Semantic Fingerprinting of Software Modules Through Task Description Analysis*

Oleh Andriichuk[1,2,*,†], Stanislav Zholobetskyi[1,†]

[1] Institute for Information Recording of the National Academy of Sciences of Ukraine, Shpak str. 2, Kyiv, 03113, Ukraine
[2] Taras Shevchenko National University of Kyiv, Volodymyrs'ka str. 64/13, Kyiv, 01601, Ukraine

Abstract

This article investigates whether semantic information from task descriptions in issue tracking systems can help identify relevant software modules. While modern code navigation tools build embeddings from source code, task descriptions capture different information: developer intent and business logic rather than implementation details. A method is proposed that creates "semantic fingerprints" for software modules by aggregating vector embeddings of historical task titles linked through Git-Jira integration. The method is evaluated through 90 experimental configurations across two large open-source projects (Apache Flink and SonarQube). Nine embedding models across three generations are compared - classical word embeddings (Word2Vec, FastText, GloVe), general-purpose transformers (BERT, RoBERTa, CodeBERT), and sentence transformers (MPNet, BGE, MS MARCO) - combined with five aggregation strategies. Sentence transformers significantly outperform earlier approaches, with BGE achieving 116% improvement in Mean Average Precision over Word2Vec baseline (0.3662 vs 0.1695). Code-specialized CodeBERT underperforms general-purpose BERT, suggesting that natural language understanding matters more than code syntax knowledge for processing task descriptions. Results generalize across both projects. The research reveals that aggregation strategy has negligible impact for high-quality embeddings: all five strategies produce identical results at four decimal places for pretrained models, while custom Word2Vec shows 8-10% variation. This phenomenon is explained through vector space geometry - pretrained models create tightly-clustered embeddings where different aggregation methods become mathematically equivalent after normalization. The best model achieves Recall@10 of 0.68 and MRR of 0.43, meaning the first relevant module typically appears around rank 2-3. However, the method shows a bias toward frequently modified modules, making it more suitable as a complementary filter that narrows the search space rather than a primary navigation system. Task-based semantic fingerprints capture an aspect of software structure that code analysis alone does not provide, and combining them with code-based approaches may offer more complete support for developers navigating large software systems.

Keywords

mining software repositories, semantic analysis, task descriptions, embedding models, module localization, word2vec, transformers, sentence transformers, vector aggregation, semantic fingerprints

1. Introduction

Modern software development uses two main systems: version control (like Git) for code changes, and issue tracking (like Jira) for task descriptions. Developers connect these systems by including task IDs in commit messages, creating links between "what was changed" and "why it was changed".

Recent code navigation systems use Retrieval-Augmented Generation (RAG) with vector embeddings of source code. These systems use models like CodeBERT to create embeddings that capture code syntax. However, source code mainly contains technical implementation details - variable names, function calls, and class structures show how something works, but not always why it exists or what business problem it solves.

*ITS-2025: Information Technologies and Security, December 11, 2025, Kyiv, Ukraine
* Corresponding author.
† These authors contributed equally.
andriichuk@ipri.kiev.ua (O. Andriichuk); s.zholobetsky@gmail.com (S. Zholobetskyi)
0000-0003-2569-2026 (O. Andriichuk); 0009-0008-6058-7233 (S. Zholobetskyi)
© 2025 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-11-26
CEUR-WS.org/Vol-4337/paper24.pdf

Task descriptions from issue tracking systems provide different information. They explain the intent behind code changes, describing problems and requirements in natural language closer to how developers think. For example you will not find “poor performance”, "unpredictable behavior" or “rounding issue” in the programme code. But you will find them on the complete Jira tickets and can understand where it was in the code if you look at the files changed with those tickets.

This leads to our research question: can we use task descriptions to create semantic representations of software modules? If we collect all task descriptions that historically changed a module, do these descriptions capture useful semantic information about what the module does?

We build vector embeddings for task descriptions and aggregate them by module. For each module, we create a "semantic fingerprint" by averaging vectors of all tasks that changed files in that module. We then test whether these fingerprints help identify relevant modules for new tasks.

We do not claim task-based embeddings replace code-based embeddings. Instead, we investigate whether they provide complementary information that reflects developer intent and business logic rather than implementation details.

Research Questions:

RQ1: How well do different embedding models capture semantic information from task descriptions? We compare nine approaches: classical word embeddings (Word2Vec, FastText, GloVe), general-purpose transformers (BERT, RoBERTa, CodeBERT), and embedding-specialized transformers (MPNet, BGE, MS MARCO).

RQ2: How do aggregation strategies affect module fingerprints? We test five strategies: averaging, sum, median, weighted average, and clustering.

RQ3: Can task-based semantic fingerprints identify relevant modules for new tasks? We use train/test splits and measure performance with MAP, MRR, and Recall@K.

2. Related Work

Our work builds on mining software repositories, semantic analysis of software artifacts, and modern embedding approaches.

2.1. Mining Software Repositories and Change Coupling

The idea of learning from software history started with Gall et al. [1], who introduced logical coupling. They observed that files often change together even without direct code dependencies. By analyzing release history, they could predict which files might need changes together. This opened a new research direction in understanding software evolution [2, 3].

Later work by Fluri, Pinzger and Gall [4] extended this to fine-grained analysis, analyzing individual commits rather than releases. This gave more precise information about related changes, treating commits as atomic units of change.

Canfora and Cerulo [5] connected version control data with issue tracking systems, showing that linking Git commits to Jira tasks provides richer context. Instead of just knowing files changed together, we can understand why they changed.

Our work follows this direction but focuses on the semantic content of task descriptions rather than co-change patterns.

2.2. Conceptual Coupling and Information Retrieval

Beyond change patterns, researchers explored semantic similarity between code artifacts. Kagdi et al. [6] introduced conceptual coupling - files sharing similar vocabulary are related even if they never change together. They used information retrieval techniques with Vector Space Model (VSM) and TF-IDF weighting to analyze identifiers and comments in source code.

Research showed that combining logical coupling (history-based) with conceptual coupling (text-based) gives better results than either alone. The approaches provide complementary information: history shows what changed together, while text analysis shows what should logically be related.

However, conceptual coupling depends on code quality. Identifiers can be cryptic, comments are often missing or outdated. This motivated looking beyond code text to other sources of semantic information.

2.3. Bug Localization and Modern RAG Approaches

Bug localization finds which source files need changes to fix reported bugs. There are three main approaches now. Early approaches used information retrieval, comparing bug report text with source code text. After that researchers collect and publish big datasets with bugs in Java and python programmes and direct location. After that they trained the different neural network classifiers and tried to predict/classify the location of the bug by their text. This approach can be evaluated with specialized benchmarks [11]. More sophisticated systems combined textual similarity with historical information and multiple data sources. Recent research explored fine-grained localization at method or statement level [12]. Modern systems use neural networks and large language models, often with RAG (Retrieval-Augmented Generation) architecture. These systems build embeddings for code files and use vector similarity for retrieval.

Each of these approaches show good results but they still depend on data quality. Recent challenges include tangled commits [7] where multiple unrelated changes mix together. Research addresses untangling commits [8] and tools like SEAL [9] that integrate program analysis with repository mining. We also understand the impact of quick remedy commits [10]. We expect our method won't be affected by these outliers since we average the module's direction based on hundreds of tasks, so this noise won't change the module's general direction.

Our work differs in two ways. First, we work at a module level that represents the business area. We search the direction for further search rather than the final location. Second, we do not analyze code text at all. We build semantic representations entirely from task descriptions, capturing intent and business logic rather than implementation patterns.

2.4. Word Embeddings and Language Models

The rise of word embeddings changed text representation. Methods like Word2Vec, GloVe, and FastText represent words as dense vectors, capturing semantic relationships. The next generation came with general-purpose transformer models like BERT, showing that context matters. For code specifically, models like CodeBERT were trained on large code corpora.

More recently, embedding-specialized (sentence transformers) models showed impressive capabilities. Models like MPNet, BGE, and MS MARCO are specifically designed for semantic search and retrieval tasks.

Our work compares these different generations of embedding models. We test whether advances in language models translate to better semantic representations of task descriptions. Importantly, we apply these models to task text, not code.

2.5. Positioning Our Work

Our work combines ideas from several areas but creates a novel combination. Like MSR research [2], we exploit historical data from version control and issue tracking. Like conceptual coupling [5], we analyze semantic similarity, but we analyze task descriptions rather than code text. Like bug localization [11], we predict relevant locations, but we work at module level, that is rather direction than a location. Like modern RAG systems [12], we use vector similarity for retrieval, but our vectors represent modules through their historical tasks.

The key distinction is our focus on task semantics. Most research analyzes code to understand software. We analyze what developers write about code - the problems they describe, the features they request. This captures a different aspect of software systems: their purpose and evolution from a human perspective.

We do not claim this replaces code analysis. Instead, we investigate whether task-based semantics provide complementary information.

3. Methodology

Our approach transforms repository data into semantic module fingerprints. We collect data from Git and Jira, create vector representations of tasks, aggregate them into module fingerprints, and use these to predict relevant modules for new tasks (see Figure 1).

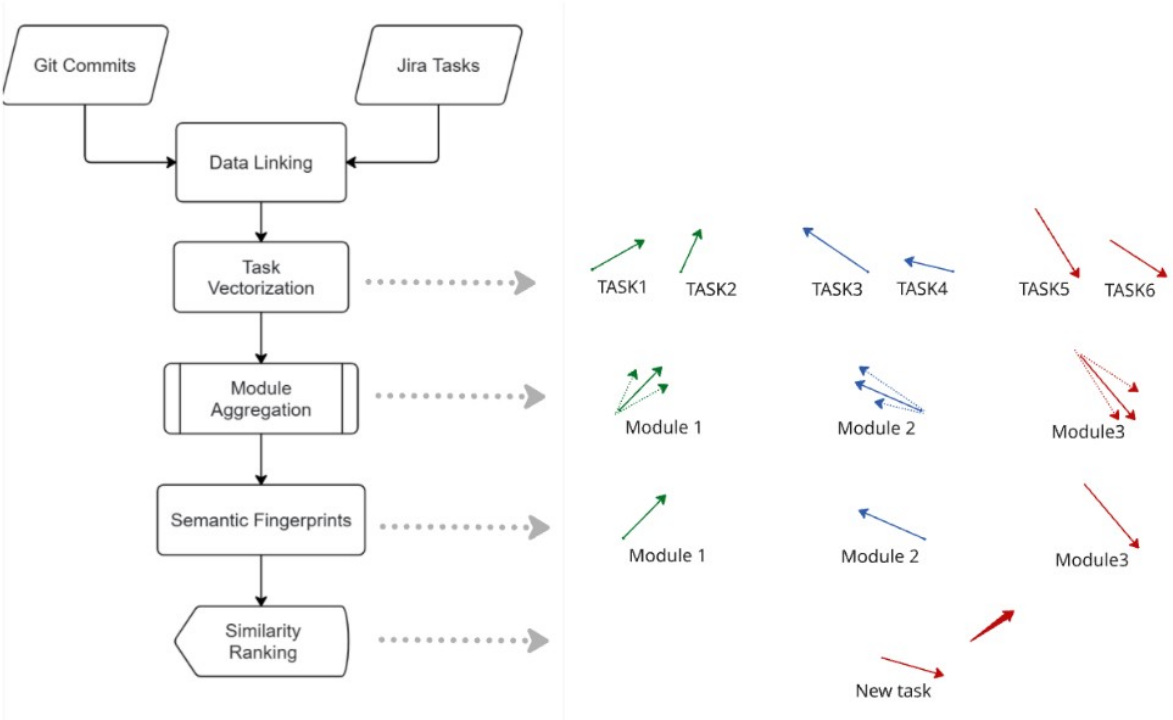


Figure 1: Dataflow from tasks to modules

3.1. Overview

To perform this prediction, we utilized the Semantic Fingerprints method. Thus, we combined and mapped information about the commit changes with information about the tasks that necessitated those changes. For each task, we found its vector representation (embedding). We then grouped the tasks by the modules they modified and found aggregated vectors that represent the modules. We saved these aggregated vectors as the module's semantic fingerprints. For new tasks, we perform the same vectorization as before, and now we search for the module whose vector is closest to the task vector by cosine distance. So the process has two phases. During training, we build semantic fingerprints for all modules using historical data. During testing, we convert new task descriptions to vectors and compare them with module fingerprints to predict relevant modules.

3.2. Multimodal Data Linking

Projects use Git for version control and Jira for issue tracking. Developers connect these by including Jira task IDs in commit messages (e.g., "FLINK-1234: Fix memory leak").

We extract this link from the commit message using pattern matching like "PROJECT-NUMBER". The culture of documentation in Apache projects is exceptionally high, ensuring that all commits consistently follow a unified pattern. The Sonar project presents a different picture, where various styles coexist, and only about 60% of commits contain valid task references.

3.3. Module Structure Extraction

A module is the first-level directory in the project. For example, flink-runtime/src/main/java/network/Buffer.java belongs to module flink-runtime. Each task has a list of files that were changed. In this way, we can associate each task with a module and vice versa - module with a list of tasks that have historically modified it. Files without clear module prefixes are ignored.

3.4. Task Vectorization

Task title, description, and comments represent text with different density of meaning (or semantic density). In previous experiments, the inclusion of comments diluted the semantic areas, which is why we have currently focused only on the title. In future experiments, we plan to study the influence of both the description and comments on the system's ability to define modules.

We test nine embedding models in three generations:

Classical Word Embeddings:

- Word2Vec: Custom trained on task titles corpus using CBOW algorithm (window size 50, 10 epochs) - baseline
- FastText: Pretrained English model (300-dimensional, resized to 100)
- GloVe: Pretrained 6B.100d model

General-purpose Transformers:

- BERT-base: bert-base-uncased using (CLS) token from second-to-last layer
- RoBERTa-base: roberta-base with improved training
- CodeBERT: microsoft/codebert-base trained on code and natural language

Sentence Transformers:

- MPNet: sentence-transformers/all-mpnet-base-v2 for semantic search
- BGE-large: BAAI/bge-large-en-v1.5 (1024-dimensional, resized to 100)
- MS MARCO: sentence-transformers/msmarco-distilbert-base-v4 for passage ranking

3.5. Module Fingerprinting

For module M, we collect all tasks T_1, T_2, ..., T_n that changed files in M. Each task has vector v_1, v_2, ..., v_n. We test five aggregation strategies:

1. Average (avg) (or mean vector) - is calculated as the arithmetic average of the set of vectors, defined as the sum of the vectors divided by their total count (1). Then normalize to unit length.

$$v_{avg} = \frac{v_1 + v_2 + v_3 + \ .. \ + v_n}{n}. \quad (1)$$

2. Sum (or aggregation vector) - we first compute the vector sum of the input embeddings (2), the resulting vector is then normalized to unit length

$$v_{\text{sum}} = v_1 + v_2 + v_3 + \ .. \ + v_n. \quad (2)$$

3. Median (component-wise or marginal median) - For a set of n vectors in d-dimensional space, the component-wise median is a vector where each j-th coordinate is the median of the j-th coordinates of all input vectors (3)

$$v_{median}[i] = median(v_1[i], v_2[i], v_3[i], \ ..., \ v_n[i]) \quad (3)$$

4. Weighted Average. Let v_i be the embedding of the i-th task, and w_i be the number of files in the target module modified by that task. The weighted average $v_{weighted_avg}$ calcualted by formula (4) and by using the number of affected files as a weight, we are saying that a task is more "representative" of a module's semantics if it has a larger impact footprint on that module.

$$v_{weighted_avg} = \frac{w_1 v_1 + w_2 v_2 + w_3 v_3 + \ .. \ + \ w_n v_n}{w_1 + w_2 + w_3 + \ .. \ + \ w_n} \quad (4)$$

5. Cluster (cluster) - In K-means, the centroid is the "ideal" member of the group, so we calulcated it with (5) using K-means with K=1.

$$v_{cluster} = centroid \ \ of \ \ \{v_1, v_2, v_3, \ .. \ , \ v_n\} \quad (5)$$

After aggregation, like in [13, 14], all module vectors are normalized to unit length.

3.6. Semantic Compass as Similarity Ranking

For every new test tasks we convert task description to vector using the same embedding model. After that we normalize it to unit length. Calculate cosine similarity with each module fingerprint and in the end we rank modules by similarity (highest first), for normalized vectors we calculate similarity as dot product for task and module vectors (6)

$$similarity(task, \ module) = v_{task} \ \cdot \ v_{module} \quad (6)$$

3.7. Evaluation Methodology

Train/Test Split: We order tasks by creation date and take the 200 tasks as a test set. We select tasks when mod(task_id, 200) = 0. This split simulates real usage and allows us to test tasks from all time periods equally. For each test task, we extract which modules were actually changed from Git commits.

We comparing impact of each models and strategies by these evaluation metrics:

1. Mean Average Precision (MAP):
   Average Precision calculated for task with R relevant modules retrieved at ranks r_1, r_2, ..., r_r with (7)

$$AP = \frac{1}{R} \sum_{k=1}^{R} P@k \cdot rel@k \quad (7)$$

   where P@k - is standard precision@k, calculated at rank k (8)

$$P@k = \frac{\text{True Positives up to rank } k}{k} \quad (8)$$

   and rel@k - Relevance indicator function, it is 1 if the item at rank k is relevant.
   The Mean Average Precision is simply the average of the Average Precision scores across all classes(9)

$$MAP = \frac{1}{C} \sum_{i=1}^{C} AP_i \quad (9)$$

where C number of classes
   MAP ranges from 0 to 1, where 1 means perfect ranking.

2. Mean Reciprocal Rank (MRR):

For each task, find rank of first relevant module (10)

$$RR = \frac{1}{rank} \tag{10}$$

MRR is simply the average of the RR across all test tasks (11)

$$MRR = \frac{1}{C}\sum_{i=1}^{C} RR_i \tag{11}$$

3. Recall@K:
   Recall@K is number of relevant modules in top K divided by total relevant modules
   We report Recall@5, Recall@10, Recall@20, and Recall@50.
   It means how many true positive modules appear in top5, top10, top20, top50 rating builded with our model.

4. Experimental Data

We selected two large open-source projects with active development and good Git-Jira integration: Apache Flink (distributed stream processing) and SonarSource SonarQube (code quality inspection platform). All commits of this projects have been downloaded and stored as separate objects in the database. Then we have extracted the task name from the commit messages. By the task name we have downloaded the task title, description and comments from the Jira and also stored it in the database. See statistics of this projects in Table 1.

Table 1
Project Statistics

Metric	Apache Flink	SonarQube
Downloaded Git commits	372,550	469,836
Commits with task refs	372,550 (100%)	248,628 (53%)
Downloaded tasks	17,073	9,799
Tasks with title	17,038 (99.8%)	9,272(94.6%)
Number of modules	82	63
Training tasks	16,838	9,072
Test tasks	200	200
Development period	~15 years	~15 years

Distribution follows a typical pattern: few modules concentrate most activity, many have moderate activity. Table 2 shows top 5 modules of two projects by task count.

Most tasks have short titles about 7 words long, as shown in Table 3 and Figure2. But there are very small numbers of words with high TF-IDF value. About two thousands tasks only can be recognised with more or less unique words for the module. Usually it is general words that can not be used to predict the relevant module with simple classification methods.

Table 2
Top 5 Modules by Task Count

Flink Modules	Tasks	SonarQube Modules	Tasks
flink-table	48,730	server	138,542
flink-runtime	45,812	sonar-server	24,683
flink-connectors	14,022	sonar-core	11,608
flink-streaming-java	13,449	plugins	10,424
flink-libraries	12,614	sonar-batch	7,942

Table 3
Task Title Length Statistics

Statistic	Flink	SonarQube
Minimum words	1	1
Average words	7	7
Median words	7	7
Maximum words	25	24

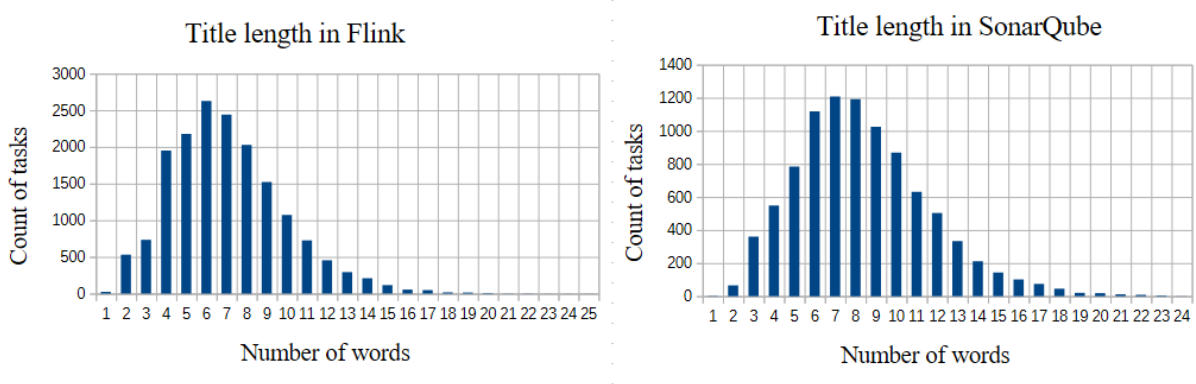


Figure 2: Length of task title per project

4.1. Data Preprocessing

Commit parsing: Extract task IDs from commit messages using patterns with brackets like (FLINK-123) or without brackets like SONAR-123. Skip commits without task references. Skip tasks that are not accessible in the active Jira system.

Module extraction: Use the first directory component (e.g., flink-runtime/src/... → flink-runtime). Exclude build files, documentation, and root-level files.

Task text: Use title only. In the title we usually have plain text only so we do not need to remove HTML tags or numbers. Lowercase for classical embeddings; original case for transformers. No stop word removal or stemming.

Multi-module tasks: A task touching N modules contributes to all N fingerprints. During evaluation, prediction is correct if it matches any true module.

5. Results and Analysis

In Table 4 was presented results from 45 experiments on each project (9 models × 5 aggregation strategies). All results use the same train/test split with 200 test tasks.

Table 4
Model Performance Comparison (averaged across all aggregation strategies)

Rank	Model	Flink MAP	Sonar MAP	Avg MAP
1	BGE	0.3611	0.3713	0.3662
2	MPNet	0.3025	0.3424	0.3225
3	MS MARCO	0.3043	0.2755	0.2899
4	GloVe	0.2431	0.3307	0.2869
5	FastText	0.2533	0.3100	0.2817
6	BERT	0.2207	0.3126	0.2666
7	RoBERTa	0.2027	0.2604	0.2316
8	CodeBERT	0.1859	0.2430	0.2144
9	Word2Vec	0.1691	0.1698	0.1695

5.1. Overall Model Performance

The experiments yielded a number of key findings. In the following statement they will considered sequentially.

Finding 1: Modern Sentence transformers significantly outperform word2vec and transformers methods. BGE achieves MAP of 0.3662 compared to Word2Vec's 0.1695 - a 116% improvement. This pattern is consistent across both projects.

Finding 2: Sentence transformers embeddings (BGE, MPNet, MS MARCO) dominate top positions. These models were specifically trained for semantic similarity tasks, and this specialization shows clear benefits.

Finding 3: CodeBERT does not outperform general-purpose BERT. Despite being trained on code, CodeBERT performs worse (MAP=0.2144) than BERT (MAP=0.2666). This suggests that understanding natural language in task descriptions is more important than understanding code syntax.

Finding 4: Results are consistent across projects. Model rankings are similar for both Flink and SonarQube, suggesting these findings generalize across different software domains.

5.2. Detailed Metrics for Best Model

As shown in Table 5, BGE shows strong performance across all metrics. Recall@10 of 0.62-0.76 means that in the top 10 predictions, we capture 62-76% of all relevant modules. Recall@50 near 1.0 indicates that almost all relevant modules appear in the top 50 predictions.

Table 5
BGE-large Performance - All Metrics

Project	MAP	MRR	Recall@5	Recall@10	Recall@20	Recall@50
Flink	0.3611	0.4306	0.4662	0.6180	0.7966	0.9842
Sonar	0.3713	0.4049	0.5758	0.7593	0.9550	1.0000

5.3. Aggregation Strategy Analysis

Finding 5: Aggregation strategy has minimal impact. As shown in Table 6, all five strategies produce nearly identical results (differences < 1%). This is especially clear for the best model (BGE), where all strategies yield exactly MAP=0.3611 for Flink and MAP=0.3713 for SonarQube when rounded to four decimal places.

Table 6
Performance by Aggregation Strategy (averaged across all models)

Strategy	Flink MAP	Sonar MAP	Average
median	0.2496	0.2907	0.2702
avg	0.2496	0.2907	0.2702
cluster	0.2494	0.2897	0.2696
weighted_avg	0.2494	0.2915	0.2705
sum	0.2480	0.2905	0.2693

This counterintuitive finding requires explanation. During development, we suspected a software bug was causing all strategies to produce identical results. Detailed debugging revealed that differences do exist, but their magnitude varies dramatically by model quality.
Strategy Impact on MAP Performance

Table 7
Word2Vec (custom trained) - shows measurable variation:

Strategy	Flink MAP	Sonar MAP
median	0.1726	0.1702
avg	0.1724	0.1706
cluster	0.1712	0.1617
weighted_avg	0.1706	0.1779
sum	0.1588	0.1686
Range:	0.0138	0.0162

Table 8
FastText (pretrained) - identical across strategies:

Strategy	Flink MAP	Sonar MAP
median	0.2533	0.3100
avg	0.2533	0.3100
cluster	0.2533	0.3100
weighted_avg	0.2533	0.3100
sum	0.2533	0.3100
Range:	0.0000	0.0000

Table 9
BGE (large-en-v1.5) - identical across strategies:

Strategy	Flink MAP	Sonar MAP
median	0.3611	0.3713
avg	0.3611	0.3713
cluster	0.3611	0.3713
weighted_avg	0.3611	0.3713
sum	0.3611	0.3713
Range:	0.0000	0.0000

Word2Vec shows variation of 0.0138-0.0162 in MAP (approximately 8-10% relative variation), while FastText and BGE show zero variation at 4 decimal places. Differences for pretrained models exist only beyond the 10th decimal place.

This phenomenon has a clear mathematical explanation:

Vector normalization eliminates magnitude differences but not directional differences. Our implementation normalizes all module vectors to unit length after aggregation, regardless of strategy. This is critical: normalization transforms vectors to lie on the unit hypersphere, where only direction matters, not magnitude.

For sum strategy we normalize the sum of vectors to unit length. For the average strategy, we likewise normalize the averaged vector to unit length. When all input vectors (v_1, v_2, ..., v_n) point in very similar directions (low angular variance), the sum and average point in nearly identical directions. After normalization, they produce the same unit vector. However, when input vectors have higher angular variance (point in somewhat different directions), sum and average can point in slightly different directions, producing different results even after normalization.

That why Word2Vec shows measurable differences. Custom-trained Word2Vec on our limited corpus (~12,000-15,000 tasks title) produces embeddings where semantically similar tasks still have significant cosine distance between their vectors.

For example, in our Word2Vec model, highly related word pairs show surprisingly low

similarity. For example "api" and "apis" have cosine similarity ≈ 0.60, "user" and "users" even worse cosine similarity ≈ 0.38. This means their vectors point in noticeably different directions with big angular separation. Two tasks both about "memory management" might use slightly different terminology ("buffer allocation" vs "memory pool") and end up with vectors that have small cosine similarity.

When aggregating such scattered vectors, different strategies produce noticeably different directional results. When the mean pulls toward the average direction across all task vectors, the median is less affected by outlier directions, staying closer to the central tendency, at the same time the sum can be pulled more strongly by the combined effect of multiple similar vectors and weighted average emphasizes tasks that changed many files.

These directional differences persist after normalization, explaining why Word2Vec shows 8-10% variation in MAP across strategies (range of 0.0138-0.0162 as shown in Table 7).

Sentence transformers show no measurable differences because they were trained on billions of words and produce high-quality embeddings where semantically similar tasks have very small cosine distance. Word pairs like "api"/"apis" or "user"/"users" have very high cosine similarity in these models. Two tasks about "memory management" typically have cosine similarity > 0.95, meaning their vectors point in nearly identical directions.

When aggregating such tightly-clustered vectors, different strategies produce nearly parallel directional results that become indistinguishable after normalization. As Table 7 shows, FastText and BGE produce identical results across all five strategies at 4 decimal places. The differences exist mathematically but appear only at the 10th decimal place (FastText) or 15th decimal place (BGE) - far beyond the precision relevant for ranking.

The fundamental difference is embedding quality, not aggregation method. High-quality embeddings naturally organize semantically related tasks into tight clusters with minimal angular variance. In such well-organized vector spaces, the aggregation method becomes irrelevant because all reasonable aggregation strategies (mean, median, weighted mean) produce nearly identical directional results. The quality of the embedding space determines the robustness to methodological choices.

As a result developers can use the simplest strategy (arithmetic mean) without concern about lost performance. Complex strategies add computational cost without measurable benefit. But normalization is essential. Vector normalization is not just a technical detail, it fundamentally determines the behavior of aggregation strategies. Any implementation must normalize module vectors before computing similarity. After the normalization the embedding quality matters most. The critical factor is using high-quality pretrained embeddings. Once embeddings create tight semantic clusters, aggregation strategy becomes irrelevant. This finding simplifies practical deployment while providing theoretical insight into how semantic information is encoded in task embeddings and why embedding quality is the dominant factor in system performance.

5.4. Comparison with Baseline

The improvement is substantial across all metrics. BGE more than doubles the prediction accuracy compared to Word2Vec baseline (see Table 10).

Table 10
Best (BGE) vs Baseline (Word2Vec) Comparison

Metric	Word2Vec	BGE	Improvement
Flink MAP	0.1691	0.3611	+113.5%
Flink MRR	0.1885	0.4306	+128.5%
Sonar MAP	0.1698	0.3713	+118.7%
Sonar MRR	0.1868	0.4049	+116.8%

5.5. Model Generation Analysis

Grouping models by generation reveals clear progression (see Figure 3):

1. Classical embeddings models (Word2Vec, FastText, GloVe): Average MAP = 0.2127
2. General-purpose transformer models (BERT, RoBERTa, CodeBERT): Average MAP = 0.2375
3. Sentence transformer models (MPNet, BGE, MS MARCO): Average MAP = 0.3262

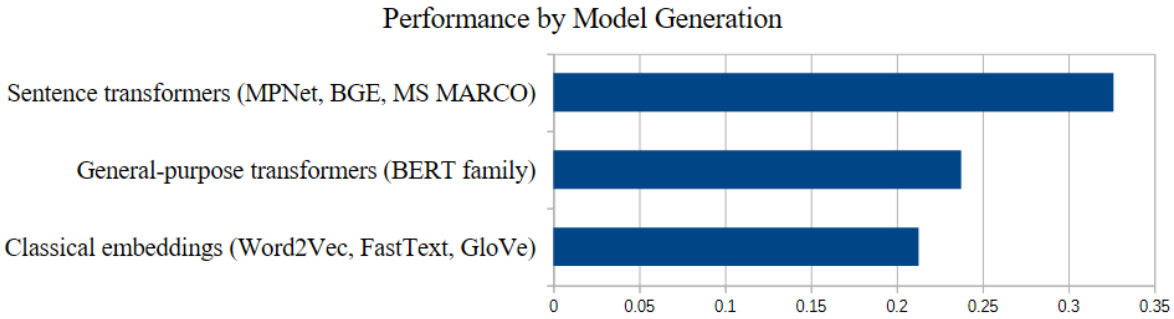


Figure 3: Compare performance by models generation

Each generation improves upon the previous, the sentence transformers showing the largest gain (+37% over BERT transformers, +53% over classical methods).

5.6. Cross-Project Consistency

To measure consistency, we calculate the correlation between Flink and SonarQube MAP scores across all models. The Pearson correlation coefficient is r=0.79, indicating strong consistency. Models that perform well on Flink tend to perform well on SonarQube, suggesting that our findings are not project-specific.

Notable exception: GloVe performs better on SonarQube (MAP=0.3307) than on Flink (MAP=0.2431). This may reflect differences in task description vocabulary between the two projects.

5.7. Practical Performance Interpretation

To understand practical implications, consider what these metrics mean for developers. So when we have MAP=0.36 (BGE) it means that usually, the first relevant module appears around rank 2-3 and the second relevant module appears around rank 5-6. MRR=0.43 (BGE) means the first correct prediction typically appears at rank 2.3. Recall@10=0.68 (BGE average) means that If a developer looks at the top 10 suggested modules, they will find 68% of all relevant modules. For tasks touching 2-3 modules, this means finding most of them in a short list. When we see Recall@50=0.99 (BGE average) this parameter says that almost all relevant modules appear in the top 50 predictions but not all. This method is good enough to recommend popular modules and exotic modules may never appear in the list only when the user will request very specific words for that module. We can see this method can not be used as a main navigation system because it proposes rather popular changes than the correct one, but it can be used as additional filter criteria. These findings demonstrate that task-based semantic fingerprints capture meaningful information about module functionality and can complement code-based approaches in developer assistance tools.

6. Limitations

Project scope: We evaluated only two large Java open-source projects with English documentation. Results may differ for smaller projects, other languages, or non-English descriptions. Our approach

is sensitive to data quality. It needs good Jira-Git integration and detailed task descriptions. Projects with poor linking or brief descriptions would perform worse.

We work at module level (first-level directories). Finer granularity (files, methods) might show different patterns and require different approaches.

We measured prediction accuracy, not developer productivity. Real-world benefits require deployment studies to measure time savings and user satisfaction. We did not test integration with IDEs or development tools. Practical deployment would require UI design and workflow integration.

7. Conclusion

We investigated whether semantic information from task descriptions can identify relevant software modules. We created "semantic fingerprints" for modules by aggregating vector embeddings of historical task titles, then used these to predict relevant modules for new tasks. We conducted 90 experimental configurations (9 embedding models × 5 aggregation strategies) across two large open-source projects: Apache Flink (16,838 training tasks) and SonarQube (9,072 training tasks).

RQ1: How well do different embedding models capture semantic information from task descriptions? Sentence transformer models capture task semantics significantly better than both classical word embeddings and general-purpose transformers. BGE achieves 116% improvement in MAP over Word2Vec baseline (0.3662 vs 0.1695). Each model generation shows clear progression: classical embeddings (average MAP = 0.2127), general-purpose transformers (0.2375), and sentence transformers (0.3262). Notably, code-specialized CodeBERT (MAP = 0.2144) underperforms general-purpose BERT (MAP = 0.2666), indicating that natural language understanding matters more than code syntax knowledge when processing task descriptions. Results are consistent across both projects.

RQ2: How do aggregation strategies affect module fingerprints? Aggregation strategy has negligible impact on performance for high-quality embeddings. All five strategies (average, sum, median, weighted average, cluster) produce identical results at four decimal places for pretrained models, while custom Word2Vec shows 8-10% variation. We explain this through vector space geometry: pretrained models create tightly-clustered embeddings where different aggregation methods become mathematically equivalent after normalization. This finding has practical value - developers can use the simplest strategy (arithmetic mean) without any loss in performance.

RQ3: Can task-based semantic fingerprints identify relevant modules for new tasks? Yes, but with important limitations. Our best model achieves Recall@10 of 0.68, meaning developers find 68% of relevant modules in the top 10 predictions, and MRR of 0.43, meaning the first correct module typically appears around rank 2-3. However, the method has a bias toward frequently modified modules - it tends to suggest popular modules rather than precisely correct ones. Less common modules may not appear in predictions unless the task description contains vocabulary strongly associated with them. Therefore, semantic fingerprints are not suitable as a primary navigation system but can serve as a complementary filter that narrows the search space based on developer intent and business logic, supplementing code-based approaches.

Our work demonstrates that task descriptions contain meaningful semantic information about software modules. Modern NLP techniques can extract this information effectively, and the resulting fingerprints capture an aspect of software structure that code analysis alone does not provide - the perspective of developer intent and historical usage. We do not claim this replaces code analysis or traditional tools. Rather, combining task-based semantics with code-based approaches may provide more complete support for developers navigating large software systems. Further research is needed to evaluate this integration and measure actual impact on developer productivity.

Declaration on Generative AI

During the preparation of this work, the authors used Gemini in order to translate to English, grammar and spelling check. After using these tools, the authors reviewed and edited the content as needed and takes full responsibility for the publication's content.

References

[1] H. Gall, K. Hajek, M. Jazayeri. Detection of Logical Coupling Based on Product Release History. In Proceedings of the International Conference on Software Maintenance. ICSM '98. 1998. https://doi.org/10.5555/850947.853338

[2] Mens, Tom, et al. "Analysing software repositories to understand software evolution." Software evolution 2008. https://doi.org/10.1007/978-3-540-76440-3_3

[3] Kagdi, Huzefa H. Mining software repositories to support software evolution. Diss. Kent State University, 2008. https://doi.org/10.1002/smr.344

[4] Gall, B. Fluri, and M. Pinzger. Change analysis with evolizer and changedistiller. IEEE Software 26(1): 2009. https://doi.org/10.1109/MS.2009.6

[5] Canfora, Gerardo, and Luigi Cerulo. "Impact analysis by mining software and change request repositories." 11th IEEE International Software Metrics Symposium. IEEE, 2015. https://doi.org/10.1109/METRICS.2005.28

[6] Kagdi, Huzefa, Malcom Gethers, and Denys Poshyvanyk. "Integrating conceptual and logical couplings for change impact analysis in software." Empirical Software Engineering 18 2013. https://doi.org/10.1007/s10664-012-9233-9

[7] Herzig, K., & Zeller, A. The impact of tangled code changes. In 2013 10th Working Conference on Mining Software Repositories (MSR). IEEE 2013. https://doi.org/10.5555/2487085

[8] Li, Y., Wang, S., & Nguyen, T. N. (2022). UTANGO: untangling commits with context-aware, graph-based, code change clustering learning model. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. https://doi.org/10.1145/3540250.3549171

[9] Sattler, Florian, et al. Seal Integrating program analysis and repository mining. ACM Transactions on Software Engineering and Methodology 32.5 2023. https://doi.org/10.1145/358500

[10] Wen, Fengcai, et al. Quick remedy commits and their impact on mining software repositories. Empirical Software Engineering 27 2022. https://doi.org/10.1007/s10664-021-10051-z

[11] Paul Kassianik, Erik Nijkamp, Bo Pang, Yingbo Zhou, Caiming Xiong. BigIssue: A Realistic Bug Localization Benchmark. https://doi.org/10.48550/arXiv.2207.10739

[12] Qianru Meng, Xiao Zhang, Zhaochen Ren, Joost Visser. Code Review Automation using Retrieval Augmented Generation. https://doi.org/10.48550/arXiv.2511.05302

[13] Kadenko, S., Tsyganok, V., Szádoczki, Z., and Bozóki, S. (2021) An update on combinatorial method for aggregation of expert judgments in AHP. Production, 31. https://doi.org/10.1590/0103-6513.20210045

[14] Andriichuk, O., Tsyganok, V., Kadenko, S., & Porplenko , Y. (2020) Experimental Research of Impact of Order of Pairwise Alternative Comparisons upon Credibility of Expert Session Results Proceedings of the 2nd IEEE International Conference on System Analysis & Intelligent Computing, (SAIC), 05-09 October, 2020, Kyiv, Ukraine, pp. 1-5. https://doi.org/10.1109/SAIC51296.2020.9239126