Vol. 3, No. 3, pp.: 1-4, 2025

Neural Code Search: Evaluating Embedding Strategies for Repository-Level Code Retrieval

Zainab Al-Rashidi[1,*], **Pavel Novotný**[2], **Min-Ji Kim**[3]

[1] Department of Computing, Imperial College London, London, UK, SW7 2AZ
[2] Faculty of Informatics, Masaryk University, Brno, Czech Republic, 60200
[3] School of Computer Science, KAIST, Daejeon, South Korea, 34141
[*] *z.alrashidi@imperial.ac.uk*

Article Information	
Received	2 June 2025
Accepted	30 September 2025
DOI	https://doi.org/10.63646/datamind.2025.030301

Abstract

Code search — retrieving semantically relevant code given a natural language query — is a foundational capability for developer tooling, code review assistance, and automated programming. Recent work has advanced single-function code search substantially, but repository-level retrieval — finding relevant code across entire codebases that may contain millions of tokens — presents distinct challenges that single-function benchmarks do not capture: cross-file dependencies, project-specific idioms, and the need to retrieve code fragments at varying granularities. This paper evaluates six embedding strategies for repository-level code retrieval: TF-IDF with BM25 (lexical baseline), GraphCodeBERT (structural), CodeT5+ (generative), UniXcoder (multi-modal), Voyage Code 3 (proprietary dense), and a late-interaction architecture adapted from ColBERT (ColCode). We construct a new evaluation benchmark (RepoSearch-1K) consisting of 1,000 repository-search queries across five programming languages and eight domains, with relevance annotations from professional software engineers. Results show that late-interaction approaches substantially outperform single-vector dense retrieval on cross-file dependency queries, and that structural embeddings (GraphCodeBERT) retain an advantage over purely semantic approaches on queries involving abstract syntax tree relationships. We release RepoSearch-1K as a community resource.

Keywords: ***code search; code embeddings; neural code retrieval; repository-level; CodeBERT; ColBERT; software engineering; NLP4Code***

1. Introduction

Ask any experienced software engineer how they navigate an unfamiliar codebase and they will describe a process that involves searching for conceptual anchors — functions whose names suggest they handle a relevant operation, comments that mention a familiar concept, class names that align with the task at hand. This is a naturalised form of code search, and it is remarkably effective when performed by humans with relevant domain knowledge. The question of how to mechanise it — how to build a system that, given a natural language description of what you need, surfaces the most relevant code in a large repository — has attracted substantial research attention.

The advances in single-function code search, driven largely by the CodeSearchNet benchmark and the models it inspired, have been genuine. Models like CodeBERT and its successors achieve

impressive results at retrieving the correct function from a set of candidates when the task is well-defined and the granularity is fixed. But repository-level retrieval is a harder problem. Real developer queries are rarely about single, self-contained functions. They involve finding the right combination of functions and data structures across multiple files, understanding how module interfaces fit together, and navigating project-specific conventions that differ from any training distribution.

1.1 Contributions

We contribute: (1) RepoSearch-1K, a new benchmark for repository-level code retrieval with 1,000 queries across five languages (Python, Java, TypeScript, Go, Rust) and eight domains; (2) a comparative evaluation of six embedding strategies on this benchmark; (3) an analysis of query type effects on relative retrieval performance.

2. Methods

2.1 RepoSearch-1K Benchmark

RepoSearch-1K was constructed from 50 open-source repositories (10 per language) selected for diversity of domain, size (10K–500K lines), and activity level. For each repository, 20 queries were written by software engineers fluent in the relevant language, drawn from the categories: API usage queries (how do I do X using this codebase?), implementation queries (find where behaviour Y is implemented), dependency queries (what code depends on component Z?), and cross-file pattern queries (find all places where pattern P is used). Relevance was annotated at the file-function granularity by two annotators each with 5+ years of experience in the relevant language, with Krippendorff's $\alpha = 0.81$.

2.2 Embedding Approaches

BM25 serves as the lexical baseline. GraphCodeBERT encodes both token sequences and data-flow graph structure. CodeT5+ is evaluated in its 2B-parameter variant with code-to-text and text-to-code pre-training. UniXcoder adds comment and AST node embeddings to the token representation. Voyage Code 3 is a proprietary model evaluated via API. ColCode is our late-interaction adaptation of ColBERT that maintains token-level embeddings for code and scores queries using maximum similarity over token representations rather than a single vector comparison.

Table 1. *Retrieval performance on RepoSearch-1K by query type. MRR@10 reported. Best result per row bold.*

Query Type	BM25	GraphCodeBERT	CodeT5+	UniXcoder	VoyageCode3	ColCode
API usage	0.42	0.58	0.61	0.63	0.67	0.69
Implementation	0.51	0.64	0.66	0.65	0.71	0.73
Dependency queries	0.31	0.47	0.44	0.49	0.52	0.61
Cross-file patterns	0.29	0.52	0.48	0.51	0.55	0.67
Average	0.38	0.55	0.55	0.57	0.61	0.68

MRR@10 = Mean Reciprocal Rank at 10. Scores reflect average over all five programming languages. Language-specific breakdowns available in the supplementary material. BM25 advantage on lexically precise queries is preserved but smaller in magnitude than expected.

3. Results

3.1 Late Interaction Advantage

ColCode's advantage over single-vector approaches is most pronounced on dependency and cross-file pattern queries (Table 1), where the margin over the second-best system (VoyageCode3) is 9 and 12 points in MRR@10 respectively. The mechanism is interpretable: single-vector embeddings compress an entire code fragment into a fixed-dimension vector, which necessarily loses the specific token-level signals that distinguish, say, a function that calls a particular API method from one that merely imports the same module. Late interaction preserves token-level granularity and allows the scoring function to match on the precise tokens that are most relevant to the query, rather than relying on the compressed global semantics captured by a single vector.

3.2 Structural Embedding Value

GraphCodeBERT shows a disproportionate advantage on cross-file pattern queries relative to its overall performance. We attribute this to its explicit data-flow graph encoding, which makes structurally similar code patterns more retrievable regardless of identifier naming conventions — a critical capability in cross-project retrieval where naming conventions differ. This advantage diminishes on API usage queries, where the specific identifier names matter more than the abstract structural pattern.

[Figure 1 — Performance breakdown by programming language for ColCode and VoyageCode3 (top two performers). ColCode's advantage is largest for Rust (MRR@10 gap of 0.12), where cross-file dependencies are particularly dense due to Rust's ownership model. The advantage is smallest for Python, where BM25 lexical matching remains relatively competitive due to descriptive identifier naming conventions.]

Figure 1. *Performance breakdown by programming language for ColCode and VoyageCode3 (top two performers). ColCode's advantage is largest for Rust (MRR@10 gap of 0.12), where cross-file dependencies are particularly dense due to Rust's ownership model. The advantage is smallest for Python, where BM25 lexical matching remains relatively competitive due to descriptive identifier naming conventions.*

4. Discussion

The headline finding — that late-interaction retrieval substantially outperforms single-vector approaches for repository-level code search — has a practical implication for developer tooling that we think is underappreciated. Most production code search systems, including many AI code assistant products, use single-vector dense retrieval for its indexing simplicity and serving speed. Our results suggest that this architectural choice comes with a meaningful accuracy cost specifically on the query types that developers find most difficult (dependency navigation, cross-file pattern search), i.e., precisely the queries for which automated assistance would be most valuable.

5. Conclusion

Repository-level code search requires retrieval architectures that preserve token-level granularity. ColBERT-style late interaction provides meaningful improvements over single-vector approaches on cross-file and dependency queries, and should be considered the preferred retrieval architecture for developer tooling applications where these query types are important. RepoSearch-1K is available at https://github.com/datamind-papers/reposearch.

References

Husain, H., Wu, H. H., Gazit, T., Allamanis, M., & Brockschmidt, M. (2019). CodeSearchNet challenge:

Evaluating the state of semantic code search. arXiv preprint arXiv:1909.09436. https://doi.org/10.48550/arXiv.1909.09436

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., & Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. Findings of EMNLP 2020, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Shujie, L., & Zhou, M. (2021). GraphCodeBERT: Pre-training code representations with data flow. ICLR 2021. https://openreview.net/forum?id=jLoC4ez43PZ

Wang, Y., Le, H., Gotmare, A. D., Bui, N. D. Q., Li, J., & Hoi, S. C. H. (2023). CodeT5+: Open code large language models for code understanding and generation. EMNLP 2023. https://doi.org/10.18653/v1/2023.emnlp-main.68

Khattab, O., & Zaharia, M. (2020). ColBERT: Efficient and effective passage search via contextualized late interaction over BERT. SIGIR 2020, 39–48. https://doi.org/10.1145/3397271.3401075

Robertson, S., & Zaragoza, H. (2009). The probabilistic relevance framework: BM25 and beyond. Foundations and Trends in Information Retrieval, 3(4), 333–389. https://doi.org/10.1561/1500000019

Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., & Liu, S. (2021). CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. NeurIPS 2021 Datasets Track. https://proceedings.neurips.cc/paper/2021/hash/c16a5320fa475530d9583c34fd356ef5-Abstract-Datasets_and_Benchmarks.html

Li, Y., Wang, S., Nguyen, T. N., & Van Nguyen, S. (2021). Editsum: A retrieve-and-edit framework for source code summarization. ASE 2021, 292–303. https://doi.org/10.1109/ASE51524.2021.9678882

Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A survey of machine learning for big code and naturalness. ACM Computing Surveys, 51(4), 81. https://doi.org/10.1145/3212695

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., & Zaremba, W. (2021). Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374. https://doi.org/10.48550/arXiv.2107.03374