier-aware code tokens provided strong retrieval performance on a paired 2,000-function test pool (MRR@10=0.6697, Recall@10=0.7950), and a simple validation-tuned fusion re-ranking delivered small but consistent gains (Recall@10=0.7985). For summarization, a function-name template established a strong lexical baseline, while a lightweight kNN retrieval-augmented summarizer with token filtering improved ROUGE-1 and ROUGE-L and yielded better end-to-end ROUGE-1 when paired with retrieval.

The results emphasize that retrieval and explanation must be evaluated together: retrieval errors directly limit achievable explainability under user-intent metrics. Lightweight RAG is beneficial when constrained by code-derived signals, offering a practical and interpretable approach for CPU-only deployments or as a fallback in hybrid systems. Future work can extend this analysis by scaling to full-corpus retrieval, incorporating neural rerankers, and studying richer human evaluation of summary usefulness in developer workflows.

## References

[1] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, “CodeSearchNet Challenge: Evaluating the State of Semantic Code Search,” arXiv:1909.09436, 2019.

[2] Z. Feng et al., “CodeBERT: A Pre-Trained Model for Programming and Natural Languages,” in Proc. Findings of EMNLP, 2020.

[3] D. Guo et al., “GraphCodeBERT: Pre-Training Code Representations with Data Flow,” in Proc. ICLR, 2021.

[4] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, “Unified Pre-training for Program Understanding and Generation,” in Proc. NAACL-HLT, 2021.

[5] Y. Wang et al., “CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation,” in Proc. EMNLP, 2021.

[6] D. Guo et al., “UniXcoder: Unified Cross-Modal Pre-training for Code Representation,” arXiv:2203.03850, 2022.

[7] P. Lewis et al., “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks,” in Proc. NeurIPS, 2020.

[8] K. Guu, K. Lee, Z. Tung, P. Pasupat, and M. Chang, “REALM: Retrieval-Augmented Language Model Pre-Training,” in Proc. ICML, 2020.

[9] V. Karpukhin et al., “Dense Passage Retrieval for Open-Domain Question Answering,” in Proc. EMNLP, 2020.

[10] S. Robertson and H. Zaragoza, “The Probabilistic Relevance Framework: BM25 and Beyond,” Found. Trends Inf. Retr., vol. 3, no. 4, pp. 333–389, 2009.

[11] G. Salton and C. Buckley, “Term-Weighting Approaches in Automatic Text Retrieval,” Inf. Process. Manage., vol. 24, no. 5, pp. 513–523, 1988.

[12] C. D. Manning, P. Raghavan, and H. Schütze, Introduction to Information Retrieval. Cambridge, UK: Cambridge Univ. Press, 2008.

[13] K. Järvelin and J. Kekäläinen, “Cumulated Gain-Based Evaluation of IR Techniques,” ACM Trans. Inf. Syst., vol. 20, no. 4, pp. 422–446, 2002.

[14] C.-Y. Lin, “ROUGE: A Package for Automatic Evaluation of Summaries,” in Proc. ACL Workshop on Text Summarization Branches Out, 2004, pp. 74–81.

[15] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “BLEU: A Method for Automatic Evaluation of Machine Translation,” in Proc. ACL, 2002, pp. 311–318.

[16] G. V. Cormack, C. L. A. Clarke, and S. Buettcher, “Reciprocal Rank Fusion Outperforms Condorcet and Individual Rank Learning Methods,” in Proc. SIGIR, 2009, pp. 758–759.

[17] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, “Summarizing Source Code Using a Neural Attention Model,” in Proc. ACL, 2016, pp. 2073–2083.

[18] U. Alon, S. Brody, O. Levy, and E. Yahav, “code2seq: Generating Sequences from Structured Representations of Code,” in Proc. ICLR, 2019.

[19] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, “code2vec: Learning Distributed Representations of Code,” in Proc. POPL, 2019.

[20] A. Vaswani et al., “Attention Is All You Need,” in Proc. NeurIPS, 2017, pp. 5998–6008.

[21] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” in Proc. NAACL-HLT, 2019, pp. 4171–4186.

[22] A. S. Soliman, “CodeSearchNet,” Hugging Face Datasets, 2023. [Online]. Available: https://huggingface.co/datasets/AhmedSSoliman/CodeSearchNet

[23] T. Shirakawa, Y. Li, Y. Wu, S. Qiu, Y. Li, M. Zhao, H. Iso, and M. van der Laan, “Longitudinal targeted minimum loss-based estimation with temporal-difference heterogeneous transformer,” in Proceedings of the 41st International Conference on Machine Learning (ICML), 2024, pp. 45097–45113, Art. no. 1836.

[24] K. Xu, H. Zhou, H. Zheng, M. Zhu, and Q. Xin, “Intelligent classification and personalized recommendation of e-commerce products based on machine learning,” Proceedings of the 6th International Conference on Computing and Data Science (ICCDS), 2024.

[25] Q. Xin, Z. Xu, L. Guo, F. Zhao, and B. Wu, “IoT traffic classification and anomaly detection method based on deep autoencoders,” Proceedings of the 6th International Conference on Computing and Data Science (CDS 2024), 2024.

[26] B. Wang, Y. He, Z. Shui, Q. Xin, and H. Lei, “Predictive optimization of DDoS attack mitigation in distributed systems using machine learning,” Proceedings of the 6th International Conference on Computing and Data Science (CDS 2024), 2024, pp. 89–94.

[27] Z. S. Zhong and S. Ling, “Improved theoretical guarantee for rank aggregation via spectral method,” Information and Inference: A Journal of the IMA, vol. 13, no. 3, 2024.

[28] Z. Ling, Q. Xin, Y. Lin, G. Su, and Z. Shui, “Optimization of autonomous driving image detection based on RFAConv and triplet attention,” Proceedings

of the 2nd International Conference on Software Engineering and Machine Learning (SEML 2024), 2024.

[30] Hanqi Zhang, “Risk-Aware Budget-Constrained Auto-Bidding under First-Price RTB: A Distributional Constrained Deep Reinforcement Learning Framework”, JACS, vol. 4, no. 6, pp. 30–47, Jun. 2024, doi: 10.69987/JACS.2024.40603.

[31] Xinzhuo Sun, Jing Chen, Binghua Zhou, and Meng-Ju Kuo, “ConRAG: Contradiction-Aware Retrieval-Augmented Generation under Multi-Source Conflicting Evidence”, JACS, vol. 4, no. 7, pp. 50–64, Jul. 2024, doi: 10.69987/JACS.2024.40705.