ArkTS-CodeSearch: A Benchmark for Code Retrieval in the Emerging ArkTS Ecosystem

Anonymous ACL submission

Abstract

Emerging programming ecosystems remain underrepresented in code intelligence benchmarks, especially when their documentation mixes natural language, identifiers, and bilingual comments. Although ArkTS is a core language of the OpenHarmony ecosystem, research on ArkTS code intelligence has been limited by the lack of public datasets and standardized benchmarks. This paper presents ArkTS-CodeSearch, a large-scale open-source benchmark for ArkTS code retrieval. We collect ArkTS repositories from GitHub and Gitee, extract function–docstring pairs using *tree-sitter-arkts*, and construct a dataset of 24,452 pairs from 1,577 repositories. Based on this dataset, we define a docstring-to-code retrieval task in which natural language comments are used to retrieve the corresponding ArkTS functions. We evaluate lexical and dense retrieval methods, including BM25 and several open-source embedding models, and further fine-tune these models using ArkTS and TypeScript training data. Experimental results show that dense embedding models substantially outperform lexical retrieval, while ArkTS-specific fine-tuning and TypeScript→ArkTS adaptation further improve retrieval performance. Additional analyses on multilingual documentation, query/code length, and repository-level splitting highlight the challenges of code search in a bilingual and emerging programming ecosystem. To support reproducibility, we release the dataset, fine-tuned models, data construction code, and training/evaluation scripts to support. The main repository is available at `https://anonymous-hf.up.railway.app/a/e7vfxujmfu1y/`.

1 Introduction

OpenHarmony has emerged as a widely adopted open-source operating system for smart devices, and ArkTS serves as its core programming language for application development. As the OpenHarmony ecosystem continues to grow (Li et al., 2025), the demand for intelligent code understanding tools, such as code search, code recommendation, code translation (Liu et al., 2024; Zhou et al., 2025) and automated code analysis, has increased accordingly (Chen et al., 2025; Liu et al., 2025a,b). However, research on ArkTS code intelligence remains limited due to the absence of publicly available datasets and standardized evaluation benchmarks.

In recent years, significant progress has been made in code semantic search, driven by large-scale datasets and pre-trained code embedding models. Benchmarks such as CodeSearchNet (Husain et al., 2019) have enabled systematic evaluation of natural language–to–code retrieval for mainstream programming languages including Python, Java, and JavaScript, among others, along with subsequent datasets such as CodeXGLUE and AdvTest (Lu et al., 2021), which further expand task coverage and evaluation settings.

To address this gap, we construct the first large-scale ArkTS dataset for code retrieval. Our dataset is collected from both GitHub and Gitee platforms, covering a wide range of OpenHarmony-related repositories. We parse ArkTS source code into abstract syntax trees (ASTs) and extract function-level code units together with their associated natural language docstrings. The resulting dataset is processed and reformatted into HuggingFace-compatible structures, facilitating reproducibility and future research.

Based on this dataset, we define a docstring-to-code retrieval task, where natural language docstrings are used to retrieve corresponding ArkTS functions. We adopt the CodeSearchNet framework as a baseline retrieval protocol and conduct a comprehensive evaluation of existing open-source code embedding models on the ArkTS code search task. Furthermore, to better capture ArkTS-specific semantics, we fine-tune pre-trained embedding models using ArkTS training data, achieving improved

retrieval performance. The resulting benchmark reveals several patterns that are specific to ArkTS code search. Dense embedding models consistently outperform BM25, while ArkTS-specific fine-tuning yields substantial gains across all evaluated embedding models. TypeScript pre-adaptation is helpful for several models, but its effect depends on the underlying encoder. We further observe that multilingual documentation, input length, and repository-level splitting have visible effects on retrieval performance.

Contributions. Our contributions are as follows:

- We construct and release ArkTS-CodeSearch, to our knowledge the first large-scale public benchmark dedicated to ArkTS code retrieval, containing 24,452 function–docstring pairs from 1,577 GitHub and Gitee repositories.
- We provide a reproducible AST-based data construction pipeline, together with repository-level metadata, license information, filtering procedures, and both random and repository-level data splits.
- We benchmark lexical and dense retrieval methods on ArkTS-CodeSearch and show that ArkTS-specific fine-tuning substantially improves retrieval performance over zero-shot embedding models.
- We study TypeScript→ArkTS adaptation and show that related-language pre-adaptation can improve retrieval performance for several medium and large embedding models.
- We provide diagnostic analyses of multilingual documentation, query/code length, and split sensitivity, revealing key challenges in bilingual code retrieval for emerging programming ecosystems.

2 Related Work

2.1 ArkTS Language

ArkTS is the primary programming language of the OpenHarmony ecosystem, designed for cross-platform application development across distributed devices (Li et al., 2023). As a TypeScript-derived language used in Huawei's HarmonyOS ecosystem, ArkTS is increasingly important but remains underexplored in code intelligence research. Although large-scale code corpora such as The Stack (Kocetkov et al., 2022) cover hundreds of programming languages, emerging ecosystems such as OpenHarmony are still underrepresented. This limits systematic evaluation of language models on ArkTS-specific constructs.

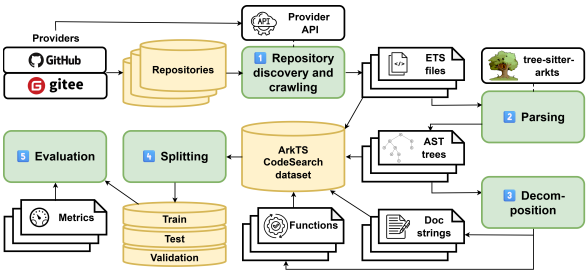

Figure 1: Workflow for curating ArkTS-CodeSearch.

This gap is becoming more consequential as HarmonyOS moves toward a more independent application ecosystem. Recent work on ArkTS static analysis and code generation highlights both the technical distinctiveness of ArkTS and the immaturity of existing tooling support (Chen et al., 2025; Zhou et al., 2025; Liu et al., 2024). Prior studies also report performance degradation when translating established Java or TypeScript programming patterns into ArkTS, suggesting that existing models have limited exposure to ArkTS-specific code.

A public ArkTS benchmark enables reproducible evaluation and lowers the entry barrier for research on underrepresented programming languages. By capturing the syntactic and semantic characteristics of ArkTS, it facilitates systematic investigation of code retrieval, cross-ecosystem transfer, and multilingual code representation learning.

2.2 Code Semantic Search

Code semantic search aims to retrieve relevant code snippets in response to a natural language query. Early approaches relied primarily on keyword matching or lexical similarity, which often fail to capture deeper semantic relationships between code and natural language. A commonly used method in this category is BM25 (Robertson and Zaragoza, 2009), a sparse retrieval model based on term frequency–inverse document frequency (TF-IDF) scoring.

Recent advances leverage neural models to learn joint embedding spaces for code and text, significantly improving retrieval performance. A range of pretrained models, such as CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2021), and CodeT5 (Wang et al., 2021), have demonstrated

strong performance on semantic code search tasks by exploiting large-scale multilingual and multi-language code corpora. Benchmarks such as CodeSearchNet (Husain et al., 2019), CodeXGLUE (Lu et al., 2021), and CoSQA (Huang et al., 2021) have standardized evaluation protocols for natural language–to–code retrieval and related code intelligence tasks, accelerating progress in semantic code search. However, these models and benchmarks predominantly focus on widely used languages, leaving emerging languages such as ArkTS underexplored. Consequently, there is a lack of systematic evaluation of code embedding models on ArkTS, highlighting the need for dedicated datasets and benchmarks to assess semantic code search performance in this setting.

3 ArkTS-CodeSearch Construction

This section describes the construction of ArkTS-CodeSearch and the retrieval evaluation setup. As shown in Figure 1, our pipeline includes repository discovery, source-code crawling, AST-based extraction, dataset preprocessing, and model training/evaluation.

3.1 Data Collection

Repository Discovery and Crawling. We construct ArkTS-CodeSearch from ArkTS repositories hosted on GitHub and Gitee, which together cover both community-driven and OpenHarmony-oriented projects. On GitHub, we search with the keyword *arkts* to collect community-developed ArkTS projects. On Gitee, we use the OpenHarmony exploration page to include repositories more closely related to the official OpenHarmony ecosystem.[1] After cross-platform deduplication, we obtain 1,577 unique repositories, including 560 from GitHub and 1,017 from Gitee.

For each repository, we recursively traverse the directory structure and retain files with the standard ArkTS extension *.ets*. Other files are discarded. For each retained source file, we record repository-level metadata, including the source platform, repository name, commit hash, and file path.

AST-Based Extraction. We parse each *.ets* file using *tree-sitter-arkts* and traverse the resulting AST to identify function- and method-level declarations. For each code unit, we record its byte offsets and recover the corresponding source span from the original file. We associate a docstring with a code unit when a line comment or block comment appears immediately before the declaration, allowing only whitespace between the comment and the code unit. The extracted docstrings are normalized by removing comment delimiters, leading comment markers, and redundant whitespace. We discard instances with empty docstrings, empty code bodies, or parsing failures. Each retained instance is stored together with repository-level metadata, including the source platform, repository name, commit hash, and file path. Additional background on AST parsing and its role in our pipeline is provided in Appendix A.

Filtering and Deduplication. After extraction, we apply filtering and deduplication to improve data quality. We remove instances with empty docstrings, empty code bodies, parsing failures, or trivial comments such as punctuation-only strings and boilerplate placeholders. Docstrings are normalized by removing comment delimiters, redundant whitespace, and line-prefix symbols.

We deduplicate repositories during collection and remove exact duplicate functions after extraction using hashes of normalized function bodies. The remaining records are aggregated into a unified dataset for retrieval evaluation.

For traceability, we retain repository-level metadata, including source platform, repository name, URL and detected license information; details are provided in Appendix B.

3.2 Data Splitting Protocol

We provide two data splitting protocols. The first is a standard random 8:1:1 split, following the common setup used in CodeSearchNet-style retrieval benchmarks. This split is mainly used for comparability with prior retrieval studies.

The second is a repository-level split, where all code units from the same repository are assigned to only one partition. This setting is designed to evaluate cross-project generalization and to reduce potential train–test contamination caused by project-specific naming conventions, duplicated utilities, or shared documentation templates. Unless otherwise stated, both splits use the same retrieval formulation and evaluation metrics.

3.3 Retrieval Task and Methods

We formulate ArkTS code search as a docstring-to-code retrieval task following the CodeSearchNet paradigm. Each data instance consists of a function-level ArkTS code snippet paired with its

[1] https://gitee.com/explore/harmony

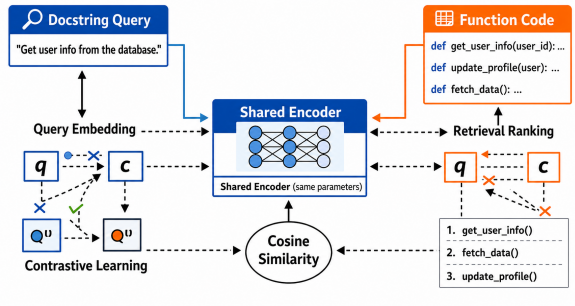

Figure 2: Overview of the CodeSearchNet-style retrieval framework.

corresponding docstring. Given a natural-language docstring as the query, a retrieval model ranks candidate ArkTS functions according to their relevance. In our experiments, each query has one corresponding target function in the candidate pool, and retrieval is performed over the full test set.

We evaluate two types of retrieval methods. First, we use BM25 as a lexical baseline. For Chinese-heavy ArkTS docstrings, we segment both queries and code snippets using the *jieba* tokenizer[2]. BM25 provides a simple sparse retrieval baseline based on lexical overlap.

Second, we evaluate dense semantic retrieval models. Both the query q and code snippet c are encoded into a shared embedding space, and candidate functions are ranked by cosine similarity: $\mathrm{sim}(q, c) = \frac{\mathbf{v}_q \cdot \mathbf{v}_c}{\|\mathbf{v}_q\| \|\mathbf{v}_c\|}$. This enables retrieval beyond exact lexical matching, which is important for ArkTS because its documentation often contains Chinese, English, and mixed-language comments.

As illustrated in Figure 2, dense retrieval models employ a shared encoder for both docstrings and code. During adaptation, contrastive learning aligns semantically matched docstring–function pairs in the embedding space, while retrieval is performed via nearest-neighbor ranking using cosine similarity.

3.4 Evaluation Metrics

We evaluate retrieval performance using standard ranking metrics (Järvelin and Kekäläinen, 2002; Manning et al., 2008), including Mean Reciprocal Rank (MRR), Normalized Discounted Cumulative Gain (NDCG), Recall@1, and Recall@5. MRR measures how highly the correct ArkTS function is ranked, while NDCG evaluates overall ranking quality with greater emphasis on top-ranked results. Recall@1 and Recall@5 indicate whether the target function is retrieved at the first position or within the top five candidates, respectively.

Because each query corresponds to a single target function, these metrics directly capture a model's ability to rank the correct code snippet ahead of competing candidates. Following prior CodeSearchNet-style studies, we use the random split as the primary evaluation setting to ensure comparability, while the repository-level split serves as a complementary setting for assessing cross-project generalization robustness. Detailed metric definitions are provided in Appendix E.

3.5 Model Selection

For dense retrieval, we select representative publicly available embedding models with reference to the MTEB leaderboard (Muennighoff et al., 2023), covering different model scales and language orientations. *Qwen3-Embedding-0.6B* (Zhang et al., 2025) is a recent Qwen-series embedding model designed for text embedding and retrieval, and represents a strong large-scale multilingual baseline in our experiments. *embeddinggemma-300m* (Schechter Vera et al., 2025) is a lightweight embedding model based on the Gemma family, allowing us to test whether a compact model can remain competitive on ArkTS retrieval. *bge-m3* (Chen et al., 2024) is a multilingual and multi-functionality embedding model that supports dense retrieval and has strong coverage across many languages, making it suitable for bilingual ArkTS documentation. We also include *bge-base-zh-v1.5* and *bge-base-en-v1.5* (Xiao et al., 2023) as Chinese- and English-oriented BGE baselines, respectively, to examine the effect of language orientation. Finally, *e5-base-v2* (Wang et al., 2022) is included as a widely used English text embedding model for retrieval. Together, these models allow us to study how multilingual, Chinese-oriented, and English-oriented embedding models perform on ArkTS code retrieval, where Chinese comments, English identifiers, and mixed-language documentation often coexist.

3.6 Model Adaptation

To study whether ArkTS-specific supervision improves retrieval performance, we fine-tune pre-trained embedding models using supervised contrastive learning. Each training instance consists of a docstring and its corresponding ArkTS function. We adopt the SentenceTransformer framework (Thakur et al., 2021; Reimers and Gurevych,

[2]https://github.com/fxsjy/jieba

2020, 2019) and train models with Multiple Negatives Ranking Loss, which encourages matched docstring–function pairs to be close in the embedding space while treating other in-batch examples as implicit negatives.

We consider three training configurations. First, ARKTS-TRAINING fine-tunes models using only ArkTS function–docstring pairs. Second, TS-TRAINING uses TypeScript function–docstring pairs from The Stack-v2 (Lozhkov et al., 2024) as related-language supervision. Third, TS→ARKTS-TRAINING first adapts models on TypeScript data and then further fine-tunes them on ArkTS data. This setting tests whether TypeScript can serve as an intermediate adaptation source for ArkTS retrieval. Detailed loss definitions and hyperparameters are provided in Appendix F.

4 Dataset Characteristics

This section summarizes the key characteristics of ArkTS-CodeSearch. More detailed statistics, distribution plots, leakage-audit results, and manual quality validation are provided in Appendix C.

4.1 Scale and Source Composition

To better capture the ArkTS ecosystem diversity, we include both community-driven projects and OpenHarmony-oriented repositories. ArkTS-CodeSearch comprises **24,452** function–docstring pairs collected from **1,577** open-source repositories across GitHub and Gitee. In total, 12,792 functions (52.3%) originate from Gitee and 11,660 functions (47.7%) from GitHub. This dual-source collection strategy mitigates platform-specific bias and improves coverage of heterogeneous coding conventions and documentation styles.

4.2 Structural and Linguistic Diversity

ArkTS-CodeSearch contains functions with diverse implementation complexity. The distributions of function length, character count, and AST node count are long-tailed, indicating that the dataset includes both short utility functions and more complex code units. This diversity provides retrieval cases with different levels of semantic and structural difficulty. Detailed complexity distributions are shown in Appendix C.1.

The dataset also exhibits substantial multilingual diversity. Its docstrings include monolingual Chinese, monolingual English, and mixed Chinese–English comments. This reflects realistic documentation practices in the OpenHarmony ecosystem, where Chinese descriptions, English identifiers, and bilingual API comments often coexist. The language distribution is stable across training, validation, and test splits, suggesting that the partitioning process does not introduce strong language imbalance. Detailed language statistics are reported in Appendix C.3.

4.3 Splitting Protocols and Leakage Control

We provide two splitting protocols. The first is a standard random 8:1:1 split, yielding 19,561 training samples, 2,445 validation samples, and 2,446 test samples. This split follows common CodeSearchNet-style evaluation practice and supports comparison with prior retrieval benchmarks.

The second is a repository-level split, where all functions from the same repository are assigned to the same partition. This setting is designed to evaluate cross-project generalization and reduce potential contamination caused by project-specific naming conventions, repeated utility functions, or shared documentation templates.

To assess potential train–test contamination, we conduct a multi-level leakage audit covering repository overlap, exact docstring overlap, near-duplicate code, and exact function-body duplication. The repository-level split eliminates repository contamination and reduces incidental overlap. Importantly, no exact function-body duplication is found across partitions under either split. Detailed leakage-audit results are provided in Appendix D.

4.4 Data Quality

To estimate the semantic quality of the extracted function–docstring pairs, two authors independently annotated 100 randomly sampled instances as either semantically aligned or not aligned. A pair was considered aligned if the docstring described the main functionality or intent of the corresponding code unit. The annotators agreed on 97 out of 100 instances, reaching substantial inter-annotator agreement with Cohen's $\kappa = 0.78$. Disagreements were resolved through discussion. After adjudication, 93 out of 100 sampled pairs were judged to be semantically aligned. The remaining cases mainly involved incomplete descriptions, or comments attached to nearby but not fully corresponding code units. This suggests that the AST-based extraction pipeline provides reliable supervision overall, while leaving room for future improvements in comment–code alignment and noise filtering.

Overall, these statistics demonstrate that ArkTS-CodeSearch provides a diverse and practically grounded benchmark for ArkTS code retrieval. Its multilingual documentation, long-tailed code complexity, and publicly released split protocols make it well suited for studying code search in an emerging bilingual programming ecosystem. At the same time, the presence of residual template overlap and near-duplicate utility functions underscores the importance of reporting both random-split and repository-level results in future evaluations.

5 Experiments and Results

We now evaluate the retrieval methods described in Section 3.3 on ArkTS-CodeSearch. We first compare zero-shot lexical and dense retrievers, then analyze multilingual behavior, fine-tuning strategies, and robustness under repository-level splitting.

5.1 Zero-Shot Retrieval Performance

Table 1 reports zero-shot retrieval performance on ArkTS-CodeSearch. We compare open-source dense embedding models with a BM25 baseline using the *jieba* tokenizer. Among dense models, *Qwen3-Embedding-0.6B* achieves the best performance, with an MRR of 0.6776 and Recall@5 of 0.7723, followed by *embeddinggemma-300m* and *bge-m3*. The strong performance of *embeddinggemma-300m*, despite its smaller size, suggests that model architecture and pretraining data quality matter in addition to parameter scale.

Among models of approximately 100M parameters, *bge-base-zh-v1.5* outperforms English-centric alternatives such as *bge-base-en-v1.5* and *e5-base-v2*, which is consistent with the substantial presence of Chinese docstrings in the dataset. BM25 performs substantially worse than dense models, with an MRR of 0.2043, NDCG of 0.2204, Recall@1 of 0.1643, and Recall@5 of 0.2690. These results show that ArkTS retrieval benefits from multilingual and code-aware representations, while lexical matching alone is insufficient.

5.2 Multilingual Performance Analysis

Given the bilingual nature of ArkTS documentation, we analyze retrieval performance by docstring language category. Table 2 reports MRR scores for monolingual Chinese, monolingual English, and mixed Chinese–English queries. Four test instances are excluded from this language-specific analysis because their docstrings cannot be reliably assigned to any predefined language category.

Mixed-language queries generally achieve higher MRR than monolingual queries across most models. This suggests that bilingual docstrings may provide complementary retrieval cues by combining natural-language descriptions with English identifiers, API names, or framework-specific terms. However, the balanced mixed-language category contains only 43 samples, so its scores should be interpreted cautiously.

Model pretraining also affects language-specific behavior. Chinese-oriented models are more competitive on Chinese-heavy queries, while English-centric models perform better on English-heavy queries. Multilingual models such as *Qwen3-Embedding-0.6B*, *embeddinggemma-300m*, and *bge-m3* show more stable performance across language categories.

These results indicate that ArkTS-CodeSearch is not merely a language-specific code retrieval benchmark, but also a realistic bilingual retrieval setting. Strong performance requires models to jointly handle code semantics, Chinese–English documentation, and identifier-level cues.

5.3 Fine-Tuning

Table 1 summarizes the fine-tuning results. Across all models, ArkTS-specific fine-tuning improves retrieval performance over the corresponding zero-shot baselines. This confirms that task-aligned supervision helps bridge the semantic gap between natural-language docstrings and ArkTS code.

Training data composition has a clear impact on retrieval quality. Models fine-tuned on ArkTS data consistently outperform those trained only on TypeScript data, showing the importance of in-domain supervision. At the same time, the TypeScript→ArkTS two-stage strategy further improves performance for most medium and large models. This suggests that related-language adaptation can provide useful intermediate supervision before domain-specific fine-tuning.

Among all evaluated models, *embeddinggemma-300m* achieves the strongest fine-tuned performance, reaching an MRR of 0.8013 under the TypeScript→ArkTS setting. Notably, this mid-sized model outperforms larger alternatives such as *Qwen3-Embedding-0.6B*, indicating that effective adaptation can be as important as parameter scale. Nevertheless, larger pretrained models also benefit substantially from ArkTS-specific supervision.

Multilingual and Chinese-oriented models, such as *bge-m3* and *bge-base-zh-v1.5*, obtain stable

(a) Original vs. Fine-tuned on ARKTS-TRAINING

Models	Params	Original				Fine-tuned (ARKTS-TRAINING)			
		MRR	NDCG	Recall@1	Recall@5	MRR	NDCG	Recall@1	Recall@5
embeddinggemma-300m	308M	0.6399	0.6654	0.5740	0.7416	**0.7788**	**0.8034**	**0.7142**	**0.8769**
Qwen3-Embedding-0.6B	596M	**0.6776**	**0.7015**	**0.6141**	**0.7723**	0.7745	0.7971	0.7146	0.8643
bge-m3	567M	0.5283	0.5603	0.4464	0.6558	0.7467	0.7729	0.6750	0.8504
bge-base-zh-v1.5	110M	0.3598	0.3903	0.2841	0.4816	0.6748	0.7055	0.5961	0.7972
bge-base-en-v1.5	110M	0.3439	0.3637	0.2935	0.4227	0.4866	0.5144	0.4195	0.5977
e5-base-v2	110M	0.3073	0.3261	0.2596	0.3823	0.4855	0.5135	0.4158	0.5973
BM25 (jieba tokenizer)	–	0.2043	0.2204	0.1643	0.2690	–	–	–	–

(b) Fine-tuned Models under Different Training Datasets

Models	Params	TS-TRAINING				TS→ARKTS-TRAINING			
		MRR	NDCG	Recall@1	Recall@5	MRR	NDCG	Recall@1	Recall@5
embeddinggemma-300m	308M	0.6259	0.6525	0.5572	0.7318	**0.8013**	**0.8229**	**0.7412**	**0.8867**
Qwen3-Embedding-0.6B	596M	**0.6522**	**0.6785**	**0.5809**	**0.7563**	0.7708	0.7947	0.7065	0.8655
bge-m3	567M	0.6051	0.6338	0.5311	0.7191	0.7581	0.7846	0.6872	0.8630
bge-base-zh-v1.5	110M	0.5270	0.5571	0.4509	0.6468	0.6759	0.7032	0.6018	0.7841
e5-base-v2	110M	0.3751	0.3955	0.3226	0.4567	0.4557	0.4821	0.3904	0.5613
bge-base-en-v1.5	110M	0.3834	0.4047	0.3299	0.4681	0.4539	0.4770	0.3949	0.5457

Table 1: Retrieval performance of original and fine-tuned embedding models under different training settings. Panel (a) compares open-source models before and after finetuning on ARKTS-TRAINING, while Panel (b) compares fine-tuned models trained on TS-TRAINING and TS→ARKTS-TRAINING.

Model	**Pure**		**Mixed**		
	CN	**EN**	**Bal.**	**CN**	**EN**
Sample Count	785	626	43	290	698
embeddinggemma-300m	0.6459	0.5917	**0.8721**	0.8524	0.8527
Qwen3-Embedding-0.6B	**0.6539**	**0.6344**	0.8543	**0.8579**	**0.8781**
bge-m3	0.5492	0.5672	0.7965	0.7408	0.7101
bge-base-zh-v1.5	0.3569	0.4179	0.7120	0.6033	0.5448
e5-base-v2	0.0143	0.5591	0.5667	0.3907	0.5871
bge-base-en-v1.5	0.0544	0.5790	0.5969	0.4279	0.6015

Table 2: MRR results by docstring language category.

gains after fine-tuning, while English-centric models remain less competitive. This trend is consistent with the multilingual composition of ArkTS-CodeSearch, where many docstrings contain Chinese or mixed Chinese–English text.

Overall, these results show that ArkTS code retrieval benefits most from in-domain supervision, while TypeScript→ArkTS adaptation provides a useful but model-dependent transfer strategy.

5.4 Cross-Project Robustness Analysis

To assess cross-project robustness, we evaluate models under a repository-level split, where all functions from the same repository are assigned to a single partition. This setting reduces train–test contamination from project-specific naming conventions, repeated utilities, or shared documentation templates. Table 4 compares MRR under the random and repository-level splits.

Zero-shot models show clear drops under the repository-level split, suggesting that random splitting supports CodeSearchNet-style comparability but may be optimistic for cross-project generalization. After ArkTS fine-tuning, all models recover substantially. Some fine-tuned models even achieve higher MRR under the repository-level split, which likely reflects differences in test-set composition rather than an easier protocol. We therefore treat repository-level splitting as a complementary robustness analysis, not a replacement for the random-split main track. We release both splitting configurations to support flexible experimental designs.[3]

5.5 Diagnostic Analysis

To better understand model behavior beyond aggregate scores, we analyze the effects of input length (query and code) and model size.

5.5.1 Effect of Query and Code Length

We group test instances by docstring length and function-body length. Table 3 shows that retrieval performance varies with both docstring length and code length. Very short docstrings often lack discriminative information, while very long docstrings may introduce auxiliary details, multiple intents, or noise. By contrast, performance is relatively stable across code-length groups, except that very short functions provide limited semantic context. These results suggest that query-side processing, such as rewriting, compression, or noise filtering, may improve ArkTS code retrieval.

5.5.2 Effect of Model Size

Figure 3 presents the corresponding results. Larger models generally achieve stronger performance

[3] https://anonymous-hf.up.railway.app/a/bg8yaq3zj3ot/

	Code				Docstring			
	Short	**Med**	**Long**	**Very-Long**	**Short**	**Med**	**Long**	**Very-Long**
Sample Count	437	769	466	774	979	602	407	458
Avg. Length (chars)	68.0	180.2	430.6	1948.1	8.8	32.4	69.6	301.6
embeddinggemma-300m	0.6187	0.7875	0.8460	0.7617	0.6140	0.7656	**0.9029**	0.7019
Qwen3-Embedding-0.6B	**0.6512**	0.4759	**0.8132**	**0.7753**	**0.6344**	**0.8085**	**0.9181**	**0.7065**
bge-m3	0.5380	**0.6591**	0.7468	0.6366	0.5347	0.6132	0.8344	0.6213
bge-base-zh-v1.5	0.3934	0.4759	0.5994	0.5074	0.3549	0.4623	0.6622	0.5194
e5-base-v2	0.3528	0.3963	0.4076	0.3721	0.0790	0.4162	0.7418	0.6362
bge-base-en-v1.5	0.3777	0.4225	0.4667	0.4070	0.1171	0.4741	0.7527	0.6309

Table 3: MRR by function code length and docstring length (query).

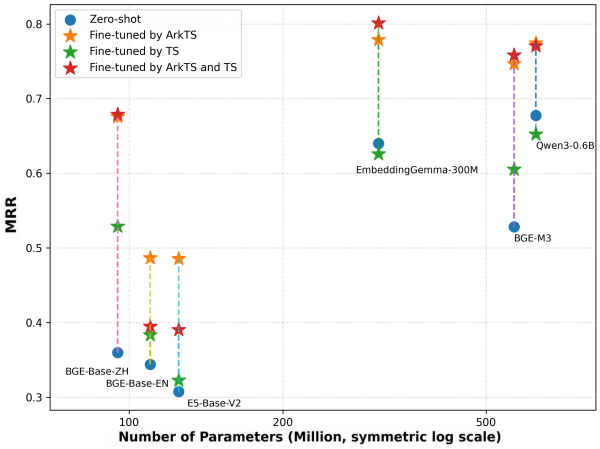

Figure 3: Model size versus MRR before and after fine-tuning on the ArkTS dataset. A small horizontal jitter is applied for clarity of visualization.

	Random		Repo	
Model	**Base**	**Tune**	**Base**	**Tune**
embeddinggemma-300m	0.6399	0.7788	0.4564	0.8507
Qwen3-Embedding-0.6B	0.6776	0.7745	0.4804	0.8015
bge-m3	0.5283	0.7467	0.3815	0.7767
bge-base-zh-v1.5	0.3598	0.6748	0.2579	0.6985
bge-base-en-v1.5	0.3439	0.4866	0.2247	0.4810
e5-base-v2	0.3073	0.4855	0.1994	0.4814

Table 4: MRR under the random-split main track and the repository-level robustness track. Base denotes zero-shot evaluation. For each split, Tune denotes fine-tuning on the corresponding training partition.

in the zero-shot setting; however, after ArkTS-specific fine-tuning, several medium-sized models become competitive with or even outperform larger models. This finding suggests that domain-specific supervision and multilingual, code-aware pretraining can be as important as model scale, highlighting the value of task-specific data construction for emerging programming languages.

6 Conclusion

This paper presented *ArkTS-CodeSearch*, the first large-scale public benchmark for ArkTS code retrieval. Our results show that dense retrieval models substantially outperform lexical baselines, while ArkTS-specific fine-tuning consistently improves performance beyond zero-shot settings. We further demonstrate that TypeScript→ArkTS transfer can yield additional retrieval gains for several embedding models. Overall, ArkTS-CodeSearch provides a reproducible foundation for code retrieval research in the OpenHarmony ecosystem and advances code intelligence for underrepresented programming languages.

Challenges & Opportunities. ArkTS retrieval remains a challenging problem. First, current embedding models must bridge Chinese, English, and mixed-language documentation with ArkTS code semantics, rather than relying primarily on English identifier overlap. Second, retrieval quality is highly sensitive to query specificity, since short docstrings are often ambiguous and underspecified, whereas long docstrings may contain multiple intents, implementation details, or irrelevant contextual noise. Third, repository-level generalization remains challenging because random-split evaluation can implicitly benefit from duplicated utilities, repository-specific naming conventions, and recurring documentation templates.

These observations open several promising research directions. Future work may explore multilingual query normalization and rewriting, ArkTS-aware pretraining and representation learning, cross-language transfer from TypeScript and related ecosystems, and hard-negative mining strategies based on semantically similar APIs or utility functions. Beyond standalone retrieval benchmarks, ArkTS-CodeSearch can also support retrieval-augmented coding agents, where ArkTS code search serves as a callable tool for locating relevant functions, APIs, usage patterns, and repository-specific implementations. More broadly, we hope this benchmark encourages future research on robust retrieval infrastructure for emerging and underrepresented programming ecosystems.

Limitations

This work has several limitations. First, ArkTS-CodeSearch focuses on a controlled docstring-to-code retrieval formulation. This setting follows the CodeSearchNet-style paradigm and provides a standardized entry point for evaluating ArkTS code retrieval, but it does not cover the full range of code intelligence scenarios. Future extensions can incorporate more challenging retrieval settings, such as larger and more open-ended candidate pools, hard negatives, semantically similar APIs, commit-message-to-code retrieval, and code-change description retrieval.

Second, although ArkTS-CodeSearch is collected from both GitHub and Gitee, it is still derived from publicly available repositories. As a result, the benchmark mainly reflects open-source ArkTS development practices and may underrepresent private, industrial, or closed-source ArkTS projects. Models evaluated on this benchmark should therefore be interpreted as measuring performance within the observable open-source ArkTS ecosystem rather than the entire ArkTS software landscape.

Third, the extracted function–docstring pairs inevitably contain some noise. Our manual validation suggests that most pairs are semantically aligned, but some comments are overly generic, incomplete, or only partially related to the extracted code unit. This reflects realistic documentation noise in open-source repositories, while future versions of the dataset can further improve comment–code alignment through stronger parsing heuristics, semantic filtering, or human-in-the-loop validation.

Finally, our main benchmark track adopts a random split to remain comparable with CodeSearchNet-style retrieval evaluations and prior code search studies. However, random splitting may allow functions from the same repository to appear in different partitions, which can introduce repository-level similarities through naming conventions, shared templates, or repeated utility patterns. To make this limitation explicit, we additionally provide a repository-level split and report leakage-audit statistics covering repository overlap, exact docstring overlap, near-duplicate code, and exact function-body duplication. We view the repository-level split as a complementary robustness setting rather than a replacement for the main random-split track, and encourage future work to report both settings when evaluating ArkTS retrieval models.

References

Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4).

Max Brunsfeld. 2020. Tree-sitter: Incremental parsing system. https://tree-sitter.github.io/.

Haonan Chen, Daihang Chen, Yizhuo Yang, Lingyun Xu, Liang Gao, Mingyi Zhou, Chunming Hu, and Li Li. 2025. Arkanalyzer: The static analysis framework for openharmony. *Preprint*, arXiv:2501.05798.

Jianlv Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. 2024. Bge m3-embedding: Multi-lingual, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. *Preprint*, arXiv:2402.03216.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcodebert: Pre-training code representations with data flow. *Preprint*, arXiv:2009.08366.

Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. 2021. Cosqa: 20,000+ web queries for code search and question answering. *Preprint*, arXiv:2105.13239.

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

Kalervo Järvelin and Jaana Kekäläinen. 2002. Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.*, 20(4):422–446.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2022. The stack: 3 tb of permissively licensed source code. *Preprint*, arXiv:2211.15533.

Li Li, Xiang Gao, Hailong Sun, Chunming Hu, Carolyn Sun, Haoyu Wang, Haipeng Cai, Ting Su, Xiapu Luo, Tegawendé Bissyande, Jacques Klein, John

Grundy, Tao Xie, Haibo Chen, and Huaimin Wang. 2025. Software engineering for openharmony: A research roadmap. *ACM Comput. Surv.*, 58(2).

Li Li, Xiang Gao, Hailong Sun, Chunming Hu, Xiaoyu Sun, Haoyu Wang, Haipeng Cai, Ting Su, Xiapu Luo, Tegawendé F. Bissyandé, Jacques Klein, John Grundy, Tao Xie, Haibo Chen, and Huaimin Wang. 2023. Software engineering for openharmony: A research roadmap. *Preprint*, arXiv:2311.01311.

Farong Liu, Mingyi Zhou, Yakun Zhang, Ting Su, Bo Sun, Jacques Klein, Xiang Gao, and Li Li. 2025a. Haptest: The dynamic analysis framework for openharmony. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, FSE Companion '25, page 422–431, New York, NY, USA. Association for Computing Machinery.

Mingwei Liu, Zheng Pei, Yanlin Wang, Zihao Wang, Zikang Li, Enci Lin, Xin Peng, and Zibin Zheng. 2025b. Framework-aware code generation with api knowledge graph-constructed data: A study on harmonyos. *Preprint*, arXiv:2512.00380.

Runlin Liu, Yuhang Lin, Yunge Hu, Zhe Zhang, and Xiang Gao. 2024. Llm-based java concurrent program to arkts converter. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 2403–2406, New York, NY, USA. Association for Computing Machinery.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, and 47 others. 2024. Starcoder 2 and the stack v2: The next generation. *Preprint*, arXiv:2402.19173.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, and 3 others. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *Preprint*, arXiv:2102.04664.

C.D. Manning, P. Raghavan, and H. Schütze. 2008. *Introduction to Information Retrieval*. Cambridge University Press.

Niklas Muennighoff, Nouamane Tazi, Loïc Magne, and Nils Reimers. 2023. Mteb: Massive text embedding benchmark. *Preprint*, arXiv:2210.07316.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

Nils Reimers and Iryna Gurevych. 2020. Making monolingual sentence embeddings multilingual using knowledge distillation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

Stephen Robertson and Hugo Zaragoza. 2009. The probabilistic relevance framework: Bm25 and beyond. *Found. Trends Inf. Retr.*, 3(4):333–389.

Henrique Schechter Vera, Sahil Dua, Biao Zhang, Daniel Salz, Ryan Mullins, Sindhu Raghuram Panyam, Sara Smoot, Iftekhar Naim, Joe Zou, Feiyang Chen, Daniel Cer, Alice Lisak, Min Choi, Lucas Gonzalez, Omar Sanseviero, Glenn Cameron, Ian Ballantyne, Kat Black, Kaifeng Chen, and 69 others. 2025. Embeddinggemma: Powerful and lightweight text representations.

Nandan Thakur, Nils Reimers, Johannes Daxenberger, and Iryna Gurevych. 2021. Augmented SBERT: Data augmentation method for improving bi-encoders for pairwise sentence scoring tasks. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 296–310, Online. Association for Computational Linguistics.

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*.

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

Shitao Xiao, Zheng Liu, Peitian Zhang, and Niklas Muennighoff. 2023. C-pack: Packaged resources to advance general chinese embedding. *Preprint*, arXiv:2309.07597.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. 2025. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*.

Bo Zhou, Jiaqi Shi, Ying Wang, Li Li, Tsz On Li, Hai Yu, and Zhiliang Zhu. 2025. Porting software libraries to openharmony: Transitioning from typescript or javascript to arkts. *Proc. ACM Softw. Eng.*, 2(ISSTA).

A AST Parser Background

AST parsers provide a structured and hierarchical representation of source code, explicitly encoding syntactic structure while implicitly capturing semantic relationships. AST-based representations

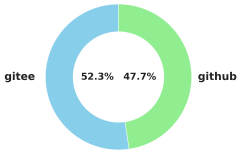

Figure 4: Distribution of source platforms.

have been widely adopted in code intelligence research, supporting tasks such as semantic code search, program analysis, and code summarization (Allamanis et al., 2018).

Compared with heuristic or regular-expression-based parsing, AST parsers offer improved robustness and syntactic correctness across diverse coding styles. Modern incremental parsers, such as Tree-sitter (Brunsfeld, 2020), enable scalable and language-aware code analysis and have been successfully applied to multiple programming languages. In this work, we leverage *tree-sitter-arkts* to parse ArkTS source files into ASTs, enabling accurate extraction of function-level code units along with their associated docstrings. This structured preprocessing facilitates reliable code unit alignment and serves as a clean foundation for downstream code representation learning and evaluation.

B Licensing and Source Metadata

ArkTS-CodeSearch is constructed from publicly available repositories on GitHub and Gitee. These repositories may adopt different licenses, including permissive, copyleft, or unspecified licenses. To improve traceability, we release repository-level metadata together with the dataset, including the source platform, repository name, commit hash, file path, and detected license fields in *repos_license_*.jsonl*.

The released license metadata is intended to help users inspect the provenance of individual examples and determine whether additional filtering is needed for their use case. Users should consult the original repository licenses and the provided metadata before using the dataset in production or commercial settings.

C Additional Dataset Statistics

This appendix provides additional statistical analysis of ArkTS-CodeSearch. These statistics are intended to complement the main paper by characterizing the structural complexity, documentation patterns, and linguistic diversity of the dataset in more detail.

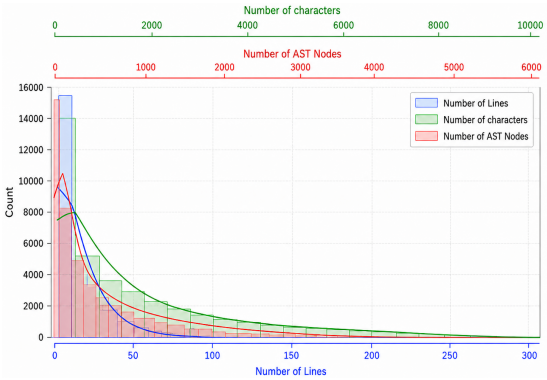

Figure 5: Distribution of ArkTS function complexity measured by lines, characters, and AST nodes.

C.1 Code Complexity Distribution

We analyze the complexity of extracted ArkTS code units from three perspectives: the number of lines, the number of characters, and the number of AST nodes. These measures provide complementary views of implementation complexity. Line and character counts reflect the surface-level length of a function, while AST node counts approximate its syntactic structure.

As shown in Figure 5, the distributions are strongly right-skewed. Most extracted functions are relatively short and structurally simple, while a small number of functions contain substantially longer implementations and larger ASTs. This pattern is common in real-world software repositories, where many utility functions are concise, whereas core modules, lifecycle handlers, or interface-related functions may involve more complex logic.

The long-tailed distribution suggests that ArkTS-CodeSearch contains retrieval examples with different levels of implementation difficulty. Short functions may provide limited semantic context, making them difficult to distinguish from other snippets. In contrast, longer functions may contain richer semantic cues but can also introduce irrelevant details. This diversity makes the dataset suitable for evaluating code retrieval models under realistic code-complexity variation.

C.2 Documentation Length Distribution

We further examine the length distribution of docstrings in ArkTS-CodeSearch. Docstring length is measured by the number of characters after comment normalization.

Figure 6 shows that most docstrings are short, typically falling within a compact range. However, the distribution also contains a secondary concen-

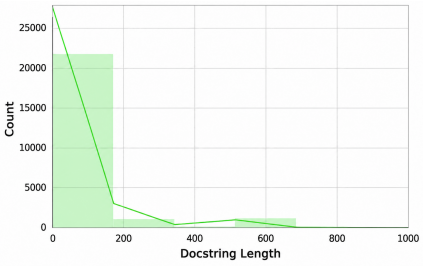

Figure 6: Distribution of docstring lengths.

Group	Type	Train	Val	Test	All
Pure	Chinese	32.48	31.78	32.09	32.37
	English	26.31	28.02	25.59	26.41
Mixed	CN-dom.	12.91	11.78	11.86	12.69
	EN-dom.	26.61	26.63	28.54	26.81
	Balanced	1.49	1.60	1.76	1.53
Contains	Chinese	*46.88*	*45.15*	*45.71*	*46.59*
	English	*54.42*	*56.24*	*55.89*	*54.75*

Table 5: Language distribution across dataset splits. Values are percentages.

tration of longer comments. This indicates heterogeneous documentation practices in ArkTS repositories. Many functions are documented with concise descriptions, while some public interfaces, UI components, or core logic functions receive more detailed comments.

This variation is important for retrieval evaluation. Very short docstrings may be underspecified and lack discriminative information, while very long docstrings may contain multiple intents, implementation notes, or auxiliary explanations. Therefore, the dataset naturally reflects both concise and verbose query scenarios.

C.3 Multilingual Documentation Composition

ArkTS-CodeSearch contains substantial multilingual documentation. In addition to monolingual Chinese and monolingual English docstrings, many comments contain mixed Chinese–English text. This reflects the bilingual nature of the OpenHarmony and ArkTS development ecosystem.

The language distribution remains stable across the training, validation, and test partitions, suggesting that the split does not introduce strong language imbalance. The coexistence of Chinese, English, and mixed-language comments makes ArkTS-CodeSearch different from many existing code search benchmarks that focus primarily on English documentation. As a result, strong performance on this benchmark requires both code

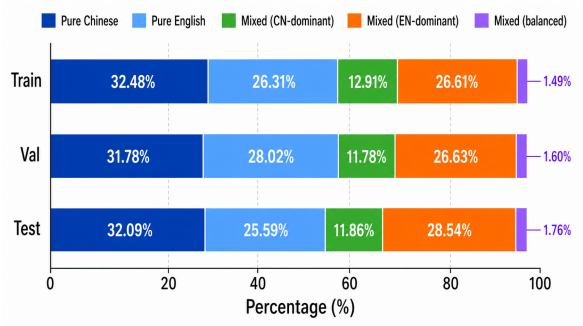

Figure 7: Distribution of language

understanding and multilingual natural-language understanding.

D Leakage Audit Details

To better understand potential train–test contamination, we conduct a multi-level leakage audit under both the random split and the repository-level split. The audit considers repository overlap, exact docstring overlap, near-duplicate code, and exact function-body duplication.

D.1 Repository Contamination

Repository contamination measures whether test samples originate from repositories that also appear in the training set. Under a random split, this type of overlap is expected because functions from the same repository can be assigned to different partitions. However, such overlap may lead to overly optimistic evaluation if models benefit from project-specific naming conventions, repeated utility functions, or similar documentation styles.

The repository-level split eliminates this issue by assigning all functions from the same repository to a single partition. Therefore, repository-level evaluation provides a stricter protocol for measuring cross-project generalization.

D.2 Exact Docstring Overlap

Exact docstring overlap measures whether normalized docstrings in the test set also appear in the training set. Some overlap remains even under the repository-level split. Manual inspection suggests that these cases often correspond to common template-style comments, such as generic API descriptions, lifecycle annotations, or frequently reused documentation patterns.

We do not treat all exact docstring overlap as harmful leakage. In real code search scenarios, retrieval systems are expected to handle recur-

Leakage Type	Random	Repo	Change	Status
Repo contam.	99.96%	**0.00%**	Eliminated	Fixed
Exact doc. overlap	26.49%	12.39%	↓ 53%	Residual
Near-dup. code	7.48%	3.74%	↓ 50%	Residual
Exact func. code	0%	0%	None	Clean

Table 6: Multi-level leakage audit results comparing random and repository-level splits. "Repo contam." denotes test samples whose source repository also appears in training; "Near-dup." denotes near-duplicate code detected by MinHash-LSH.

ring documentation patterns across independent projects. Nevertheless, reporting this statistic is important for transparency and for interpreting benchmark results.

D.3 Near-Duplicate Code

Near-duplicate code is detected using MinHash-LSH. Such examples often correspond to common utility functions, including formatting, validation, encryption, or wrapper logic. Although the repository-level split substantially reduces near-duplicate code across partitions, it does not remove it entirely, since similar functions can be independently implemented in different projects.

This residual similarity reflects realistic cross-project code reuse and common programming patterns. It also provides a useful test of whether retrieval models can generalize to semantically similar but not identical implementations.

D.4 Exact Function-Body Duplication

Exact function-body duplication is the strongest form of code leakage. In ArkTS-CodeSearch, no identical function bodies are found across train, validation, and test partitions under either split. This confirms that the core retrieval target is not duplicated across partitions.

Overall, the repository-level split removes the primary contamination vector and reduces incidental overlap. We therefore release both random and repository-level splits to support different evaluation needs: the random split enables comparison with CodeSearchNet-style protocols, while the repository-level split provides a stricter cross-project robustness setting.

E Evaluation Metrics

We evaluate retrieval performance using Mean Reciprocal Rank (MRR), Normalized Discounted Cumulative Gain (NDCG), and Recall@k. Each docstring query is used to retrieve its corresponding ArkTS function from the candidate pool.

E.1 Mean Reciprocal Rank

MRR measures the rank position of the first relevant code snippet. It is defined as:

$$\mathrm{MRR} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{\mathrm{rank}_i},$$

where Q denotes the set of queries and rank_i is the rank of the correct code snippet for the i-th query. A higher MRR indicates that the correct function tends to appear earlier in the ranked list.

E.2 Normalized Discounted Cumulative Gain

NDCG evaluates the quality of the ranked retrieval list by assigning larger weights to relevant results that appear at higher positions. It is defined as:

$$\mathrm{NDCG@}k = \frac{\mathrm{DCG@}k}{\mathrm{IDCG@}k},$$

where

$$\mathrm{DCG@}k = \sum_{i=1}^{k} \frac{2^{rel_i} - 1}{\log_2(i+1)}.$$

Here, rel_i denotes the relevance score of the item at rank i, and $\mathrm{IDCG@}k$ is the ideal DCG value used for normalization.

E.3 Recall@k

Recall@k measures whether the correct function appears within the top k retrieved results:

$$\mathrm{Recall@}k = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \mathbf{I}(\mathrm{rank}_i \leq k),$$

where $\mathbf{I}(\cdot)$ is an indicator function. In our experiments, we report Recall@1 and Recall@5.

F Fine-Tuning Details

This section provides additional details on the fine-tuning setup used in our experiments.

F.1 Training Data Configurations

We consider three training configurations:

- **ARKTS-TRAINING**: the model is fine-tuned only on ArkTS function–docstring pairs from ArkTS-CodeSearch.
- **TS-TRAINING**: the model is fine-tuned only on TypeScript function–docstring pairs.
- **TS→ARKTS-TRAINING**: the model is first adapted on TypeScript pairs and then further fine-tuned on ArkTS pairs.

The motivation for using TypeScript data is that ArkTS is derived from TypeScript and shares related syntax, type-system features, and programming abstractions. TypeScript therefore provides a useful intermediate domain for adaptation before ArkTS-specific fine-tuning.

F.2 Training Objective

We fine-tune embedding models using Multiple Negatives Ranking Loss. Given a query embedding q, its positive code embedding c^+, and a set of in-batch negative code embeddings $\{c_i^-\}_{i=1}^N$, the loss is defined as:

$$\mathcal{L} = -\log \frac{\exp(\mathrm{sim}(q, c^+))}{\exp(\mathrm{sim}(q, c^+)) + \sum_{i=1}^{N} \exp(\mathrm{sim}(q, c_i^-))}.$$

Here, $\mathrm{sim}(\cdot)$ denotes cosine similarity. This objective encourages semantically aligned docstring–function pairs to be close in the embedding space while pushing mismatched pairs apart.

F.3 Optimization Hyperparameters

All models are trained using the same general optimization setup to ensure fair comparison across architectures.

In our experiments, models are trained for two epochs with a batch size of 32 and a learning rate of 2×10^{-5}. We use the Adam optimizer with a linear warm-up schedule over 10% of the total training steps. The maximum input length is set to 512 tokens. Training samples are shuffled at each epoch.

F.4 Computational Budget and Infrastructure

All fine-tuning and retrieval experiments were conducted on NVIDIA A100 GPUs. Unless otherwise specified, each training run used one A100 GPU with 80GB memory. Fine-tuning each model on ARKTS-TRAINING required approximately 10 GPU-hours, training on TS-TRAINING required approximately 40 GPU-hours, and the two-stage TS→ARKTS-TRAINING setting required approximately 9 GPU-hours. Across all models and training configurations, the total computational budget was approximately 50 A100 GPU-hours. Retrieval evaluation was performed on the same infrastructure and required approximately 0.1 GPU-hours in total.