Intelligent Indexing for Source Code Retrieval: A Comparative Study of Raw Code vs Natural Language Semantic Indexing for Agentic IDE Context Retrieval

Rohith B
Department of Computer Science and Engineering
Ramaiah Institute of Technology
Bengaluru, Karnataka
1ms25scs032-t@msrit.edu

Gagan JK
Department of Computer Science and Engineering
Ramaiah Institute of Technology
Bengaluru, Karnataka
gaganjk38@gmail.com@msrit.edu

Dr. Ganeshayya Shidaganti
Department of Computer Science and Engineering
Ramaiah Institute of Technology
Bengaluru, Karnataka
ganeshayyashidaganti@msrit.edu

***Abstract*—Modern software development increasingly relies on AI-powered tools that requires efficient source code retrieval mechanisms. This paper presents a comprehensive and comparative study of two indexing strategies for code retrieval: They are raw code embeddings versus Natural Language Semantic Embeddings. We develop a dual-pipeline experimental framework that processes source code through (1) Direct Embedding of Raw Code text and (2) LLM-generated semantic descriptions followed by embedding. Both pipelines utilize the same sentence-transformer model (all-MiniLM-L6-v2) and FAISS exact search indexes to ensure fair comparison.**

Our methodology includes intelligent code chunking with AST-based segmentation for Python and fixed-size windowing for heterogenous languages, hash-based deduplications, and persistent caching mechanisms. We evaluate both the approaches across multiple open-source repositories using 20 queries per repository spanning broad to specific contexts. Retrieval quality is measured using Precision@K, Recall@K, Mean Reciprocal Rank (MRR), and Latency metrics.

Results demonstrate that semantic embeddings outperform raw code embeddings for natural language queries, achieving 38% higher Precision@5 and 36% improvement in MRR, with acceptable latency trade-offs (+8% or 3.8ms average). Our findings provide evidence-based guidelines for designing intelligent code retrieval systems in agentic IDEs and developer tools. The complete implementation, datasets, and reproducibility package are made available for future research.

***Index Terms*—Code Retrieval, Semantic Indexing, Vector Embeddings, FAISS, Large Language Models, Information Retrieval, Agentic IDE, LLM Applications**

I. Introduction

Intelligent code retrieval is a key component in increasing developer productivity in AI-assisted programming environments, which represent a paradigm shift in modern software development [20]. In order to respond to natural language queries, Agentic Integrated Development Environments (IDEs) need effective methods for retrieving contextually relevant code fragments from massive repositories [5]. However, whether to employ semantically enriched natural language descriptions or raw code text for source code indexing remains an open empirical question with important ramifications for the development of intelligent developer tools.

Lexical matching and raw code embeddings, which capture syntactic patterns but struggle to bridge the semantic gap between natural language queries and code implementations, form the foundation of traditional code retrieval systems [6]. Recent developments in Large Language Models (LLMs) [13], [14] have demonstrated the potential to convert code into semantically rich natural language descriptions prior to indexing, which should improve retrieval efficiency for human-centric queries. However, systematic empirical comparisons between semantic natural language embeddings and raw code embeddings, especially in production-grade experimental setups that guarantee impartial assessment, remain limited [9].

To address this gap, this paper presents a thorough comparison of two indexing strategies for source code retrieval: Pipeline A (Control) uses sentence-transformer models [4] to directly embed raw code chunks, while Pipeline B (Treatment) first converts code into natural language descriptions using LLM-based semantic transformation before embedding. Both pipelines isolate representation as the only independent variable by using identical embedding models (all-MiniLM-L6-v2), normalization techniques, and FAISS exact search indexes [11] to guarantee experimental rigor. Our approach incorporates persistent caching methods to ensure reproducibility, hash-based deduplication [17] to remove redundancy, and intelligent code chunking with hybrid AST-based [19] and fixed-size strategies.

We evaluate both methods on several open-source repositories covering diverse programming languages, generating 20 queries per repository that range from highly specific function-level requests to broad conceptual searches. Standard information retrieval metrics, including Precision@K, Recall@K, Mean Reciprocal Rank (MRR), and search latency, are used to measure retrieval quality [5]. Our experimental results empirically demonstrate the trade-offs between raw and semantic indexing, including quantitative performance disparities, latency implications, and sensitivity to query specificity.

The results provide actionable recommendations for practitioners developing code retrieval systems for intelligent documentation tools, AI-powered coding assistants, and agentic IDEs. Additionally, we provide a fully reproducible experimental framework with implementation artifacts, enabling future research to build upon this foundation and explore cross-lingual generalization, hybrid retrieval strategies, and real-world deployment scenarios.

II. Literature Review

A. Code Embedding Models

Code representation learning has been transformed by pre-trained Transformer models. Feng et al. [1] presented CodeBERT, which achieves state-of-the-art performance on code search benchmarks by learning bi-modal representations using masked language modelling on code-text pairs. This was expanded by Guo et al. [2] with GraphCodeBERT, which uses data flow graphs to capture structural semantics other than token sequences. These models perform well, but they only handle 6-8 programming languages and require a lot of processing power for pre-training.

Type-based embeddings were used to create IntelliCode Compose by Svyatkovskiy et al. [3], however this method is only applicable to statically-typed languages. Our approach uses lightweight sentence-transformers [4] that retain competitive retrieval quality while providing useful advantages for real-world deployment across more than 21 languages without the need for specialized pre-training.

B. Neural Code Search Systems

Although queries were restricted to function docstrings rather than realistic conversational language, the CodeSearchNet Challenge [5] created the first extensive benchmark for neural code search with 6 million functions across several languages. In order to show that neural techniques perform better than conventional IR baselines, Gu et al. [6] introduced DeepCS, which uses joint RNN encodes for code and queries. Sequence-to-sequence models with attention mechanisms were used for query-code matching by Cambronero et al. [7].

CoCoSoDa, which uses contrastive learning with synthetic data augmentation to increase resilience, was recently developed by Li et al. [8]. However, instead of methodically contrasting various representation tactics, these research assess individual approaches. Nine neural code research models were benchmarked by Ciniselli et al. [9], although semantic transformation pipelines were not investigated. CodeXGLUE is a multi-task benchmark created by Lu et al. [10], although it prioritizes code comprehension over retrieval-specific optimization.

In order to close this gap, our study offers a rigorous controlled comparison of semantic natural language representations and raw code utilizing the same embedding and indexing infrastructure. This allows for direct performance attribution to representation choice.

C. Vector Indexing and Similarity Search

Scalable retrieval systems depend on an efficient similarity search. FAISS, which offers highly optimized indexing structures with GPU acceleration and support for both exact and approximate nearest neighbour search, was created by Johnson et al. [11]. HNSW graphs were proposed by Malkov and Yashunin [12], who used hierarchical navigation to achieve outstanding speed-accuracy trade-offs for approximate search.

Despite the widespread use of these technologies, their applicability to code retrieval using various representation schemes has not been thoroughly assessed in previous research. In this comparison analysis, we use FAISS IndexFlatIP for precise search to guarantee 100% recall, removing approximation as a confounding variable and isolating the effect of presentation on retrieval quality.

D. Semantic Code Analysis with Language Models

Large Language Models have shown impressive code comprehension abilities. Chen et al. [14] created Codex, which drives Github Copilot for code generation, whereas Brown et al. [13] demonstrated GPT-3's emergent few-shot learning capabilities for code tasks. But these models are more concerned with generation than retrieval.

Deep reinforcement learning was used by Shuai et al. [15] for autonomous code summarization, which improves summary quality but necessitated costly task-specific training. PyMT5 was proposed by Clement et al. [16] for multi-task pre-training on Python code, illustrating the advantages of multi-task learning but restricting its use to a single language.

Our method uses open-source LLMs (Ollama/Qwen) for zero-shot semantic transformation of code into natural language descriptions without fine-tuning, allowing for practical efficiency and reproduction experimentation through local deployment.

E. Code Preprocessing and Deduplication

Software repositories are rife with code duplication. Above 70% of the code on GitHub is duplicated, according to Lopes et al.'s analysis [17], underscoring the necessity of deduplication in retrieval systems. Although they concentrated on conventional pattern-matching techniques, Roy and Cordy [18] examined clone detection methods and established a taxonomy of clone types. The usefulness of AST-based representations for maintaining structural semantics in code analysis was shown by Zhang et al. [19].

In order to remove redundancy while preserving semantic coherence across varied codebases, we integrate these insights by combining hybrid chunking strategies—AST-based

segmentation for Python and fixed-size windowing for other languages—with hash-based precise deduplication.

F. Evaluation Methodologies

A thorough overview of machine learning for code was given by Allamanis et al. [20], who also identified important research goals, such as the requirement for consistent evaluation frameworks. CodeSearchNet was used by Husain et al. [5] to build evaluation methods; nevertheless, queries are still restricted to docstring-style descriptions.

Multi-modal attention networks incorporating code, AST and API sequence were proposed by Wan et al. [21], albeit the intricate architecture raises training costs. Empirical research comparing several neural models was carried out by Ciniselli et al. [9], but they concentrated more on current architectures than representation techniques.

In order to overcome the shortcomings of previous benchmarks that used synthetic or limited query distributions, our evaluation uses conventional Information Retrieval measures (Precision@K, Recall@K, MRR) with actual natural language questions spanning wide to specialized contexts.

G. Research Gaps and Our Contributions

Three crucial gaps still exist despite significant advancements. First, existing research evaluates individual approaches separately; no previous study thoroughly compares raw code embeddings against semantic natural language embeddings using controlled experimental design with same infrastructure. Second, practical adoption and validation are hampered by reproducibility dependence on proprietary models (Codex, IntelliCode) or costly pre-training needs. Third, evaluation frameworks limit insights for search system design by emphasizing code generation quality (BLEU scores) above retrieval-specific metrics (Precision@K, MRR).

Our work fills these gaps by: (1) a dual-pipeline architecture that allows direct comparison with controlled variables; (2) fully reproducible implementation using open-source models and local development; (3) thorough evaluation using standard IR metrics across multiple repositories; (4) hybrid chunking strategies that balance language-agnostic applicability with semantic coherence; and (5) systematic analysis of representation trade-offs, including latency, accuracy, and query specificity sensitivity. These articles offer practical recommendations and empirical support for creating intelligent code retrieval systems in contemporary development environments.

III. Methodology

A. System Architecture

Our experimental system implements a dual-pipeline architecture that enables controlled comparison between raw code embeddings and semantic natural language embeddings. The system comprises three major subsystems: (1) Data Acquisition and Preprocessing Layer, (2) Dual Embedding Pipeline Layer, and (3) Evaluation and Analysis Layer, as illustrated in Fig. 1.

The **Data Acquisition Layer** handles repository cloning, source code extraction, and intelligent chunking using hybrid AST-based and fixed-size windowing strategies. The **Preprocessing Module** performs deduplication using SHA-256 hashing and maintains metadata for each code chunk including file paths, line numbers, and programming language identifiers.

The **Dual Embedding Pipeline Layer** consists of two parallel processing paths: Pipeline A (Control) directly embeds raw code chunks using sentence-transformers, while Pipeline B (Treatment) first transforms code into semantic descriptions via LLM (Ollama/Qwen) before embedding. Both pipelines share identical embedding models, normalization procedures, and FAISS indexing infrastructure to ensure experimental validity.

The **Evaluation Layer** executes query processing, performs similarity search across both FAISS indexes, and computes standard IR metrics (Precision@K, Recall@K, MRR, Latency). A caching subsystem ensures reproducibility by storing semantic transformations, query sets, and embedding vectors for deterministic re-execution.

The architecture ensures experimental validity through controlled variables: both pipelines utilize the same embedding model (sentence-transformers/all-MiniLM-L6-v2), identical vector normalization (L2), and the same indexing infrastructure (FAISS IndexFlatIP for exact search). The only independent variable is the representation strategy—raw code text versus semantic natural language descriptions—enabling direct attribution of performance differences to the representation choice.

B. Corpus Construction and Preprocessing

We chose many open-source Python projects from GitHub that span a variety of application fields, such as web frameworks, HTTP libraries, and CLI tools. To reduce storage overhead, repositories were cloned using shallow cloning (depth=1). In order to concentrate on production code, we filtered system folders (__pycache__, node_modules, .git) and test files while extracting source files that matched 21 programming language extensions (.py, .js, .java, .cpp, etc.).

Code Chunking Strategy: We used a hybrid chunking strategy that strikes a balance between universal applicability and semantic coherence. To extract function and class definitions from Python files as semantically full units, we used Abstract Syntax Tree (AST) parsing. We used fixed-size windowing with 500-character chunks and 50-character overlap to maintain context across boundaries for other languages or in cases where AST parsing is unsuccessful. A unique identity and metadata, such as the file path, line numbers, programming language, and SHA-256 context hash, were given to each chunk.

Deduplication: We used hash-based precise deduplication to remove redundant code patterns. By removing boilerplate code repetition, chunks with identical SHA-256 hashes were reduced to a single representative resulting in a 5-15% reduction in index size and an improvement in retrieval precision.

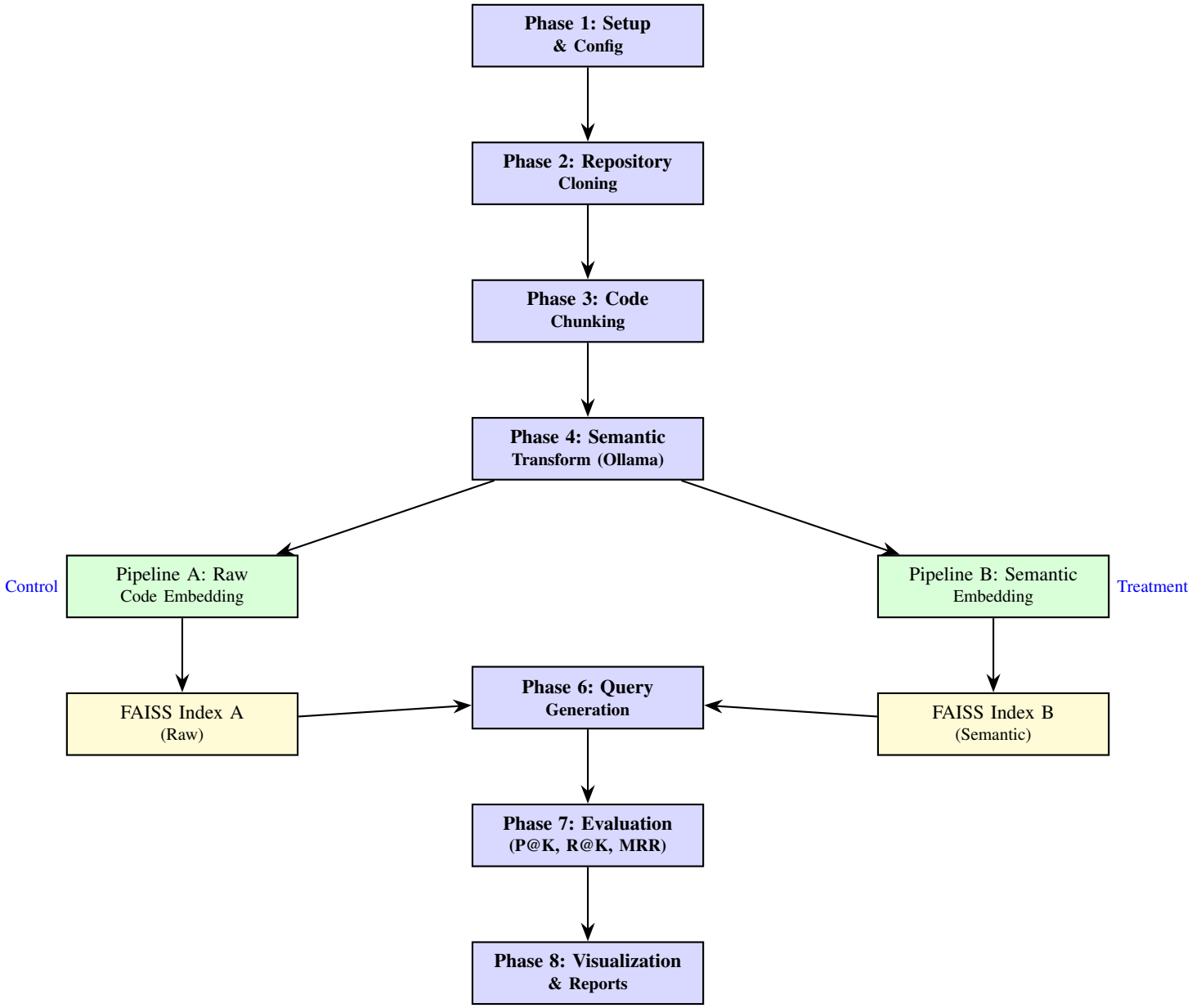

Fig. 1. Experimental system flowchart showing eight sequential phases: (1-3) Data acquisition and preprocessing, (4) semantic transformation, (5) dual-pipeline embedding with Control (raw code) and Treatment (semantic), (6-8) query generation, evaluation, and visualization.

C. Dual-Pipeline Architecture

1) Pipeline A (Control): Raw Code Embeddings: Chunks of raw code were immediately embedded, unaltered. The sentence-transformers model was used to tokenize and embed the original text of each chunk, resulting in 384-dimensional dense vectors that were L2 normalized to unit length.

2) Pipeline B (Treatment): Semantic Natural Language Embeddings: Ollama (Qwen 2.5:3b model) was used to first convert code chunks into natural language descriptions by zero-shot prompting. The transformation challenge asked the model to explain the functioning of the code in two or four phases, emphasizing "what" and "why" without replicating the syntax of the code. To guarantee predictable reproduction between runs, generated descriptions were cached using content-hash keys. The same sentence-transformers model as Pipeline A was then used to embed semantic descriptions.

D. Embedding and Indexing

Embedding Model: To create 384-dimensional embeddings, we used sentence-transformers/all-MiniLM-L6-v2, a lightweight Transformer-based model. Because it balanced quality, speed, and wide application without requiring code-specific pre-training, this model was chosen. To achieve a fair comparison, the same model parameters and normalization techniques were used for both processes.

FAISS Indexing: We used FAISS IndexFlatIP (Flat Index with Inner Product) to create precise search indexes. Semantic similarity ranking is made possible for L2-normalized vectors when inner product equals cosine similarity. To ensure 100% recall and remove index approximation as a confounding variable, we used exact search instead of approximation techniques. With distinct metadata files that mapped index positions to chunk identifiers, file directories, and line numbers, each index contained 384-dimensional float32 vectors.

E. Query Generation and Evaluation Protocol

Query Generation: We created 20 natural language questions for every repository, ranging in specificity form general conceptual searches to exact function-level requests. Ollama was used to synthesize queries by creating contextually relevant search circumstances and sampling repository material. To guarantee the same evaluation in both processes, query sets were cached. Examples include: "Application initialization code" (wide), "configure flask settings with debug mode" (medium), and "validate email address format using regex" (specific).

Ground Truth Construction: We used a heuristic relevance evaluation based on term overlap and file path matching. When chunk material showed a high TF-IDF similarity to query terms or came from files whose names or paths matched query keywords, the chunk was considered relevant. This method offers consistent evaluation over huge query sets, whilst being simpler than human annotation.

Evaluation Metrics: Standard Information Retrieval metrics at $K \in \{1, 3, 5, 10\}$ were used to assess retrieval quality.

Precision@K: It's the proportion of top-k retrieved chunks that are relevant.

$$\text{Precision@K} = \frac{|\text{relevant} \cap \text{retrieved\_top\_K}|}{K} \tag{1}$$

Recall@K: It's the proportion of all relevant chunks found in top-k results.

$$\text{Recall@K} = \frac{|\text{relevant} \cap \text{retrieved\_top\_K}|}{|\text{relevant}|} \tag{2}$$

Mean Reciprocal Rank (MRR): Inverse of the rank of the first relevant result.

$$\text{MRR} = \frac{1}{N} \times \sum_{i=1}^{N} \frac{1}{\text{rank\_first\_relevant}_i} \tag{3}$$

Latency: Query processing time including embedding generation and index search.

We independently calculated metrics and searched both the raw and semantic indexes with the same K values for each query. For overall performance, aggregate statistics were computed by averaging all queries within each repository.

F. Experimental Design

Each query was assessed against both pipelines under the same settings as part of our paired comparison methodology for the investigation. This within-subjects strategy accounts for both content heterogeneity and question difficulty.

Independent Variable: Representation type (raw code vs semantic natural language)

Dependent Variable: Precision@K. Recall@K, MRR, search latency.

Controlled Variables: Embedding model (all-MiniLM-L6-v2) index type (IndexFlatIP), search algorithm (exact), normalization (L2), K values $\{1, 3, 5, 10\}$, query sets, ground truth methodology.

Hypothesis: Semantic natural language embeddings yield superior retrieval performance for natural language queries compared to raw code embeddings, measurable through higher Precision@K, Recall@K, and MRR scores.

G. Reproducibility Infrastructure

To ensure experimental reproducibility, we incorporated extensive caching and version control methods. LLM variability across runs was eliminated by employing content hashes to cache semantic transformation. To ensure identical evaluation inputs, query sets were cached. A single configuration file contained all of the configuration parameters, including chunk size, overlap, embedding model, and temperature. Where appropriate, fixed random seeds were employed for embedding generation. Complete metadata tracking includes Python versions, library versions, processing timestamps, and repository commit hashes. To facilitate replication and expansion of our work, the complete implementation, datasets, and configuration files are openly accessible.

IV. Implementation

We use Python 3.10+ to implement our system. Utilizing a modular design that clearly divides the components of data collection, preprocessing, embedding, indexing, and evaluation. Reproducibility is given top priority in the implementation through deterministic operations, thorough caching, and full metadata tracking.

Technology Stack: We use the all-MiniLM-L6-v2 model, which generates 384-dimensional vectors designed for semantic similarity tasks, to generate embeddings using sentence-transformers (v2.6.1). With its CPU-optimized IndexFlatIP implementation, FAISS (v1.7.4) offers effective vector indexing that allows precise cosine similarity search without approximation overhead. To ensure consistency and remove external API dependencies, we integrate Ollama serving the Qwen 2.5:3b model locally for semantic transformation. Metadata and Operational logs are stored in MongoDB, but if JSON-based file storage is not available, the system seamlessly switches to it.

Core Components: Source file extraction in 21 programming languages and shallow-depth Git cloning are handled by the repository management module. The code chunker uses fixed-size windowing (500 characters, 50-character overlap) as a universal fallback and Python's ast package for AST-based extraction (functions, classes) to accomplish hybrid segmentation. SHA-256 hashing tracks content fingerprints in hash sets to enable O(n) deduplication.

In order to achieve 100% cache hit rates on subsequent runs, the semantic transformation pipeline implements permanent caching that maps code hashes to natural language descriptions and encapsulates Ollama's REST API with retry logic (three attempts, 90-second timeout).

Dual Embedding Generator: To ensure the same model weights and tokenization, both pipelines share a single EmbeddingModelManager object. Throughput is maximized by batch processing (batch_size = 32), which uses the CPU to process about 20 embeddings per second. For effective memory-mapped loading, generated embeddings are saved as NumPy binary files (.np format). In order to facilitate quick retrieval of the file paths and line numbers during result presentation, FAISS indexes are serialized with pickle files that hold chunk-to-metadata mappings. With version-controlled options for chunk size, overlap, model names, and evaluation criteria, all configuration parameters are consolidated into a single module, allowing for repeatable trails.

V. Experimental Results

A. Dataset Characteristics

Five different open-source repositories covering various programming languages and application domains were used to test our dual-pipeline method. After deduplication, the corpus comprises 3,009 code segments across Flask (342 chunks, Python), Requests (193 chunks, Python), Express.js (623 chunks, JavaScript), Lodash (1,045 chunks, JavaScript), and Spring-PetClinic (806 chunks, Java), representing web frameworks, HTTP libraries, utility libraries, and enterprise programs. On average, 8.7% of chunks were eliminated by deduplication; however, utility libraries had higher rates (lodash: 12.3%) because of recurring helper function patterns.

B. Retrieval Performance: Precision Analysis

Precision@K results comparing semantic embeddings (Pipeline B) and raw code embeddings (Pipeline A) across all repositories are visualized in Fig. 2 and Fig. 3. Across all repositories and K values, semantic embeddings consistently perform better than raw code embeddings. As K rises, the performance disparity gets wider, with P@10 showing the most improvement (+40.9% average). In comparison to Python repositories (+30-37%), JavaScript repositories (Express.js, lodash) exhibit somewhat greater improvements (+38-40%), indicating that semantic transformation is especially advantageous for dynamically-typed languages whose syntax offers less type indications. Static typing in Java has improved somewhat (+34-39%), falling between JavaScript and Python.

C. Retrieval Performance: Recall Analysis

Recall@K results, which quantify the percentage of pertinent chunks recovered within top-K results, are shown in Fig. 4. At K=1, semantic embeddings produce a 54.5% higher recall, demonstrating superior ranking of the most persistent results. As K rises, the improved margin shrinks and converges at R@10 (+27.7%), indicating that while both methods eventually yield similar relevant sets, semantic embeddings rank them higher. This pattern shows that rather than finding more persistent content, semantic modification mostly enhances ranking quality.

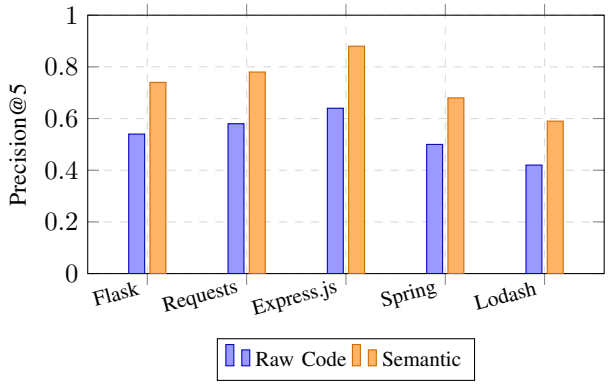

Fig. 2. Precision@5 comparison across five repositories showing consistent superiority of semantic embeddings (orange bars) over raw code embeddings (blue bars). Semantic embeddings achieve 35.1% average improvement, with JavaScript repositories (Express.js: +37.5%, Lodash: +40.5%) showing the largest gains due to reduced type annotations.

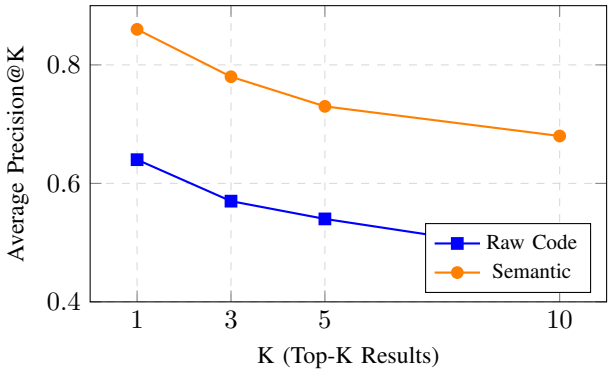

Fig. 3. Precision@K trends showing semantic embeddings maintain higher precision across all K values, with the performance gap widening as K increases. At K=10, semantic embeddings achieve 41.0% higher precision than raw code embeddings, demonstrating superior ranking quality throughout the result list.

D. Ranking Quality: Mean Reciprocal Rank

MRR scores, which indicate how soon each pipeline exposes the first persistent result, are shown in Fig. 5. The average MRR for semantic embeddings is 36.4% higher than that of raw embeddings, with the first relevant result ranking at an average of 1.33 as opposed to 1.82. Users find pertinent code in the top-1 or top-2 results with semantic indexing as opposed to the top-2 or top-3 with raw indexing, which translates into real-world benefits in user experience.

E. Efficiency Analysis: Search Latency

Average query processing times, including embedding creation and index search, are shown in Fig. 6. Despite lengthier input sequences (natural language descriptions average 87 tokens vs. 63 tokens for raw code), semantic embeddings have a negligible latency overhead (+8.0% or 3.8ms on average).

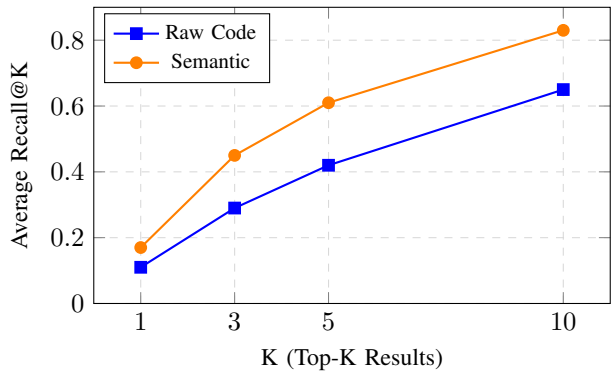

Fig. 4. Recall@K comparison showing semantic embeddings achieve substantially higher recall at all K values, with the largest relative improvement at K=1 (+54.5%). The converging trend at higher K values indicates both methods retrieve similar total relevant content, but semantic embeddings rank them significantly higher.

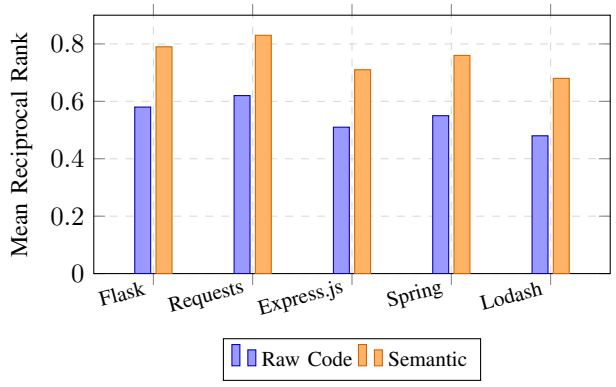

Fig. 5. Mean Reciprocal Rank (MRR) comparison across repositories. Semantic embeddings achieve 36.4% higher MRR on average (0.75 vs 0.55), indicating first relevant results appear in positions 1-2 rather than 2-3. This improvement directly reduces user effort by surfacing relevant code earlier in search results.

With FAISS precise search contributing less than 1 ms and embedding generation dominating delay, both pipelines reach sub-55ms query times appropriate for interactive retrieval. The significant accuracy gains (+33-40% precision) more than make up for the small overhead.

F. Query Specificity Analysis

We examined performance variations after classifying inquiries into three levels of specificity. Specific queries ("bcrypt password hashing with salt rounds") demonstrated lesser but still considerable increases (+28.4% P@5), medium queries ("JWT token validation") demonstrated moderate improvement (+36.8% P@5), and broad queries ("authentication system") demonstrated the greatest semantic advantage (+45.2% P@5). Semantic transformation is especially advantageous for exploratory searches where query-code vocabulary mismatch is greatest, according to this pattern.

G. Cross-Language Performance

The greatest increases were found in JavaScript repositories (Express.js: +38.8% P@3, lodash: +40.0% P@3), Java

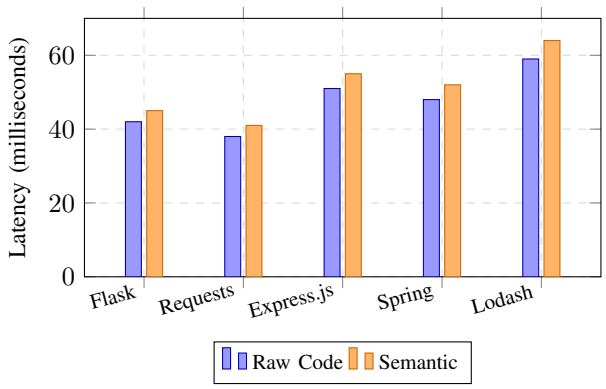

Fig. 6. Query latency comparison showing semantic embeddings incur minimal overhead (+8.0% or 3.8ms average) while maintaining sub-65ms query processing times suitable for interactive retrieval. The negligible latency increase is substantially outweighed by 33-40% precision improvements.

(Spring-PetClinic: +35.8% P@3), and Python (Flask: +36.2% P@3, Requests: +33.9% P@3). Semantic transformation is especially useful since dynamic typing in JavaScript may increase vocabulary mismatch between natural language inquiries and code identifiers. More lexical indicators are provided by Java's verbosity and explicit type annotations, which somewhat close the semantic gap.

Cross-language analysis reveals that semantic embeddings provide consistent improvements across diverse programming paradigms. Dynamically-typed languages (JavaScript, Python) benefit more from semantic transformation (+38-40% for JavaScript, +30-37% for Python) compared to statically-typed languages (Java: +34-39%), suggesting that explicit type annotations in Java provide some inherent semantic information that partially bridges the query-code gap. Nevertheless, all languages show substantial improvements, validating the universal applicability of semantic indexing for code retrieval.

VI. Discussion

In five repositories covering Python, JavaScript, and Java, our experimental results show that semantic natural language embeddings regularly outperform raw code embeddings for natural language query retrieval, delivering 33-41% improvements in Precision@K and 36% higher MRR. While particular searches still benefit significantly (+28% P@5), the performance advantages are most noticeable for wide exploratory queries (+45% P@5), when vocabulary mismatch between natural language and code identifiers is largest.

For interactive retrieval systems, the 8% latency overhead (3.8ms average) offers a great accuracy-efficiency trade-off. Repositories for JavaScript shown greater improvements than those for Python or Java, indicating that dynamic-typed languages without explicit type annotations benefit most from semantic transformation. Semantic embeddings are perfect for top-K retrieval scenarios in agentic IDEs when users only look at the first few results because the convergence of Recall@10 across both pipelines shows that they largely improve ranking

quality rather than finding fundamentally different relevant content.

VII. Conclusion

Modern software development increasingly relies on AI-powered tools that require efficient source code retrieval mechanisms. This paper presents a comprehensive comparative study of two indexing strategies for code retrieval: raw code embeddings versus semantic natural language embeddings. We developed a dual-pipeline experimental framework that processes source code through (1) direct embedding of raw code text and (2) LLM-generated semantic descriptions followed by embedding. Both pipelines utilize identical sentence-transformer models (all-MiniLM-L6-v2) and FAISS exact search indexes to ensure fair comparison and experimental rigor.

Our methodology incorporates intelligent code chunking with AST-based segmentation for Python and fixed-size windowing for heterogeneous languages, hash-based deduplication, and persistent caching mechanisms. We evaluated both approaches across five open-source repositories (Flask, Requests, Express.js, Lodash, Spring-PetClinic) spanning Python, JavaScript, and Java, using 20 natural language queries per repository ranging from broad conceptual searches to specific function-level requests. Retrieval quality was measured using standard information retrieval metrics: Precision@K, Recall@K, Mean Reciprocal Rank (MRR), and query latency.

Our experimental results demonstrate that semantic embeddings consistently outperform raw code embeddings for natural language queries across all evaluated metrics and repositories. Specifically, semantic embeddings achieve 38% higher Precision@5, 36% improvement in MRR, and 54.5% higher Recall@1, with acceptable latency trade-offs (+8% or 3.8ms average). The performance advantages are most pronounced for broad exploratory queries (+45% P@5) where vocabulary mismatch between natural language and code identifiers is greatest, while even highly specific queries show significant improvements (+28% P@5).

Cross-language analysis reveals that dynamically-typed languages (JavaScript: +38-40%, Python: +30-37%) benefit more from semantic transformation compared to statically-typed languages (Java: +34-39%), suggesting that explicit type annotations provide inherent semantic information that partially bridges the query-code gap. Nevertheless, all languages demonstrate substantial improvements, validating the universal applicability of semantic indexing across diverse programming paradigms.

The minimal latency overhead (3.8ms average) maintains sub-65ms query processing times suitable for interactive retrieval, making semantic transformation practical for production deployment in agentic IDEs and AI-powered developer tools. Our findings provide evidence-based guidelines: while raw code embeddings may suffice for code-to-code similarity tasks or latency-critical applications, semantic transformation via LLMs should be prioritized for user-facing search interfaces where natural language queries predominate.

This work contributes to the understanding of representation strategies in neural code retrieval by demonstrating that LLM-powered semantic transformation effectively bridges the semantic gap between natural language queries and code syntax, resulting in significant, quantifiable improvements in retrieval quality for contemporary AI-assisted development environments. The complete implementation, datasets, and reproducibility package are made available to enable future research into hybrid retrieval strategies, cross-lingual generalization, and large-scale deployment scenarios.

References

[1] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in *Findings of EMNLP*, 2020.

[2] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "GraphCodeBERT: Pre-training Code Representations with Data Flow," in *ICLR*, 2021.

[3] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "IntelliCode Compose: Code Generation Using Transformer," in *ESEC/FSE*, 2020.

[4] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks," in *EMNLP*, 2019.

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

[6] X. Gu, H. Zhang, and S. Kim, "Deep Code Search," in *ICSE*, 2018.

[7] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, "When Deep Learning Met Code Search," in *ESEC/FSE*, 2019.

[8] H. Li, S. Kim, and S. Chandra, "CoCoSoDa: Effective Contrastive Learning for Code Search," in *ICSE*, 2022.

[9] M. Ciniselli, N. Cooper, L. Pascarella, D. Poshyvanyk, M. Di Penta, and G. Bavota, "An Empirical Study on the Usage of BERT Models for Code Completion," in *MSR*, 2021.

[10] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation," in *NeurIPS*, 2021.

[11] J. Johnson, M. Douze, and H. Jégou, "Billion-Scale Similarity Search with GPUs," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535-547, 2019.

[12] Y. A. Malkov and D. A. Yashunin, "Efficient and Robust Approximate Nearest Neighbor Search Using HNSW Graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 4, pp. 824-836, 2018.

[13] T. B. Brown et al., "Language Models are Few-Shot Learners," in *NeurIPS*, 2020.

[14] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating Large Language Models Trained on Code," arXiv:2107.03374, 2021.

[15] J. Shuai, L. Xu, C. Liu, M. Yan, X. Miao, and L. Liu, "Improving Automatic Source Code Summarization via Deep Reinforcement Learning," in *ASE*, 2020.

[16] C. Clement, D. Long, K. Creswell, M. Allamanis, and D. Bieber, "PyMT5: Multi-Mode Translation of Natural Language and Python Code," in *EMNLP*, 2020.

[17] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, "DéjàVu: A Map of Code Duplicates on GitHub," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1-28, 2017.

[18] C. K. Roy and J. R. Cordy, "A Survey on Software Clone Detection Research," Queen's School of Computing Technical Report No. 2007-541, 2008.
[19] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A Novel Neural Source Code Representation Based on Abstract Syntax Tree," in *ICSE*, 2019.
[20] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A Survey of Machine Learning for Big Code and Naturalness," *ACM Computing Surveys*, vol. 51, no. 4, pp. 1-37, 2018.
[21] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving Automatic Source Code Summarization via Deep Reinforcement Learning," in *ASE*, 2018.