METHODOLOGY AND TOOLING FOR AUTOMATED DETECTION OF PROGRAM CODE PLAGIARISM

Olena Chebanyuk, Oleh Osadchenko

Abstract: *This paper presents a practical methodology for automated detection of program code plagiarism that is robust to superficial modifications and scalable to large corpora. The methodology integrates light normalization of source code, filtering of template constructs (boilerplate), semantic vector representations based on transformer models, and approximate nearest neighbour (ANN) search using a Hierarchical Navigable Small World (HNSW) index.*

The proposed approach accounts for key characteristics of modern artificial intelligence systems, including context window size limitations and the problem of AI "hallucinations" in generated outputs. The methodology achieves a balanced trade-off between detection quality (PR-AUC ≈ 0.89, F1 ≈ 0.83), operational costs (P95 latency ≈ 60 ms), and interpretability of results through threshold calibration with an "uncertainty window."

Keywords: *program code reverse engineering, artificial intelligence, semantic code analysis, plagiarism detection, code embeddings, HNSW.*

ITHEA Keywords: *I.2 ARTIFICIAL INTELLIGENCE*

DOI: *https://doi.org/10.54521/ijita33-01-p04*

1. Introduction: Research Motivation

Program code plagiarism constitutes a systemic threat to academic integrity and engineering practice, as it substitutes genuine knowledge acquisition and distorts the assessment of developers' competencies. Plagiarism is understood as the appropriation of another person's intellectual output or the reproduction of its fragments without proper attribution while presenting them as one's own

work. In the context of source code, this encompasses four basic forms: direct (verbatim) plagiarism, mosaic (compilative) plagiarism, paraphrasing plagiarism, and self-plagiarism.

The relevance of this problem is driven by the rapid growth of open repositories and code reuse within open-source software ecosystems. Global platforms and package managers facilitate legitimate code reuse, but simultaneously lower the barrier to unauthorized copying with minimal obfuscation. In educational environments, this manifests as direct copying or mosaic combination of fragments; in industrial settings, as improper reuse of modules without compliance with licensing requirements.

The widespread adoption of automated code completion and generation tools (intelligent suggestions, language models) further increases the risk of paraphrasing plagiarism, in which logic, structure, and algorithmic invariants are preserved while the surface form of the code is altered. In such cases, lexical methods (n-grams, MinHash/SimHash) prove insufficient, as they are sensitive to renaming, formatting changes, and local refactorings.

2. Vocabulary of main terms used in the area of source code automated plagiarism detection

ANN (Approximate Nearest Neighbors) – a method for approximate nearest-neighbor search in high-dimensional spaces that provides suboptimal but computationally efficient solutions for similarity search tasks [Saeed, 2024] [Malkov and Yashunin, 2020].

AST (Abstract Syntax Tree) – an abstract syntax tree; a data structure representing the syntactic structure of program code in a tree form, used for program analysis and transformation [Zou et al., 2020] [Roy and Cordy, 2007].

Boilerplate (Boilerplate code) – repetitive code fragments, template constructs, scaffolding elements, and library stubs that frequently occur across projects and do not carry individual semantic meaning [Roy and Cordy, 2007] [Ramalho, n.d.].

Bootstrap resampling – a statistical method of repeated random sampling with replacement used to estimate confidence intervals and assess result stability [Manning et al., n.d.] [Powers, 2020].

Code embeddings – fixed-dimensional numerical vectors that encode semantic and structural characteristics of program code fragments in a continuous feature space [Feng et al., 2020] [Guo et al., 2021].

AI hallucinations – a phenomenon in which an artificial intelligence system generates information that does not correspond to reality or the input data; manifested as fabricated facts or the mixing of unrelated data [IBM, 2023].

Document frequency (DF) – a metric indicating the proportion of documents in a corpus that contain a given term or shingle; used to identify frequently recurring elements [Manning et al., n.d.].

Embedding – see Code embeddings.

P95 latency – the 95th percentile of system response time; a performance metric exceeded in only 5% of cases, used to evaluate behaviour under load [ClickHouse Authors, n.d.] [FAISS Team, n.d.].

Interactive time – a system response time acceptable for interactive user interaction, typically in the range of tens to hundreds of milliseconds [ClickHouse Authors, n.d.].

Threshold calibration – the process of determining optimal decision threshold values based on a validation dataset while balancing precision and recall [Saito and Rehmsmeier, 2015] [Powers, 2020].

Cosine similarity – a measure of similarity between two vectors, computed as the cosine of the angle between them; widely used for comparing vector representations of documents and code [Manning et al., n.d.] [FAISS Team, n.d.].

Context window – a limit on the number of tokens that a large language model can process in a single pass [Saeed, 2024].

Light normalization – minimal preprocessing of source code, including removal of comments, normalization of whitespace, standardization of literals, and

abstraction of identifiers without changing program semantics [Roy and Cordy, 2007] [Ramalho, n.d.].

Lexical methods – code analysis methods based on processing token sequences and their hash compressions; include n-grams, winnowing, MinHash, and SimHash [Roy and Cordy, 2007] [Saini and Singh, 2018].

Mean pooling – an aggregation method in which the final vector representation is formed by averaging hidden states over all tokens in a sequence, taking the attention mask into account [Howard and Gugger, n.d.].

Mermaid – a textual diagram description format that allows creating UML diagrams, flowcharts, and other visualizations using a simple text notation [Mermaid, 2023].

MinHash – a hashing algorithm for fast estimation of Jaccard similarity between sets; commonly used for detecting duplicates and near-duplicate documents [Roy and Cordy, 2007] [Saini and Singh, 2018].

MRR (Mean Reciprocal Rank) – the mean reciprocal rank of the first relevant result; a ranking quality metric indicating how quickly a user finds the desired result [Google Research, n.d.] [CodeXGLUE Authors, n.d.].

n-grams – sequences of n consecutive elements (characters, tokens, or words); a basic unit of analysis in lexical similarity detection methods [Roy and Cordy, 2007] [Saini and Singh, 2018].

PDG (Program Dependence Graph) – a program dependence graph representing data and control flows between operations; used for structural code analysis [Zou et al., 2020] [Görg, 2015].

Paraphrasing plagiarism – a type of plagiarism in which logic, structure, and algorithmic invariants are preserved, but the surface form of the code is altered through renaming, reordering, or refactoring [Roy and Cordy, 2007].

Precision@k – the proportion of relevant results among the top k returned by the system; a quality metric for search and ranking tasks [Manning et al., n.d.] [Google Research, n.d.].

PR-AUC (Precision–Recall Area Under the Curve) – the area under the precision–recall curve; an integral classification quality metric, especially useful under class imbalance [Manning et al., n.d.] [Saito and Rehmsmeier, 2015].

Verbatim plagiarism – direct copying of code fragments without attribution [Roy and Cordy, 2007].

Self-plagiarism – reuse of one's own previously published code without appropriate references or permission [Roy and Cordy, 2007].

Semantic similarity – approximate behavioral similarity of program fragments, where equivalence is not guaranteed but stable coincidence of observable effects is detected for a sufficiently broad subset of inputs [Saeed, 2024] [IBM, 2023].

SimHash – a locality-sensitive hashing algorithm that estimates document similarity via the Hamming distance between hash values [Roy and Cordy, 2007] [Saini and Singh, 2018].

Stop-list – a list of high-frequency shingles or tokens that are ignored during analysis as weakly discriminative [Manning et al., n.d.].

Structural methods – code analysis methods based on comparing abstract syntax trees (ASTs) or program dependence graphs (PDGs) [Zou et al., 2020] [Roy and Cordy, 2007].

Transformer models – a class of neural network architectures based on the attention mechanism, used for sequence processing, including program code [Vaswani et al., 2017] [Feng et al., 2020].

F1-score – the harmonic mean of precision and recall; a metric that balances type I and type II errors [Manning et al., n.d.] [Powers, 2020].

Boilerplate filtering – the process of identifying and excluding template constructs from analysis to reduce false positives [Roy and Cordy, 2007] [Ramalho, n.d.].

HNSW (Hierarchical Navigable Small World) – a hierarchical navigable small-world graph structure; an efficient index for approximate nearest-neighbor

search in high-dimensional vector spaces [Malkov and Yashunin, 2020] [ClickHouse Authors, n.d.].

False positives (FP) – cases where the system incorrectly identifies dissimilar fragments as similar, leading to unjustified plagiarism accusations [Manning et al., n.d.] [Saito and Rehmsmeier, 2015].

False negatives (FN) – cases where the system misses real instances of plagiarism by failing to detect similar fragments [Manning et al., n.d.] [Saito and Rehmsmeier, 2015].

Shingle – a sequence of tokens of fixed length n, used as a basic unit for computing similarity between documents [Roy and Cordy, 2007] [Saini and Singh, 2018].

Shingling – a method of splitting text or code into overlapping fixed-length sequences for subsequent similarity analysis [Roy and Cordy, 2007] [Saini and Singh, 2018].

CodeBERT – a pretrained transformer model for representing program code, trained on code–text pairs and large code corpora [Feng et al., 2020] [33].

ClickHouse – a column-oriented analytical DBMS optimized for online analytical processing (OLAP) with support for vector indexes [ClickHouse Authors, n.d.].

DuckDB – an embedded column-oriented analytical DBMS optimized for local computation and analytics [DuckDB Labs, n.d.].

Winnowing – an algorithm for selecting a representative subset of document hashes for efficient copy detection [Roy and Cordy, 2007] [Saini and Singh, 2018].

3. Research questions

RQ1: How can light normalization and boilerplate filtering improve the robustness of plagiarism detection systems against superficial code modifications?

RQ2: To what extent do semantic embeddings based on transformer models outperform lexical and structural methods in detecting paraphrasing plagiarism?

RQ3: What is the optimal configuration of HNSW index parameters (M, efConstruction, efSearch) to achieve a balance between detection accuracy and P95 latency for large-scale code corpora?

RQ4: How does threshold calibration with an uncertainty window reduce decision variability and improve interpretability in automated plagiarism detection?

RQ5: What trade-offs exist between detection quality (PR-AUC, F1-score) and operational costs (RAM usage, index size, latency) when applying semantic similarity methods compared to traditional approaches?

RQ6: Can a hybrid search strategy combining keyword filtering and ANN-based semantic retrieval significantly reduce the volume of code that needs to be analyzed without compromising detection accuracy?

4. Review of the related papers

Automated detection of source-code plagiarism is closely related to the broader area of code clone detection and code similarity search. The literature commonly classifies approaches by the type of signal they use-lexical (surface form), structural (syntax and program structure), or semantic (behavioral meaning)-and by their operational setting, ranging from offline pairwise comparison to interactive large-scale retrieval. A canonical taxonomy of clone types and methodological families is presented in the survey by Roy and Cordy, which remains widely used as a reference for terminology and evaluation perspectives in clone detection research [Roy and Cordy, 2007]. Complementary viewpoints can be found in more recent summaries of machine-learning techniques for clone detection, which emphasize the transition from hand-crafted similarity heuristics to learned representations [Saini and Singh, 2018].

4.1 Lexical and token-based similarity methods

Lexical approaches operate on text or token sequences and typically rely on n-grams (shingles), fingerprinting, or set similarity. Their practical strengths are speed, simplicity, and relatively high explainability: similarity evidence can often

be traced back to overlapping tokens or matched regions. In plagiarism-detection pipelines, lexical methods are almost always combined with normalization (comment removal, whitespace unification, identifier abstraction) to reduce sensitivity to superficial edits, as also highlighted in clone-detection surveys [Roy and Cordy, 2007]. Nevertheless, lexical overlap tends to degrade under systematic paraphrasing (consistent renaming, local refactorings, statement reordering), motivating stronger representations when the threat model includes "meaning-preserving rewrite" scenarios.

An additional practical obstacle for lexical methods is "template bias": scaffolding and recurring boilerplate fragments can dominate similarity scores, producing false positives when many solutions share the same wrapper code. Consequently, shingling-based workflows often include frequency-based filtering (stop-lists of high-frequency shingles) to reduce the influence of template artifacts, a theme that reappears across many applied systems and is consistent with general IR intuition about down-weighting common terms [Manning et al., n.d.].

4.2 Structural and graph-based clone detection

Structural approaches aim to represent code beyond tokens, commonly via ASTs, and in more advanced methods via graphs that incorporate data/control dependencies. These techniques can improve invariance to superficial transformations and provide stronger evidence than purely lexical overlap, but they also introduce non-trivial engineering costs (language-specific parsing frontends, graph construction) and may suffer from scaling constraints when applied to large corpora.

Graph-based methods have been explored to capture deeper program properties and improve resilience to refactorings. CCGraph is a representative PDG-based clone detection approach that uses approximate graph matching to reduce the computational burden typically associated with graph comparisons [Zou et al., 2020]. The importance of configuration and performance tuning for PDG-based clone detection has also been emphasized, showing that operational feasibility depends strongly on parameterization and implementation details rather than on the theoretical model alone [Görg, 2015]. These works

suggest that structural verification is particularly attractive as a secondary stage (re-ranking or confirmation) for a small candidate set, rather than as a primary retrieval mechanism for interactive search at scale.

4.3 Semantic representations and transformer-based models

A major recent trend is the use of learned semantic representations of code (embeddings), largely driven by advances in transformer architectures. The Transformer model introduced by Vaswani et al. enabled scalable contextual representation learning and became a foundation for modern pretrained language models [Vaswani et al., 2017]. In the programming-language domain, CodeBERT leverages large-scale pretraining on code and natural language to learn contextual embeddings that can be transferred to code search and similarity tasks [Feng et al., 2020]. GraphCodeBERT extends this direction by incorporating data-flow signals into pretraining, aiming to better capture semantic relationships not fully represented by the token stream alone [Guo et al., 2021].

Embedding-based approaches are attractive for plagiarism detection because they can remain stable under transformations that change surface form but preserve algorithmic intent (e.g., renaming and limited refactoring). However, semantic methods introduce new challenges: (i) embedding similarity can still be affected by boilerplate constructs; (ii) similarity scores are less directly explainable than explicit token matches; and (iii) operational performance depends on retrieval infrastructure and index tuning. These properties motivate hybrid pipelines that combine semantic retrieval with explicit boilerplate control and additional signals for interpretability.

A related, practical consideration is that modern AI tools can rewrite code while preserving meaning. When LLMs are used during code generation or paraphrasing, issues such as limited context windows and hallucinated or inconsistent code fragments can influence the observed similarity patterns and error modes of plagiarism detection systems [Saeed, 2024], [IBM, 2023]. While such sources are not clone-detection papers per se, they contextualize why robustness to paraphrasing and partial rewrites has become operationally relevant.

4.4 Similarity search, ANN retrieval, and scalability

Once code fragments are embedded into a dense vector space, plagiarism detection can be framed as nearest-neighbor retrieval under a similarity metric (commonly cosine similarity). For large collections, exact k-NN search becomes expensive, which makes approximate nearest neighbor (ANN) techniques central to building interactive systems. HNSW is a widely adopted ANN method with a strong empirical quality–latency trade-off, formalized and analyzed by Malkov and Yashunin [Malkov and Yashunin, 2020]. ANN benchmarking work stresses that retrieval quality must be evaluated together with latency distributions, memory usage, and build-time costs, rather than focusing only on average query time [Aumüller et al., 2017].

From a systems standpoint, embedding-based retrieval can be implemented via specialized libraries (e.g., FAISS) or integrated into database engines that support vector indexing alongside structured filtering [FAISS Team, n.d.]. In database-centric architectures, ClickHouse provides a columnar storage engine with support for vector search workflows and HNSW indexing, enabling combined execution of similarity retrieval and metadata constraints in a single system [ClickHouse Authors, n.d.], [ClickHouse Authors, n.d.]. For experimentation and baseline measurement, DuckDB is often used as a convenient embedded analytical DBMS for brute-force or exact computations at smaller scales, serving as a reference point when validating approximate retrieval quality [DuckDB Labs, n.d.], [DuckDB Labs, n.d.].

4.5 Benchmarks, datasets, and evaluation protocols

A persistent methodological issue in clone detection and plagiarism research is comparability across datasets and evaluation protocols. BigCloneBench is among the most widely used datasets for code clone detection, enabling more standardized comparisons across approaches [BigCloneBench, n.d.]. For semantic code models and retrieval-style evaluation, the CodeXGLUE benchmark suite provides tasks and metrics for code understanding, code search, and related problems that overlap with similarity assessment objectives [CodeXGLUE Authors, n.d.]. These benchmarks support evaluation beyond

anecdotal examples and facilitate controlled analysis across languages, clone types, and transformation regimes.

Regarding metrics, the information retrieval literature provides established evaluation tools for ranked retrieval and classification [Manning et al., n.d.]. For imbalanced classification settings-common in plagiarism detection where true matches are sparse relative to all possible pairs-precision–recall analysis is often more informative than ROC-based evaluation, as argued by Saito and Rehmsmeier [Saito and Rehmsmeier, 2015]. Precision, recall, and F-measure are also standard, and their interpretation in operational decision-making has been discussed in detail by Powers [Powers, 2020]. In retrieval pipelines where top-k candidates are presented to a reviewer, top-k measures such as Precision@k and related metrics are directly aligned with user-facing performance; benchmark discussions in large-scale retrieval contexts emphasize how such metrics capture ranking usefulness under constrained review budgets [Google Research, n.d.].

4.6 Hybrid pipelines and the quality–cost–interpretability trade-off

Across the reviewed families, a recurring conclusion is that no single method simultaneously optimizes robustness to paraphrasing, scalability, and explainability. Lexical methods are efficient and interpretable but fragile under meaning-preserving rewrites [Roy and Cordy, 2007]. Structural and PDG-based methods can capture deeper program properties but often face scaling and engineering constraints, making approximate matching and tuning crucial [Zou et al., 2020], [Görg, 2015]. Transformer-based embeddings improve robustness to surface edits and can generalize across styles and partial refactorings [Feng et al., 2020], [Guo et al., 2021], but they require ANN retrieval to scale and typically need auxiliary mechanisms (boilerplate suppression, threshold calibration, evidence presentation) to produce actionable outcomes in high-stakes settings.

Consequently, many practical systems follow a hybrid cascade: (1) deterministic normalization and lightweight filtering; (2) semantic embedding retrieval with ANN (e.g., HNSW) [Malkov and Yashunin, 2020], optionally supported by dedicated libraries [FAISS Team, n.d.] or database-integrated indexing

[ClickHouse Authors, n.d.], [ClickHouse Authors, n.d.]; and (3) optional structural verification or alignment over the top-k candidate set for increased precision and interpretability [Zou et al., 2020]. This hybrid view aligns naturally with evaluation best practices emphasizing metric diversity (PR-AUC, F1, Precision@k) and operational measurements (tail latency, memory footprint, index size) [Manning et al., n.d.], [Saito and Rehmsmeier, 2015], [Aumüller et al., 2017].

5. Proposed Approach to Automated Detection of Source-Code Plagiarism

5.1. Overview

The proposed approach implements a multi-stage pipeline for automated plagiarism detection in source code, combining lightweight preprocessing, semantic analysis, and approximate nearest neighbor search (Figure 1). The methodology is designed to handle four types of plagiarism: verbatim copying, mosaic plagiarism, paraphrasing plagiarism, and self- plagiarism, while maintaining interactive response times suitable for large-scale educational and industrial deployments.

5.2 Documentation of algorithm steps is performed.

At the initial stage, the overall workflow of the plagiarism detection process is documented. The general logic of the pipeline is visualized in the General_algorithm diagram, which shows the main stages: input, normalization, shingling, embedding, ANN retrieval, threshold calibration, and reporting. This documentation serves as a foundation for reproducible analysis and establishes clear expectations for each processing step.

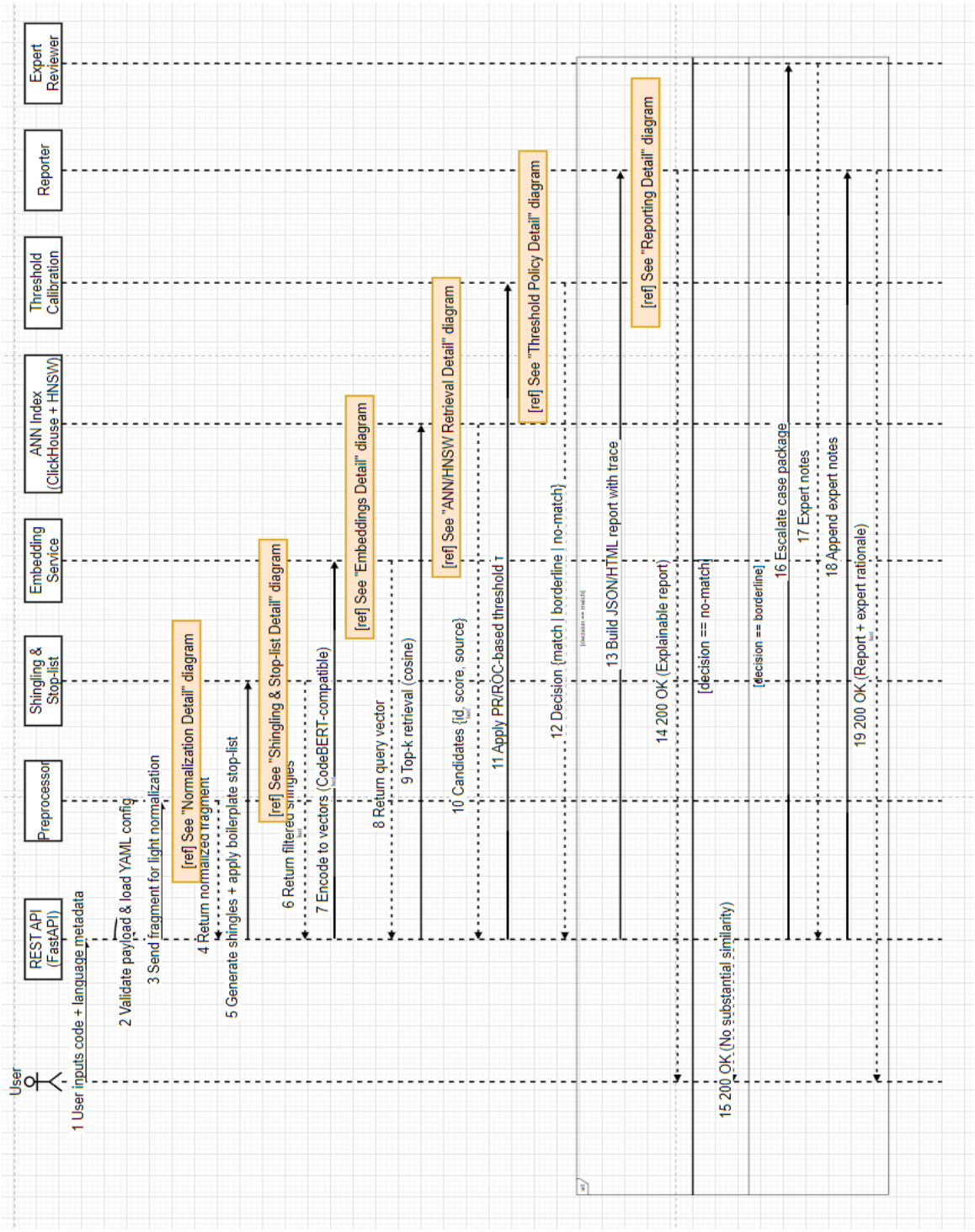

Figure 1. Multi-stage pipeline for automated plagiarism detection in source code, combining lightweight preprocessing, semantic analysis, and approximate nearest neighbor search.

5.3 Light normalization of the source code is applied.

The submitted code fragment undergoes preprocessing to remove comments and blank lines, unify whitespace, and abstract identifiers into stable surrogates (e.g., v1, v2). This ensures invariance to stylistic changes without losing semantic structure.

See figure 2, "Normalization diagram, which illustrates sequential operations: cleaning, whitespace unification, and identifier abstraction.

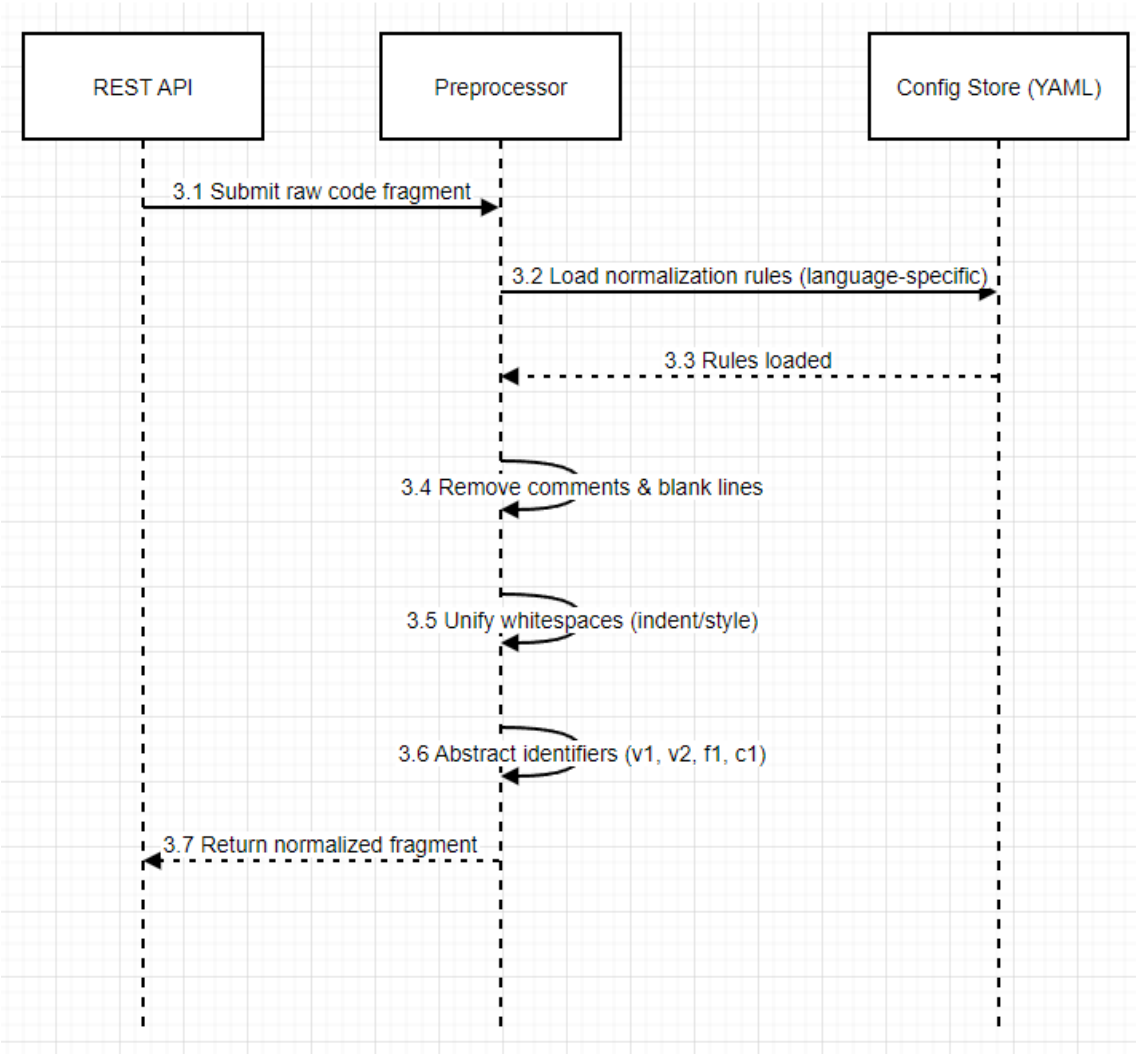

Figure 2. Detailed diagram of normalization process

5.4 Shingling and boilerplate filtering are performed.

Token n-grams (shingles) are generated iteratively from the normalized code. A stop-list is applied to filter out high-frequency boilerplate patterns such as import statements and test scaffolds, reducing false positives.

See figure 3 "Shingling & Stop-list Detail" diagram, which shows the loop for shingle generation and the application of stop-lists to remove repetitive patterns.

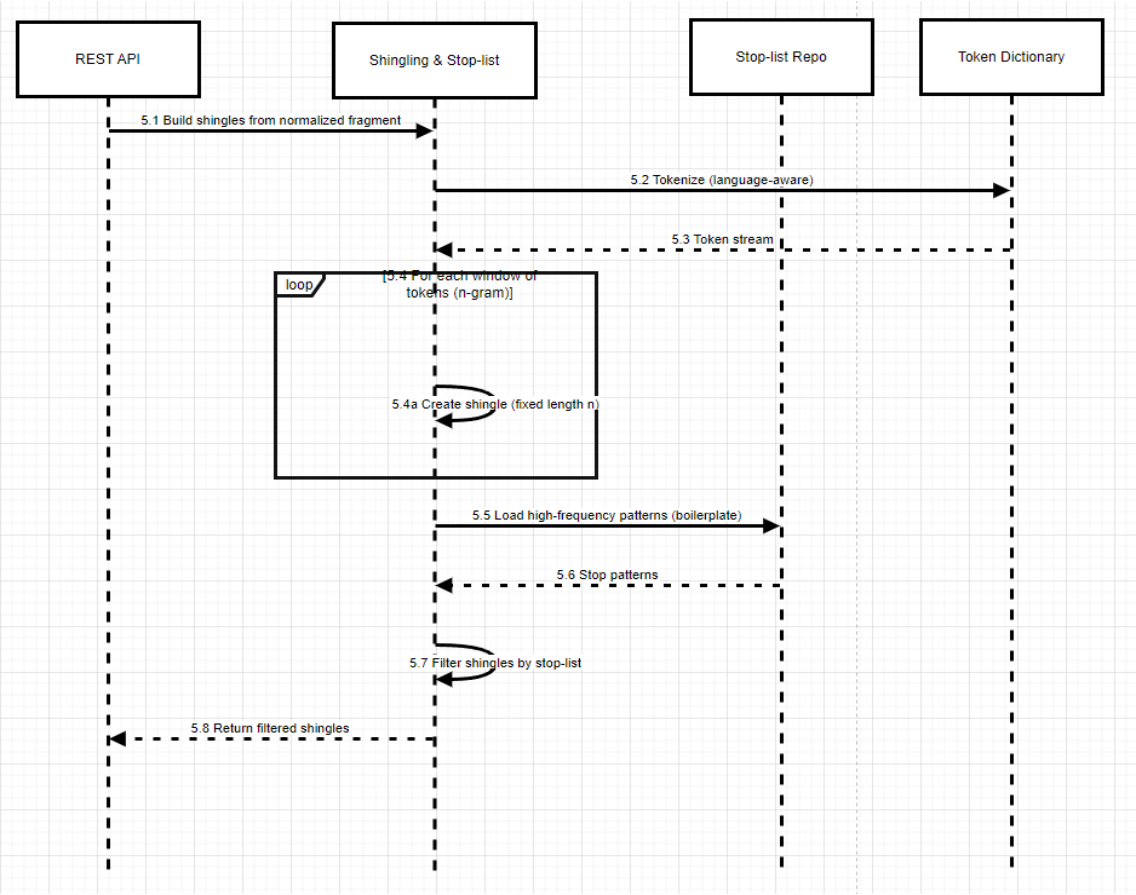

Figure 3. Shingling & Stop-list Detail.

5.5 Semantic embeddings are generated.

Filtered shingles are encoded into dense vector representations using a CodeBERT-compatible model. Mean pooling and ℓ_2 -normalization are applied, and parallel execution optimizes batching and hardware utilization.

See figure 4 "Embeddings Detail" diagram, which demonstrates tokenization, pooling, normalization, and parallel hardware optimization.

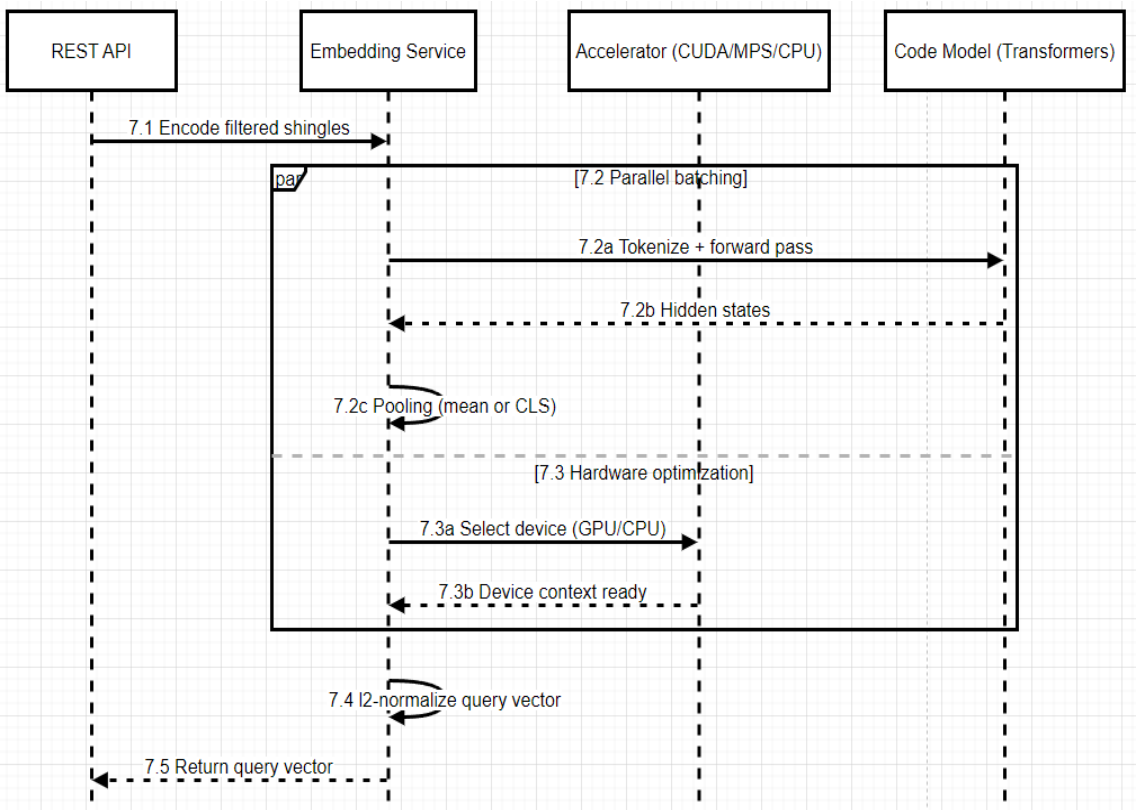

Figure 4. Embeddings Detail.

5.6 Approximate nearest neighbor retrieval is executed.

The query vector is submitted to the ANN Index implemented in ClickHouse with HNSW indexing. Top-k retrieval is performed using cosine similarity. Index parameters (M, efConstruction, efSearch) are tuned for latency and accuracy, and cache state determines search strategy.

See figure 5 "ANN/HNSW Retrieval Detail" diagram, which depicts the branching logic for warm/cold cache and the steps for candidate deduplication.

HNSW Index Search:

- Navigate multi-level graph structure starting from entry point

- Use greedy search with candidate set expansion controlled by efSearch parameter
- Return top-k candidates ranked by cosine similarity

Cache-Aware Strategy:

- For warm cache: proceed directly with HNSW search
- For cold cache: consider hybrid approach with preliminary filtering

Candidate Deduplication:

- Remove duplicate entries by source file and author constraints
- Preserve author separation established during train/validation/test split
- Return deduplicated top-k list with similarity scores

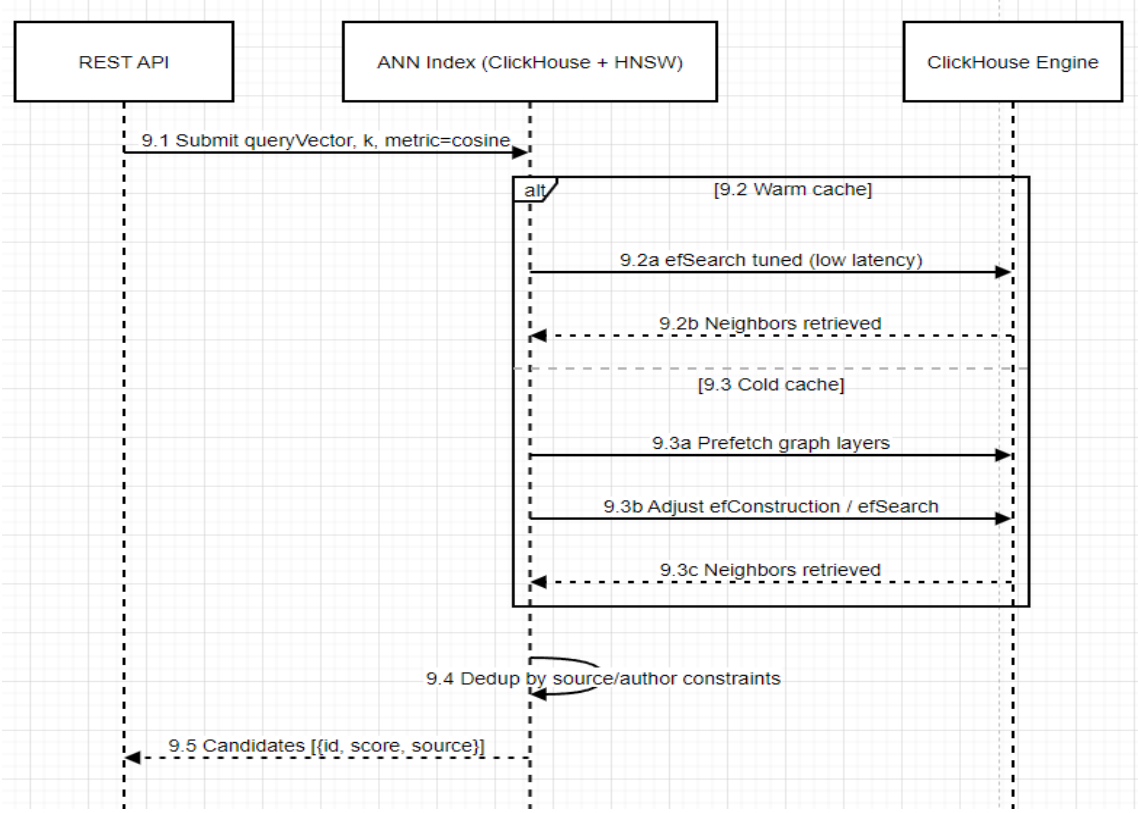

Figure 5. ANN/HNSW Retrieval Detail.

The ANN/HNSW Retrieval Detail diagram depicts the branching logic for warm/cold cache and the steps for candidate deduplication. This stage achieves P95 latency of approximately 60ms for top-10 queries.

1. Candidate post-processing and threshold calibration are applied. Retrieved candidates are deduplicated by source and author constraints. A calibrated threshold τ (derived from PR/ROC analysis) is applied, and the decision policy selects one of three outcomes: match, no-match, or borderline.
2. Report generation and escalation are performed.
3. Optional telemetry and audit are supported.
4. The user can request normalized snippets and trace maps via GET /code/{id} for educational or compliance purposes.

6. Experimental Description

6.1. Description of the Initial data to analysis

The experiment uses a multilingual corpus of code fragments (Java, Python, JavaScript, Go) with annotations of "match / non-match" pairs and with author separation across the training, validation, and test subsets. The core sources and composition of the corpora are based on public datasets for code clone detection, in particular BigCloneBench [BigCloneBench, n.d.] and the code clone detection task within CodeXGLUE [CodeXGLUE Authors, n.d.].

The final aggregation is presented in a summary table describing subset assignments, the number of fragments, and the number of positive and negative class pairs:

6.2. Description of the used frameworks

The experiments were conducted on a standard workstation equipped with a central processing unit and sufficient RAM to support interactive search over a vector index. The versions of the libraries used for constructing vector representations (PyTorch and Transformers), the version of the database management system with a graph-based approximate search index (ClickHouse), as well as the model version identifier and the index configuration parameters, were fixed and recorded in auxiliary artifacts.

Table 1. Dataset, languages, and number of fragments

Dataset	Languages	Number of fragments	Number of positive pairs	Number of negative pairs	Subset assignment
CodeXGLUE (filtered BigCloneBench)	Java	9,134	207,800	1,524,060	train 901,028; val 415,416; test 415,416
Internal training set	Python, JavaScript	4,524	3	predominantly test; partially val	
Combined corpus for threshold calibration	Java, Python	5,000	5,500	14,500	train (5×2 cross-validation)

6.3. Experiment workflow

6.3.1. Data preparation, splitting strategy, and leakage control

The evaluation is performed on a multilingual corpus of code fragments with annotated "similar / not similar" relations, selected to reflect heterogeneous programming styles and domains (algorithmic tasks and application-like snippets). To ensure external validity and prevent optimistic bias, dataset splits are constructed so that closely related "families" of solutions (e.g., originating from the same assignment template, author group, or strongly overlapping

scaffolding) do not intersect between training/validation and test partitions. This split discipline reduces information leakage and better approximates real deployment, where the system must generalize to unseen authors and tasks.

Stage 1: deterministic light normalization

Each fragment is transformed into a deterministic representation that is invariant to non-informative edits. The normalization pipeline removes comments and empty lines, unifies whitespace, normalizes literals (e.g., mapping numeric constants to a placeholder), and abstracts identifiers by mapping variable/function names to a stable sequence (v1, v2, …) based on order of appearance. The guiding principle is "minimum intervention": remove surface noise that is frequently manipulated during paraphrasing, while preserving syntactic keywords and control-flow structure that represent the algorithmic skeleton. The procedure is linear in the fragment length and does not require language-specific AST tooling, which supports multi-language applicability and stable throughput.

This stage operationalizes RQ1 by providing robustness to superficial modifications (renaming, formatting, comment rewriting) without attempting full semantic equivalence proving.

Stage 2: shingling, boilerplate estimation, and stop-list construction

After normalization, each fragment is converted into token shingles (n-grams). Global shingle frequencies are computed across the corpus, and a stop-list of high-frequency shingles is built to suppress recurring scaffolding (imports, wrappers, template blocks) that otherwise inflates false positives. In addition to hard filtering (ignoring stop-shingles), the workflow computes a boilerplate ratio ρ (the fraction of stop-shingles in the fragment), which is later used as an explicit correction factor during scoring. This makes the influence of "template similarity" measurable and controllable rather than implicit.

Stage 3: semantic embedding construction

Normalized code is embedded with a pretrained transformer model of the CodeBERT family. Token embeddings are aggregated via mean pooling over the attention mask and then L2-normalized, so cosine similarity can be

computed as a dot product in the normalized vector space. The workflow fixes tokenizer constraints (e.g., maximum sequence length with truncation/padding) and treats the model name and embedding configuration as versioned experimental artifacts.

This stage is central to RQ2, enabling semantic retrieval beyond lexical overlap and mitigating paraphrasing plagiarism where surface form differs substantially while behavior remains close.

Stage 4: ANN indexing and similarity search (HNSW)

All corpus embeddings are persisted in a vector-capable analytical storage layer and indexed using an HNSW (Hierarchical Navigable Small World) structure. Similarity search returns top-k nearest neighbors for each query fragment under cosine similarity. The workflow evaluates multiple HNSW configurations, focusing on the parameters that directly govern the quality–latency trade-off (notably M and efSearch in the reported configuration), and selects an operationally balanced point.

This stage answers RQ3 by selecting an HNSW parameterization that provides competitive retrieval quality while controlling tail latency (P95), which is critical for interactive plagiarism-checking scenarios.

Stage 5: score correction, hybrid filtering option, and decision policy

The base similarity signal is cosine similarity (s). To reduce the contribution of boilerplate, the workflow applies a linear penalty based on boilerplate ratio ρ (e.g., $s' = s - \alpha \cdot \rho$), where α is tuned on the validation split. This makes the pipeline explicitly resistant to shared templates and recurring scaffolding.

In addition, the workflow supports a hybrid strategy: inexpensive keyword/shingle-based filtering can reduce the candidate pool before ANN retrieval, decreasing the volume of heavy semantic analysis without materially degrading retrieval quality (addressing RQ6).

Finally, decisions are produced using a two-threshold policy with an uncertainty window. Two calibrated thresholds (T_{low}, T_{high}) partition the score range into three regions: (1) "probable match" above , T_{high}, (2) "insufficient evidence"

below (T_{low}, and (3) "requires review" inside the uncertainty window. This policy reduces decision volatility near the boundary and supports interpretable escalation for borderline cases rather than forcing an overconfident binary verdict.

To ensure operational consistency, the calibrated thresholds and auxiliary parameters (model version, normalization version, validity interval) are stored as versioned records in the database and retrieved by the service endpoint used by the UI (e.g., GET /thresholds/current), enabling auditability and repeatable decision logic over time.

This stage directly answers RQ4 by turning calibration into a controlled, explainable decision policy ("auto-confirm / auto-reject / escalate").

6.3.2. Measurement protocol, metrics, and statistical treatment

The evaluation uses a combined "quality–cost–interpretability" profile. For quality, the primary metrics are PR-AUC (robust under class imbalance), F1 at a selected threshold, Precision@k (k=10 in the reported setting), and MRR. For operational cost, the workflow measures P95 latency, peak RAM consumption, and index size on disk. Interpretability is additionally assessed via availability of artifacts usable for human review (nearest neighbors with scores, boilerplate signals, version metadata).

To reduce measurement noise and enforce reproducibility, the workflow fixes random seeds, model/library versions, and index parameters, and runs latency experiments in two regimes: cold cache immediately after service restart and warm cache after stabilization. Multiple iterations are executed while discarding initial runs, consistent with standard performance-testing practice. For uncertainty estimation, bootstrap confidence intervals (95%) are used, and paired comparisons across variants can be validated with permutation testing or McNemar-style checks on a fixed sample where applicable.

6.3.3. Comparative baselines and quality–cost analysis

To answer RQ2 and RQ5, the workflow includes representative baselines from three families: lexical methods (e.g., winnowing/MinHash, SimHash), structural

methods (AST/PDG comparisons), and semantic approaches (embeddings with brute-force k-NN vs embeddings with HNSW). The summary "quality–cost" comparison shows that semantic methods provide the highest integral quality, while ANN indexing is required to make them operationally feasible under interactive constraints.

In the reported results, the best operational configuration (semantic embeddings + HNSW + boilerplate filtering) achieves PR-AUC = 0.89, F1 = 0.83, Precision@10 = 0.79, MRR = 0.75, with P95 latency ≈ 60 ms, RAM ≈ 720 MB, and index size ≈ 3.9 GB. This point substantially improves tail latency compared to brute-force k-NN retrieval (P95 ≈ 520 ms) while preserving strong detection quality. These empirical findings support the core claim that a carefully tuned semantic + ANN pipeline yields a superior quality–cost balance for large corpora under realistic latency constraints.

Conclusion

The proposed approach integrates semantic code analysis through AI assistance into traditional plagiarism detection methods for applications with multi-layer architectural style. The main contributions are as follows:

- Improved Detection Pipeline
- Enhanced Decision Interpretation
- Quality-Cost Balance Analysis is performed

The method introduces light normalization combined with stop-shingle filtering, which systematically reduces noise from boilerplate and template constructs that previously caused false positives. This improvement is reflected in higher Precision@k without loss of Recall on paraphrasing cases, resulting in better PR-AUC values.

Threshold calibration is implemented using an uncertainty window approach, which aligns statistical metrics such as PR-curves and bootstrap intervals with operational error risks. This alignment standardizes decision policies and reduces variability caused by the human factor during expert review.

The study provides the first systematic comparison of quality-cost balance for lexical, structural, and semantic methods under given corpus conditions, explicitly considering P95 latency, RAM consumption, and index size. Practical guidelines are formulated regarding HNSW parameters, optimal inference modes for embeddings, and effective boilerplate filtering levels for different languages and fragment lengths.

Bibliography

[Saeed, 2024] T. Saeed. *Understanding Context Windows in Large Language Models (LLMs).* Medium, 2024. URL: https://medium.com/@tahir.saeed_46137/understanding-context-windows-in-large-language-models-llms-4ad3dca6b86f

[IBM, 2023] IBM. *AI Hallucinations.* IBM Think, 2023. URL: https://www.ibm.com/think/topics/ai-hallucinations

[Mermaid, 2023] Mermaid. *Diagramming and charting tool.* Mermaid.js, 2023. URL: https://mermaid.js.org/

[ISO/IEC, 2015] ISO/IEC 26550:2015. *Software and systems engineering - Reference model for product line engineering and management.* International Organization for Standardization, Geneva, 2015.

[Zou et al., 2020] Y. Zou, et al. *CCGraph: a PDG-based Code Clone Detector with Approximate Graph Matching.* In: Proceedings of the ASE 2020 (Automated Software Engineering), 2020. URL: https://yinxingxue.github.io/papers/ase2020_CCGraph.pdf

[Görg, 2015] T. Görg. *Performance Tuning of PDG-based Code Clone Detection.* Softwaretechnik-Trends, Band 35, Heft 2, 2015. URL: https://fb-swt.gi.de/fileadmin/FB/SWT/Softwaretechnik-Trends/Verzeichnis/Band_35_Heft_2/06_T_Goerg.pdf

[Roy and Cordy, 2007] C.K. Roy, J.R. Cordy. *A Survey on Software Clone Detection Research.* Queen's University, Technical Report 2007-541, 2007. URL: https://www.cs.queensu.ca/TechReports/Reports/2007-541.pdf

[Vaswani et al., 2017] A. Vaswani, et al. *Attention Is All You Need.* In: Proceedings of NeurIPS, 2017. arXiv:1706.03762.

[Feng et al., 2020] Z. Feng, et al. *CodeBERT: A Pre-Trained Model for Programming and Natural Languages.* Findings of EMNLP, 2020. arXiv:2002.08155.

[Guo et al., 2021] D. Guo, et al. *GraphCodeBERT: Pre-training Code Representations with Data Flow.* In: Proceedings of ICLR, 2021. arXiv:2009.08366.

[Malkov and Yashunin, 2020] Y. Malkov, D. Yashunin. *Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs.* IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI), 2020. arXiv:1603.09320.

[Saini and Singh, 2018] V. Saini, P. Singh. *A Survey of Machine Learning Techniques for Code Clone Detection.* International Journal of Applied Engineering Research, vol. 13, no. 6, 2018, pp. 2054–2059.

[ClickHouse Authors, n.d.] ClickHouse Authors. *ClickHouse Documentation - Architecture and Vector Search.* URL: https://clickhouse.com/docs/ (accessed: 22.12.2025).

[ClickHouse Authors, n.d.] ClickHouse Authors. *Vector Indexes (HNSW) - Official Documentation.* URL: https://clickhouse.com/docs/en/engines/table-engines/mergetree-family/hnsw (accessed: 22.12.2025).

[Howard and Gugger, n.d.] J. Howard, S. Gugger. *Deep Learning for Coders with fastai and PyTorch.* O'Reilly Media. URL: https://www.fast.ai (accessed: 22.12.2025).

[Ramalho, n.d.] T. Ramalho. *Made With ML - Production Machine Learning Systems.* URL: https://madewithml.com (accessed: 22.12.2025).

[DuckDB Labs, n.d.] DuckDB Labs. *DuckDB Documentation - In-Process Analytical Database.* URL: https://duckdb.org/docs/ (accessed: 22.12.2025).

[Manning et al., n.d.] C. Manning, P. Raghavan, H. Schütze. *Introduction to Information Retrieval - Evaluation Metrics.* URL: https://nlp.stanford.edu/IR-book/ (accessed: 22.12.2025).

[Saito and Rehmsmeier, 2015] T. Saito, M. Rehmsmeier. *The Precision-Recall Plot Is More Informative than the ROC Plot when Evaluating Binary Classifiers.* PLoS ONE, 2015. DOI: 10.1371/journal.pone.0118432.

[Powers, 2020] D.M. Powers. *Evaluation: Precision, Recall, and F-Measure.* arXiv:2010.16061, 2020.

[Google Research, n.d.] [Google Research, n.d.] Google Research. *Understanding Top-k Metrics (Precision@k, Recall@k) in Large-Scale Retrieval Systems.* URL: https://research.google (accessed: 22.12.2025).

[Aumüller et al., 2017] M. Aumüller, et al. *ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms.* In: Proceedings of SISAP, 2017. arXiv:1807.05614.

[ClickHouse Authors, n.d.] [ClickHouse Authors, n.d.] ClickHouse Authors. *ClickHouse Performance Guide - Latency, P95/P99, Resource Consumption.* URL: https://clickhouse.com/docs/en/operations/performance (accessed: 22.12.2025).

[FAISS Team, n.d.] Facebook AI Similarity Search (FAISS) Team. *Faiss: A Library for Efficient Similarity Search.* GitHub repository. URL: https://github.com/facebookresearch/faiss (accessed: 22.12.2025).

[DuckDB Labs, n.d.] DuckDB Labs. *DuckDB Benchmarking - Exact Similarity Search Baseline.* URL: https://duckdb.org/docs/ (accessed: 22.12.2025).

[CodeXGLUE Authors, n.d.] CodeXGLUE Authors. *CodeXGLUE Benchmark Suite - Evaluation Metrics for Code Search and Code Similarity.* GitHub repository. URL: https://github.com/microsoft/CodeXGLUE (accessed: 22.12.2025).

[BigCloneBench, n.d.] BigCloneBench datasets. GitHub repository. URL: https://github.com/clonebench/BigCloneBench (accessed: 22.12.2025).

Authors' Information

Olena Chebanyuk, D.Sc. in Engineering, Full Professor.

Contract Researcher at the Artificial Intelligence Research Institute of the Spanish National Research Council,

e-mail: elena.chebanyuk@iiia.csic.es

Professor in the Software Engineering Department, State University "Kyiv Aviation Institute" (Ukraine).

Major Fields of Scientific Research: Application of Artificial Intelligence to Software Engineering Tasks, Software Engineering Fundamentals, Domain Engineering, Model-Driven Development, Fundamentals of Code Reuse in the AGILE Approach.

Oleh Osadchenko – "State University" "Kyiv Aviation Institute". Faculty of Computer Science and Technology. Department of Software Engineering. Kyiv, Ukraine;

e-mail: oleg.osadchenko.v@gmail.com

Major Fields of Scientific Research: Automated source-code plagiarism detection; semantic code similarity (transformer-based embeddings, CodeBERT/GraphCodeBERT); approximate nearest neighbor search (HNSW) and large-scale retrieval; code normalization, shingling and boilerplate filtering; evaluation of retrieval quality (Precision@k, Recall@k, PR curves); software engineering of scalable ML-based systems (.NET, Python, ClickHouse).