Thesis Dissertation

METHOD MAPPING IN CROSS-OVER ECOSYSTEM PACKAGES

Stefanos Efstathiou

University Of Cyprus

COMPUTER SCIENCE DEPARTMENT

May 2026

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

Method Mapping in Cross-Ecosystem Packages

Stefanos Efstathiou

Supervisor

Eleni Constantinou

Thesis submitted in partial fulfillment of the requirements for the award of degree of bachelor's in computer science at University of Cyprus

May 2026

I declare that this dissertation and any accompanying software are my own original work, conducted in full compliance with the University of Cyprus Code of Conduct for Research (https://www.ucy.ac.cy/legislation/kodikas-deontologias-kai-politikes/), and with full accountability on my part for the accuracy, integrity, and validity of all content.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Eleni Constantinou for her continuous guidance, support, and valuable feedback throughout the development of this thesis. Her mentorship was extremely important in helping me complete this work.

I would like also to thank my family and friends for their constant support, patience, and encouragement during my studies. Their help and motivation gave me strength to continue and successfully complete this thesis.

Abstract

This thesis focuses on the methodology of method mapping across software packages in different programming ecosystems, aiming to analyze their functional similarity. In particular, the study investigates the mapping of methods between Python and JavaScript packages by leveraging natural language processing and semantic embeddings.

The proposed approach includes the extraction of methods from software packages, their enrichment with semantic representations (embeddings), and the computation of similarity between them. Furthermore, large language models (LLMs) are utilized to evaluate the semantic relevance of the identified mappings.

The results demonstrate that the combination of embeddings and large language models can effectively identify meaningful correspondences between methods across different ecosystems even when they differ syntactically.

Contents

Section 1

Introduction

1.1 Introduction and Motivation

Modern software development relies heavily on reusable libraries and packages across multiple programming ecosystems. Modern applications are rarely implemented using single programming language or ecosystem. Instead, software systems often combine backend frameworks, frontend technologies, APIs, and cloud services. These ecosystems often provide similar functionalities through different implementations, naming conventions, and structural designs. As a result, identifying equivalent or similar methods across ecosystems has become a challenging but important problem in modern software engineering.

The need for cross-ecosystem understanding has grown significantly due to the increasing demand for interoperability, code migration, and knowledge transfer between programming languages. Developers frequently need to translate functionality from one ecosystem to another, which requires identifying corresponding methods that perform similar operations. However, this process is typically manual, time consuming, and error-prone, especially in large repositories where semantic equivalence may exist despite substantial syntactic and structural differences.

Traditional similarity detection techniques are often insufficient for this problem because semantically equivalent methods across ecosystems may use entirely different syntax, identifiers, frameworks, and implementation strategies. Methods that perform the same functionality may therefore appear structurally unrelated when analyzed using conventional

textual or syntactic comparison techniques. This limitation makes cross-language semantic mapping significantly more difficult than traditional mono-language clone detection.

Recent advances in machine learning and transformer-based language models have introduced new possibilities for semantic software analysis. Embedding models trained in source code can capture deeper semantic relationships between methods by transforming source code into numerical vector representations that preserve contextual and functional information. Additionally, LLMs have demonstrated strong reasoning capabilities in software engineering tasks such as code generation, summarization, translation, and semantic analysis. These technologies create new opportunities for identifying semantic correspondences between methods implemented in different programming languages.

This thesis is motivated by the need to automate and improve this process through method mapping techniques. By leveraging embeddings, cosine similarity, and LLMs, the proposed methodology aims to identify possible Type-IV semantic clones between Python and JavaScript methods. Unlike traditional approaches that rely mainly on syntactic similarity, this work focuses on semantic equivalence and functional similarity across ecosystems. The overall goal is to investigate whether modern AI-based approaches can effectively support semantic method mapping between programming languages.

1.2 Methodology Overview

This thesis proposes a methodology for mapping methods across software packages from different programming ecosystems, specifically focusing on Python and JavaScript. The approach begins with the extraction of methods from source code, capturing relevant information. These methods are then transformed into semantic representations using embeddings, which encode their meaning in a numerical vector space.

Next, similarity measures are computed between methods from different ecosystems, allowing the identification of potential correspondences. A ranking mechanism is applied to select the most relevant matches, typically focusing on the top candidates of each method.

To further enhance the evaluation process, large language models assess the semantic relevance of the identified method pairs. This step provides an additional layer of validation beyond purely mathematical similarity measures.

1.3 Thesis Structure

This thesis is organized into seven sections, each addressing a specific aspect of the research. Section 2 presents the necessary background by introducing fundamental concepts and definitions including software ecosystems, cross-ecosystems, method representation, embeddings and similarity measures, and LLMs.

Section 3 reviews the related work in the field, focusing on previous studies in code similarity, cross-language analysis, and application of machine learning techniques and embeddings in software engineering.

Section 4 describes the proposed methodology in detail, covering data collection, preprocessing, embedding generation, similarity computation, and evaluations using large language models.

Section 5 presents the experimental results and evaluates the effectiveness of the proposed method.

Section 6 provides a discussion of the findings including their implications, limitations, and threats to validity.

Finally, Section 7 concludes the thesis by summarizing its contributions and suggesting directions for future research.

Section 2

Background knowledge

2.1 Software Ecosystems

Software ecosystems represent interconnected environments consisting of programming languages, libraries, developers, and tools that evolve together. Modern software development heavily relies on such ecosystems with platforms like Python and JavaScript offering extensive repositories (e.g. PyPi and npm) that provide reusable components.

The increasing reliance on third-party libraries has led to highly interconnected ecosystems where similar functionalities are implemented across different programming languages. These ecosystems often overlap and interact, forming what is commonly referred to as cross-ecosystem development. Cross-ecosystems refer to scenarios where software components, libraries, or functionalities exist and are used across multiple programming ecosystems.

Recent studies highlight that developers frequently need to switch between ecosystems which requires identifying equivalent libraries or methods across languages [18]. in such cases, the same functionality may be implemented differently depending on ecosystem making direct comparison difficult. Additionally, many software systems are intentionally developed in multiple languages to support different platforms[21].

2.2 Type IV Code Clones (Semantic clones)

Code clone detection is well established research area, focusing on identifying similarities between code fragments. Clones are typically categorized into different types based on their level of similarity, ranging from exact copies to semantically equivalent implementations. [1]

Among these categories, Type IV clones, also known as semantic clones, represent the most complex and challenging form of code similarity. Type IV clones refer to code fragments that perform the same functionality but are implemented using different syntax, structures or algorithms. Unlike other clone types differences cannot be detected through simple textual or structural comparison.

The detection of Type IV clones is particularly important in modern software systems, especially in cross-language environments. When code is written in different programming languages, similar functionality is often implemented using entirely different constructs, libraries, and paradigms.

2.3 Semantic Embedings

Semantic embeddings play a central role in modern approaches to code analysis. They provide a way to represent textual or code–based information as numerical vectors in a high-dimensional space where semantic similarity corresponds to geometric proximity.

In software engineering, embeddings are widely used to represent code fragments, API methods, and documentation. These representations enable the comparison of code elements based on meaning rather than syntax.

2.4 Similarity Measures

Similarity measures are used to quantify the relationship between code elements based on their representations. In embedding-based approaches, similarity is typically computed between vector representations of methods.

One of the most widely used similarity metrics is cosine similarity, which measures the angle between two vectors in high dimensional space. This metric is particularly effective for comparing embeddings as it focuses on the direction of vectors rather than their magnitude. High coisine similarity values indicate that two methods are semantically similar.

2.5 Large Language Models (LLMs)

Large language models are a class of machine learning models based on deep neural networks, typically built using transformers architectures. They are trained in vast amounts of textual and code data to learn patterns, syntax, and semantic relationships in languages. As a result, LLMs can understand, generating and reasoning both natural language and source code.

In recent years, LLMs have become increasingly important in software engineering tasks. Their ability to interpret and process code enables applications such as code generation, documentation, bug detection, and code similarity analysis. Unlike traditional statistical approaches, LLMs can capture deeper semantic relationships by leveraging contextual information learned during training.

2.6 Evaluation Metrics

To evaluate the effectiveness of the proposed approach, several performance metrics commonly used in machine learning, information retrieval, and code clone detection were utilized. These metrics provide quantitative measurement of how accurately the methodology identifies semantically similar methods across programming ecosystems.

Before computing the evaluation metric, a Confusion Matrix was used ino order to organize and analyze the classification results produced by the methodology. A confusion matrix is a tabular representation commonly used in classification problems to summarize the number of correct and incorrect predictions generated by a system. Its visualization of how predictions are distributed across the four fundamental classification outcomes:

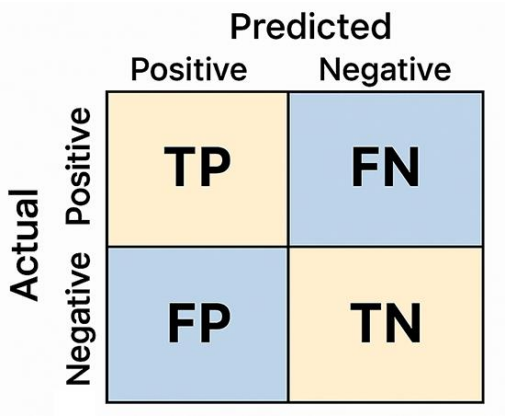

Figure 2.1 Confusion Matrix

The evaluation process is based on four fundamental classification outcomes:

- True Positive (TP)
- True Negative (TN)
- False Positive (FP)
- False Negative (FN)

A True Positive occurs when the methodology correctly identifies two methods as semantically equivalent. A True Negative occurs when two unrelated methods are correctly classified as non-matching. A False Positive occurs when the methodology incorrectly identifies two unrelated methods as semantically similar. Finally, a False Negative occurs when semantically equivalent methods are incorrectly rejected as non-matching. These four values form the basis for the computation of several evaluation metrics.

Accuracy measures the overall proportion of correct predictions produced by the methodology. It considers both correctly identified matches and correctly rejected non-matches:

Accuracy = (TP + TN) / (TP + TN + FP + FN)

Higher accuracy values indicate better overall classification performance

Precision measures how many of the predicted matches are correct:
Precision = TP / (TP + FP)
High precision indicates that the methodology produces fewer false positive mappings.

Recall, also known as sensitivity measures the ability of the methodology to identify all relevant semantic mappings:
Recall = TP / (TP+FN)
High recall indicates that fewer correct mappings are missed.

Specificity measures the ability of the methodology to correctly reject unrelated method pairs:
Specificity = TN / (TN + FP)
Higher specificity indicates fewer incorrect matches.

False Positive Rate (FPR) measures the proportion of unrelated method pairs incorrectly classified as matches:
FPR = FP/ (FP + TN)
Lower FPR indicates better resistance to false positive mappings.

F1-Score combines Precision and Recall into a single metric using their harmonic mean:
F1=2×(Precision×Recall) / (Precision+Recall)
Higher F1-Score values indicate a better balance between precision and recall.

Mean Reciprocal Rank (MRR) is a ranking-based evaluation metric commonly used in information retrieval and recommendation systems:

$$MRR = \frac{1}{Q}\sum_{i=1}^{Q}\frac{1}{rank_i}$$

Where:

- Q is the total number of queries.
- Rank_i is the position of the first relevant result dor the i-th query.

For each evaluated method that contained a valid semantic mapping in the ground truth dataset, the reciprocal rank was computed as the inverse of the ranking position of the first correctly

identified mapping. If the correct mapping was returned as the first result, the reciprocal rank value was equal to 1. If the correct mapping appeared in the second or third position, the reciprocal rank became 1/2 or 1/3 respectively. If no correct mapping was returned, the reciprocal rank was assigned a value of 0.

Methods that did not contain any valid semantic mapping in the ground truth dataset were excluded from the MRR computation, since no correct ranking position exists for these cases.

The final MRR score was calculated as the average reciprocal rank across all evaluated methods that contained valid semantic mappings.

Higher MRR values indicate that the methodology ranks correct mappings close to the top of the candidate list

Section 3

Related Work

3.1 Ecosystems and Cross-Ecosystem Packages

Previous research has shown that modern software ecosystems should not be viewed as isolated environments. Instead, many libraries and packages are increasingly developed, distributed and maintained across multiple programming ecosystems. This phenomenon is commonly referred to as cross-ecosystem development, where a software package or library appears in more than one ecosystem.

Cross-ecosystems packages were first introduced by Constantinou et al. [13], who investigated cross-ecosystem packages across twelve package distributions. Their findings indicate that although the number of packages distributed across multiple ecosystems is relatively small compared to the total number of packages, these packages are often popular, actively maintained and can affect multiple ecosystems through their updates, bugs or security issues. The study also showed that issues originating from a single cross-ecosystem package may propagate across several ecosystems simultaneously, increasing the overall impact of failures and vulnerabilities.

Kannee et al. [18] studied libraries that cross-software ecosystem boundaries. Their study analyzed 1.1 million libraries from five major ecosystems, specifically PyPI (Python), NPM (JavaScript), Maven (Java), CRAN (R), and RubyGems (Ruby), and identified 4,146 GitHub repositories that release libraries across these ecosystems. Their results show that cross-ecosystem libraries are not rare cases but an important part of modern software development especially when projects aim to support developers from different programming communities.

The study also demonstrates that these libraries often become highly dependent upon within their ecosystems and attract contributors from multiple programming communities.

Li and Bezemer [14] further investigated cross-ecosystem software development. The authors analyzed 2,436 cross-ecosystem bindings for 546 machine learning libraries across thirteen software ecosystems. Their study focused on bindings, which allow libraries originally implemented in one programming language to be reused in different ecosystems. The results show that maintaining semantic consistency across ecosystems is challenging, as bindings frequently experience delays in supporting newer releases and often provide incomplete functionality compared to the original implementations.

Summarizing, the studies above are important because they show that software ecosystems are becoming increasingly interconnected. A library may be implemented in more than one language, released in different package managers or maintained under the same repository while targeting different developer communities. This creates practical challenges for developers because similar functionality may exist across ecosystems but may not be expressed using the same method names structures or implementation patterns.

3.2.1 Type IV Clone Detection / Method Mapping Related Work on Method Mapping Using Embeddings

The problem of identifying equivalent functionality across different software systems has been extensively studied in the context of API mapping, function mapping and code similarity analysis. In recent years embedding-based approaches have emerged as a powerful solution for capturing semantic relationships between code elements, enabling more accurate mapping across programming languages.

Traditional approaches to method mapping relied on syntactic analysis, heuristic matching, or manually defined rules. While these techniques can be effective for simple cases, they struggle when methods differ significantly in structure or naming. As a result, research has shifted toward learning-based approaches that can capture deeper semantic relationships.

One of the key advancements in this area is the use of code embeddings, which represent code elements as vectors in high-dimensional space. These embeddings encode both syntactic and semantic information, allowing similar methods to be positioned close to each other in the

vector space. This enables similarity-based matching between methods, even when they are implemented in different programming languages.

Several studies have explored embedding-based techniques for cross-language mapping. The SAR approach [21] proposes a method for learning cross-language mappings by aligning vector spaces of different programming languages using domain adaptation techniques. By combining embedding representations with adversarial training, SAR can identify mappings using only a small number of initial seed pairs. This demonstrates the effectiveness of embeddings in bringing the gap between different programming ecosystems.

Another important contribution is DeepAM [15] which applies deep learning techniques to learn semantic representations of API sequences across languages. Unlike traditional approaches that rely on bilingual projects, DeepAM learns mapping directly from large-scale code corpora by leveraging sequence to sequence learning. This approach highlights the potential of embeddings to capture relationships between code elements without requiring explicit alignment data.

Earlier work has also explored the use of textual information to support mapping. The TMAP [16] approach utilizes text mining techniques to compare descriptions and identify likely mappings based on similarity in documentation by representing textual descriptions in vector space, this method demonstrates the semantic similarity can be inferred from natural language providing an alternative to purely code-based analysis.

In addition, embedding-based approaches have been widely applied in code clone detection particularly for identifying semantic clones (Type-IV) [9]. In cross-language clone detection embedding models map code fragments from different programming languages into a shared representation space enabling direct comparison. This is closely related to method mapping as both tasks identifying semantically equivalent functionality across languages

Despite their effectiveness, embedding-based approaches also face challenges. The quality of the results depends on the representational power of the embedding model and the availability of high-quality training data. Additionally, similarity scores alone may not always reflect true functional equivalence, especially in complex cases. Therefore, embedding-based methods are often combined with additional validation techniques to improve accuracy.

Overall, existing research demonstrates that embeddings provide a robust and scalable solution for identifying semantic similarities between code elements. Their ability to represent methods in shared vector space makes them particularly suitable for cross-ecosystem method mapping. This thesis builds upon these ideas by using embeddings to compute similarity between methods from different programming ecosystems and identify the most relevant mappings.

3.2.2 Type IV Clone Detection / Method Mapping Related work on method mapping using LLMs

The recent advancement of Large Language Models (LLMs) has significantly influenced research in software engineering, particularly in tasks involving code understanding, generation, and similarity analysis. Due to their ability to process both natural language and source code, LLMs have increasingly explored as tools for identifying semantic relationships between code elements, including method level correspondences across programming languages.

Unlike embedding-based approaches, which rely on numerical vector representations, LLMs operate by interpreting code in a contextual and semantic manner. This enables them to reason for the functionality of code fragments, making them particularly suitable for tasks involving semantic similarity, such as identifying clones type IV. As a result, LLMs have been investigated as an alternative or complementary approach to traditional method mapping techniques.

Several studies have explored the use of LLMs for code clone detection. One such study evaluates the effectiveness of LLMs in detecting both mono-language a cross-language code clones, demonstrating that LLMs can achieve high performance [8]. The results indicate that LLMs are capable of understanding code semantics beyond surface-level syntax, making them a promising tool for identifying functional similarities.

Further research compares LLM-based approaches with embedding-based models for cross-language clone detection [9]. Findings suggest that while LLMs can achieve strong performance in simpler cases, embedding models often outperform them in more complex

scenarios due to their ability to provide consistent numerical representations. This highlights the trade-off between the reasoning capabilities of LLMs and the stability of embedding-based approaches.

In addition, studies evaluating the performance of different LLMs such as GPT models reveal that their effectiveness varies depending on the type being analyzed. While LLMs perform well on simpler clone types, their accuracy decreases when detecting complex semantic clones (type IV) which require deeper functional understanding [3]. This limitation is particularly relevant in method mapping tasks where subtle differences in implementation may affect correctness.

Beyond detection, LLMs have also been used for generating and analyzing code. Research on code clone generation shows that LLMs can produce syntactically diverse implementations of similar functionality, but may struggle to maintain consistent behavioral equivalence, especially in cross-language settings [2]. This suggests that while LLMs can capture semantic intent, they may introduce variability that complicates accurate mapping.

To address these limitations, recent approaches propose hybrid methods that combine LLMs with other techniques. For example, HyClone framework integrates LLM-based semantic analysis with execution-based validation to improve the detection of semantic clones [7]. By combining reasoning capabilities with runtime verification, such approaches achieve higher accuracy and reliability.

Furthermore, studies on LLM-based clone detection and refactoring highlight that while modern models can effectively identify simpler code similarities, they still struggle with more complex transformations and mappings [5]. This reinforces the idea that LLMs should be used as part of a broader system rather than as standalone solutions.

Overall, the existing literature demonstrates that LLMs offer significant potential for identifying semantic similarities between methods, particularly in cross-language contexts. However, their limitations in handling complex cases and ensuring consistency suggest that they are best used in combination with other techniques. In this thesis, LLMs are employed as complementary evaluation mechanism to assess the quality of method mappings generated

through embedding-based similarity, combining the strengths of both approaches to improve overall accuracy.

Section 4

Methodology

4.1 Data Collection

The data used in this thesis originates from the dataset introduced in the study "intertwining communities: exploring libraries that cross-software ecosystems" by Kanee et al.[19]. The dataset was made publicly available through Zenodo and contains information about libraries that exist across multiple software ecosystems. The study identifies repositories that distribute packages in more than one ecosystem, making the dataset particularly suitable for research involving cross- ecosystem analysis.

This dataset was selected because it provides real-world examples of software projects that operate across different programming environments. Since the objective of this thesis is to identify semantically similar methods between ecosystems, it is important to work with packages that are already related through shared functionality or common development goals. Cross-ecosystem packages offer a natural foundation for this type of analysis as they often implement equivalent features in multiple programming languages.

Another important advantage of the dataset is its scale diversity. The original study analyzed more than one million libraries from ecosystems such as, PyPi, npm, Maven, RubyGems and CRAN. This large scale perspective increases the reliability of the dataset and ensures that selected packages represent realistic software development practices rather than isolated examples.

From the available ecosystems, this thesis focuses specifically on Python and JavaScript. These ecosystems were selected for several reasons. First, they are among the most widely used programming languages in modern software development, supported by large developer

communities and extensive package repositories. Python is heavily used in data science, machine learning, backend systems, and automation while JavaScript dominates web development and interactive applications. As a result, many projects provide implementations in both ecosystems to support broader adoption and interoperability.

Second, Python and JavaScript represent two ecosystems with different programming styles and conventions. Python emphasizes readability and simplicity, whereas JavaScript is closely associated with event driven and web-oriented development. These differences make cross-language method mapping more challenging and therefore more suitable for evaluating semantic similarity techniques.

Finally, the existence of many shared or equivalent libraries makes these ecosystems particularly appropriate for cross-ecosystem analysis. Packages developed for both ecosystems often provide similar functionality while using different implementations, naming conventions, and APIs. This creates an ideal environment for studying method level semantic similarity and evaluating the effectiveness of embedding-based and LLM-based approaches.

After selecting Python and JavaScript as the target ecosystems, the original dataset was downloaded from Zenodo as two separate CSV files: one containing PyPi packages and one containing npm packages. The PyPi dataset initially contained approximately 153806 entities while the npm dataset contained 818,786 entities. Since the aim of the thesis is to study packages that exist across both ecosystems it was necessary to identify which Python and JavaScript packages belong under the same underlying project.

To achieve this, the two datasets were matched using their repository URLs. The repository URL was chosen as the matching attribute because cross-ecosystem packages may have different package names, but they often point to the same GitHub repository or source code location. Therefore, matching repository URL provides a more reliable way to identify packages that are connected across ecosystems.

Before performing the matching, the repository URLs were cleaned and normalized. This step was necessary because the same repository can be written in different formats such as with https://, http://, www., a. git suffix or trailing slashes. If these formatting differences were not removed, two URLs referring to the same repository could be treated as different values. For

this reason, the code converted URLs to lowercase and removed common formatting variations.

After cleaning the URLs, an inner merge was performed between PyPi and npm dataset using a normalized repository URL. An inner merge was selected because the objective was to keep only the packages that appeared in both ecosystems. The result of this process was a new CSV file containing matched Python – JavaScript package pairs. The final merged dataset contained approximately 21,265 cross-ecosystem entities representing projects that exist in both Python and JavaScript ecosystems.

After creating the merged PyPi-npm dataset, an additional cleaning step was required because the dataset contained several duplicated entities. These duplicates occurred because multiple package records could point to the same repository URL. Since the goal is to analyze unique cross-ecosystem projects, keeping duplicated repositories would have affected the reliability of the later stages.

To solve this problem, duplicate entries were removed using the normalized URL column, repo_clean. This column was selected because it represents the cleaned version of the repository address and therefore acts as a unique identifier for each project. The script loaded the merged CSV file, checked whether the repo_clean column existed, and then removed duplicate rows based on this column using the drop_duplicated() function in pandas.

This approach was used because repository URLs provide a more reliable identifier for that package name. By removing duplicates based on repo_clean column, the dataset reduced to unique cross-ecosystem entities. After this cleaning step, the final dataset contained 660 unique cross-ecosystem entities.

The dataset was then divided into two subsets. The validation subset contains the first 150 entities of the original dataset while the testing subset contained the rest of 510 entities. This separation was performed to support a more reliable evaluation process and to avoid performing all experiments on a single dataset.

The validation subset was used for preliminary experimentation, parameter adjustment, and verification of the methodology pipeline, while the larger testing subset was reserved for the

final evaluation of the proposed approach. Splitting the dataset in this way is common practice in machine learning and software engineering research because it helps reduce bias and improves the reliability of experimental results. By evaluating the methodology on unseen data, it becomes possible to better estimate how well the approach generalizes different cross-ecosystem projects.

After splitting the dataset, an additional filtering step was required to ensure that all selected repositories contained source code written in both Python and JavaScript. Although the repositories were identified as cross-ecosystem projects, some repositories contained code only for one language or included only configuration, documentation or auxiliary files for the other ecosystem. Including such repositories would negatively affect the later stages of method extraction and semantic similarity analysis.

To address this issue, the CLOC (Count Lines of Code) tool was used [31]. CLOC is an open-source utility designed to analyze software repositories and count lines of code for different programming languages. It provides statistics such as the number of files, blank lines, comment lines, and actual lines of source code for each detected language. CLOC was selected because it offers a fast and reliable way to automatically identify the programming language present in a repository and measure the amount of source code associated with each language.

First clone each GitHub repository locally and then execute the CLOC tool using JSON output format. The script extracted language specific statistics, particularly the number of lines of Python and JavaScript code. Repositories with zero lines of Python or JavaScript were automatically discarded. This filtering ensured that all remaining repositories genuinely contained implementations in both target languages.

In addition to filtering repositories, several useful metrics also stored, including the total number of files, total lines of code, number of detected languages, and language specific code statistics. These metrics were saved back into new CSV files for future processing.

After applying the CLOC filtering process, the resulting datasets were one subset for validation with 45 entities and one subset for testing with 257 entities. From these filtered datasets, a smaller subset of repositories was manually selected for experimentation. Specifically, 10 repositories were selected for validation, and 30 repositories were selected for testing. The

selection process was not random. Instead, the repositories were intentionally chosen to ensure diversity in both repository size and functional similarity across ecosystems.

The objective of this selection strategy was to cover different experimental scenarios. Therefore, the selected repositories included:

- Repositories with similar functionality and small codebases
- Repositories with similar functionality and large codebases
- Repositories with similar functionality and medium size codebases
- Repositories without similar functionality and small codebases
- Repositories without similar functionality and large codebases
- Repositories without similar functionality and medium size codebases

This approach was important because it allowed evaluation of the proposed method mapping technique under different conditions. Small repositories are generally easier to analyze and may contain fewer methods, while larger repositories introduce higher complexity and greater variety of implementations. Similarly, repositories with highly related functionality are expected to contain stronger semantic correspondences whereas unrelated repositories help evaluate whether the methodology can avoid false positive mappings.

The categorization of repositories according to size was primary based on the number of lines of code obtained from the CLOC analysis. The decision regarding whether repositories shared similar functionality was performed manually through inspection of the repositories for their description and their overall purpose. This process relied on direct observation and logical evaluation to ensure that the selected repositories represented a broad range of realistic cross-ecosystem scenarios.

By using this structured and non-random selection strategy, the experimental datasets provided a more comprehensive and balanced evaluation environment for the later stages of the thesis.

4.2 Method Extraction

After constructing the final cross-ecosystem datasets, the next stage of the methodology focused on extracting methods from the selected Python and JavaScript repositories. Method extraction was necessary because the main objective of this thesis is to identify semantically similar methods across ecosystems that likely implement similar functionality in each programming language. Therefore, the repositories first had to be transformed from raw source code into structured collections of methods that could later be analyzed and compared.

Two separate extraction pipelines were developed: one for Python repositories and one for JavaScript repositories. Separate implementations were required because the two ecosystems use different syntax rules, language constructs, and programming paradigms. Although both pipelines followed the same general objective, each pipeline used language-specific parsing techniques that are suitable for the corresponding ecosystem.

For the Python repositories, the extraction process was implemented using Python's build-in AST (Abstract Syntax Tree) module [32]. The AST module parses Python source code into an abstract syntax tree, which is a tree-based structural representation of the source code. This allows to get syntactical analysis of the source code rather than relying on simple text processing. This approach was selected because AST-based analysis is significantly more reliable than regular expression or keyword based parsing, especially when dealing with large and complex source code.

The Python extraction pipeline recursively traversed repository directories and collected all .py files while ignoring non relevant directories such as .git, node_modules, dist, virtual environments, cache directories and build folders. These directories were excluded because they typically contain generated files, dependencies, or temporary artifacts that are not part of the actual implementation. Excluding them, reduces noise and improves extraction efficiency. Different Python extensions were considered but were not found in the gathered dataset.

Each Python file was parsed into an Abstract Syntax Tree, and a custom visitor class based on ast.NodeVisitor was used to traverse the tree structure and identify methods and functions. The extractor handled:

- Regular functions
- Class methods
- Asynchronous functions

- Asynchronous methods
- Nested functions

For every extracted method several attributes were stored, including:

- Method name
- Fully qualified name
- File path
- Method type
- Parameter list
- Starting and ending line numbers
- Source code snippet
- A unique method identifier

The fully qualified name was particularly important because methods with identical names may exist in different classes or scopes. By storing hierarchical information, the methodology preserved the contextual structure of each method inside the repository.

The JavaScript extraction process followed a similar philosophy but required different tools because JavaScript syntax is significantly more dynamic and flexible. The implementation was developed using Node.js together with the Babel parser and Babel traversal utilities [33]. Babel was selected because it supports modern JavaScript syntax including JSX, class properties, decorators, optional chaining, arrow functions and other advanced language features commonly found in contemporary repositories.

The JavaScript extractor recursively traversed repository directories and collected files with extensions such as .js, .jsx, .ts, .tsx, mjs and .cjs. Like Python pipeline directories such as node_modules, .git, dist and build were excluded to avoid processing external dependencies and general artifacts.

After parsing each file into an Abstract Syntax Tree, the traversal process identified multiple types of functions and methods including:

- Function declarations
- Arrow functions
- Object methods

- Object-property functions
- Class methods
- Private class methods
- Class- property functions
- And assignment-based methods

Supporting assignment-based methods was particularly important for JavaScript because functions are often dynamically assigned to objects or prototypes rather than declared explicitly. The extraction process therefore included additional logic to reconstruct qualified method names and preserve ownership relationships between functions and objects.

For both languages, code snippets were extracted together with structural metadata. Limiting the extracted snippet length ensured that the stored methods remained manageable while still preserving enough contextual information for later semantic analysis. Additionally, every extracted method received a unique identifier generated from the file path, qualified name, and line numbers. This guaranteed consistent traceability across the later stages of the methodology.

Finally, the extracted methods were stored in JSON and CSV formats. The JSON files preserved the complete extracted information, while the CSV files provide a more compact tabular representation suitable for further processing and analysis.

This method extraction stage was essential because it transformed large software repositories into structured method level datasets. By using AST-based parsing and language aware extraction techniques, the methodology ensured that the extracted methods accurately represented the functional structure of the original repositories while minimizing noise and parsing errors.

4.3 Embeddings and Cosine Similarity

After extracting methods from the selected Python and JavaScript repositories, the next stage of the methodology focused on generating semantic vector representations, also known as embeddings, for each extracted method. The objective of this stage was to transform methods

into numerical representations that preserve semantic information, enabling comparison between methods implemented in different programming languages.

Traditional textual comparison techniques[4] are generally insufficient for identifying semantic similarity across ecosystems because methods with equivalent functionality may use completely different syntax, naming conventions, or implementation styles. For this reason, embedding-based representations were used as they provide a way to capture contextual and semantic information from source code.

To generate embeddings, this thesis used UniXcoder [23], a transformer-based model developed for code understanding tasks. UniXcoder is a pre-trained language model specifically designed for source code representation learning and supports multiple programming languages. It was selected because it provides strong cross-language semantic representations and has been widely used in tasks such as code search, clone detection and code similarity analysis.

The embedding process was implemented separately for Python and JavaScript methods due to differences in the extraction pipelines, repository structures, and parsing mechanism used for each programming language. Python embedding generation was integrated directly into the extraction pipeline because the Python extraction stage already produced the required structured method representation during AST traversal. In contrast, JavaScript repositories required separate embedding generation stage after extraction due to the different parsing and preprocessing workflow used for JavaScript files.

Before computing embeddings, the extracted method information was transformed into a combined code and metadata representation. Instead of using only the raw source code, the input combined:

- The programming language
- Qualified method name
- Method type
- Source code snippet

This information was merged into a textual structure before being passed to UniXcoder. Including metadata together with the source code was important because semantic similarity is

often influenced not only by implementation details, but also by contextual information such as method hierarchy and method type. This design aimed to improve cross-language semantic alignment between methods that may perform similar functionality while using different naming conventions or implementations.

The UniXcoder tokenizer converted the textual representation into tokens, which were processed by the transformer model. The resulting hidden state representations were aggregated using masked mean pooling to generate a fixed length embedding vector for each method. This pooling strategy computes the average contextual representation of valid tokens while ignoring padding tokens producing a compact semantic representation suitable for similarity analysis. Padding tokens are placeholder tokens added to shorter input sequences to ensure that all input sequences have the same length during transformer processing.

The implementation also included several preprocessing and optimization mechanisms. Input sequences were truncated to a maximum length of 512 tokens to ensure compatibility with the transformer architecture and reduce computational overhead. Additionally, GPU acceleration was used whenever CUDA was available, significantly improving the performance of embedding generation for large repositories.

Once embeddings were generated, they were stored together with the extracted method metadata in CSV files. Each row contained:

- The method name
- Qualified name
- File path
- Line numbers
- Method type
- Unique method identifier
- Corresponding embedding vector

This structured representation enables efficient downstream processing and similarity computation.

After generating embeddings for both Python and JavaScript methods, cosine similarity was used to compare embedding vectors across languages. Cosine similarity measures the angular similarity between two vectors and is commonly used in embedding-based semantic analysis

because it focuses on vector orientation rather than magnitude. Methods with higher cosine similarity scores were considered more likely to represent semantically equivalent functionality.

The cosine similarity between two vectors A and B is computed using the following equation:
$\cos(\theta) = A \cdot B / \| A \| \| B \|$
Where:

- $A \cdot B$ represents the dot product of the vectors
- $\| A \|$ and $\| B \|$ represent the vector magnitudes
- And values closer to 1 indicates stronger semantic similarity

To reduce computational complexity and focus on the most relevant candidate mappings, the methodology computed the Top 10 most similar candidate methods for the language containing the smaller number of extracted methods. This approach was selected because repositories frequently contain different numbers of methods between the Python and JavaScript implementations. Performing exhaustive comparisons between all methods would significantly increase computational cos and generate a large number of irrelevant candidate mappings. By selecting the smaller method set as the source comparisons, the total number of pairwise computations was reduced while still preserving the most semantically relevant matches.

Restricting the comparison results to the Top 10 candidates was also important for the subsequent LLM evaluation stage. Passing every possible method pair to the LLM would introduce excessive noise and substantially increase processing time and computational requirements. Therefore, retaining only the highest ranked candidates according to cosine similarity allowed the methodology to focus on the most promising semantic mappings.

In addition to Top 10 restrictions, a cosine similarity threshold equal to 0.50 was applied to further reduce the number of candidate mappings passed to the LLM stage. Only method pairs with cosine similarity scores greater than or equal to 0.50 were retained as valid candidates.

The threshold value of 0.50 was selected after evaluating multiple experimental thresholds. Initially, a lower threshold of 0.30 was tested. Although this threshold produced a larger number of candidate mappings, many of these additional mappings were weakly related and

introduced considerable noise into the evaluation process. As a result, the lower threshold did not significantly improve the quality of the semantic matches.

A higher threshold of 0.75 was also evaluated. However, this threshold proved to be too restrictive, as many semantically correct mappings failed to satisfy the threshold requirement. This occurs because semantically equivalent methods implemented in different programming languages may still produce moderate embedding similarity scores due to differences in syntax, implementation style, framework usage, and ecosystem specific programming conventions.

Therefore, the threshold value of 0.50 provided a balanced tradeoff between precision and coverage. It successfully reduced the number of irrelevant candidate mappings while still preserving a sufficiently large set of potentially correct semantic correspondences for further analysis using LLMs.

In addition to the similarity scores, the generated Csv files stored detailed information about each candidate mapping. Each entry included:

- The python method identifier
- The JavaScript method identifier
- Method names
- Qualified names
- File paths
- Method types
- Cosine similarity score
- And ranking position within the top 10 candidate list

The pipeline also included mechanism for:

- Avoiding repeated processing
- Skipping repositories that were already processed
- Validating output files
- Organizing outputs into repository specific directories

This automation improved scalability and ensured reproducibility across multiple repositories and experiments.

Overall, the embedding stage transformed extracted methods into semantic vector representations that could be compared across ecosystems. By combining transformer-based embeddings with cosine similarity analysis, the methodology enabled the identification of candidate method mappings between Python and JavaScript repositories despite differences in syntax and implementation style.

4.4 LLMs

After generating candidate method mappings using embeddings and cosine similarity, the final stage of the methodology focused on evaluating these candidates using a Large Language Model (LLM). The objective of this stage was to improve the semantic validation of the candidate mappings and reduce false positive matches that could not be reliably filtered using embedding similarity alone.

Although embedding-based similarity is effective for identifying semantically related methods, cosine similarity scores alone cannot always determine whether two methods truly implement equivalent functionality. Methods from different programming languages may appear semantically similar due to shared terminology or contextual patterns while still performing different operations. For this reason, an additional semantic reasoning stage using LLM was introduced.

The LLM evaluation stage was implemented using Ollama together with the CodeLlama model [23]. Ollama is a framework that follows LLMs to run locally on a machine without requiring external API services. It was selected because it enables efficient local inference, improved reproducibility, and full control over evaluation pipeline. Running the model locally also avoided external API cost and reduced dependency on third party services.

The selected model, CodeLlama, is a transformer-based language model specialized for source code understanding and generation tasks. It was chosen because it has been trained on large scale code datasets and demonstrates strong performance in code analysis, reasoning, and semantic understanding. These characteristics made it suitable for evaluating whether two methods written in different programming languages could represent possible Type-IV semantic clones.

The LLM stage operated on the candidate mappings generated during the cosine similarity stage. Instead of evaluating all possible method pairs, only the filtered top 10 candidates with cosine similarity score above the selected threshold were passed to the LLM. This significantly reduced computational complexity and allowed the model to focus on the most promising semantic candidates.

Before sending data to the LLM, the methodology constructed detailed method specifications for both the source method and the candidate methods. The implemented pipeline extracted:

- Method identifiers
- Programming language
- File path
- Qualified method name
- Generated method signature
- Dependencies / imports
- Detected exceptions
- Outputs
- Source code excerpts

The source code snippets were extracted directly from repository files using the line ranges stored inside the method identifiers. This ensured that LLM evaluated the actual implementation code rather than only metadata or embeddings. Additionally, imports and dependencies were extracted because external libraries and framework usage can provide useful contextual information about method functionality.

To improve efficiency, generated method specifications were cached as JSON files. This caching mechanism prevented repeated extraction and preprocessing of methods that had already been analyzed in previous executions. As a result, the pipeline became more scalable and avoided unnecessary repeated computations.

A prompt engineering strategy was then used to guide LLM during semantic evaluation. The methodology experimented with multiple prompt designs before selecting the final prompt used in the experiments.

Initially, a large and highly detailed prompt was tested. This prompt included extensive explanations about Type IV semantic clones, detailed comparison rules, and multiple evaluation criteria. However, the excessive amount of information often caused confusion for the LLM. In several cases, the model produced inconsistent results or focused on less relevant details instead of the core semantic functionality of the methods.

A second version used significantly smaller and simpler prompt. Although this prompt reduced complexity, it did not provide enough contextual guidance for accurate semantic reasoning. As a result, the model occasionally produced weak or overly simplistic matched because the instructions were insufficient to clearly define the expected semantic comparison behavior.

After experimentation, a balanced prompt design was selected as the final approach. The final prompt provided concise but clear instructions regarding cross-language Type IV semantic clone detection while avoiding unnecessary detail. The prompt instructed the model to:

- Compare one source method against up to ten candidate methods
- Identify up to three best matches
- Avoid relying only on method-name similarity
- Return only valid JSON output containing candidate identifiers, confidence scores and Type IV clone indicators

The prompt also explicitly defined that higher scores should indicate stronger semantic similarity, and that zero matches should be returned when no plausible semantic correspondence existed. This instruction was particularly important because it reduced the tendency of the LLM to force weak matches.

The communication with Ollama was implemented using HTTP requests to the local Ollama inference server. Several inference parameters were configured to improve consistency and stability:

- Temperature = 0.0 was used to reduce randomness
- JSON output format was enforced to simplify automated parsing
- Context length parameters were increased to support larger code snippets
- Retry mechanisms were added to handle inference failures or malformed outputs

After receiving LLM response the output was parsed and validated. Only candidate method identifiers that originally appeared in the cosine similarity top 10 candidate set were retained. The returned matches were then sorted according to the LLM confidence score, and only up to the top 3 highest ranked matches were stored in the final results.

The generated CSV output stored:

- Source method information
- Candidate method information
- LLM similarity score
- Whether the candidate was classified as a possible Type IV semantic clone

Overall, the LLM evaluation stage provided an additional semantic layer on top of the embedding base similarity approach. By combining embedding similarity with LLM-based semantic validation, the methodology aimed to improve the quality and reliability of cross-ecosystem method mapping while reducing false positive candidate matches.

4.5 Evaluation Outcome Classification

Before presenting the repository-specific results, the evaluation used for assigning True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN) must first be defined. Since the proposed methodology combines embeddings, cosine similarity, and LLM-based semantic evaluation, the classification process differs slightly from traditional binary classification systems.

A True Positive (TP) was assigned when LLM successfully identified the correct semantic mapping between a Python and JavaScript method and labeled it as a possible Type-IV semantic clone. In other words, the correct candidate method existed within the Top 10 cosine similarity candidates, and the LLM correctly returned it as a valid match.

A False Positives (FP) occurred in two different scenarios. The First case occurred when the embedding and cosine similarity stage passed only incorrect candidates to the LLM, but the LLM still returned one or more matches instead of correctly returned zero results. In this scenario, the LLM incorrectly interpreted unrelated methods as semantic clones. The second case occurred because LLM was instructed to return up to three possible matches for each

source method. In several situations, the LLM correctly identified the valid mapping but additionally returned one or two extra incorrect matches and labeled them as possible Type-IV semantic clone. Although the correct mapping existed, the additional incorrect matches were still counted as false positives.

A True Negatives (TN) was assigned when the LLM correctly rejected unrelated candidate methods. This occurred in multiple scenarios. The first scenario was when the LLM received only incorrect candidates and correctly returned zero matches. The second scenario occurred when the LLM returned a candidate method but correctly labeled it as not a possible Type-IV clone. In both cases, the system correctly avoided classifying unrelated methods as semantic mappings. Additionally, a True Negative was also assigned when no candidates correctly passed the embedding and cosine similarity threshold for methods that did not contain valid semantic mappings.

Finally, a False Negative (FN) occurred when the correct semantic mapping successfully passed the embedding and cosine similarity threshold stage and was available to the LLM, but the LLM incorrectly returned zero matches. In this situation, a valid semantic correspondence existed but was missed during the LLM evaluation stage. A False Negative also assigned when methods that contained valid semantic mappings incorrectly failed to pass any candidates from the threshold, preventing the correct candidate from reaching the LLM evaluation stage.

This evaluation strategy was designed to reflect the behavior of the complete methodology pipeline rather than only the final LLM output. Since the system allows multiple candidate mappings and combines both similarity filtering and semantic reasoning, these definitions provide a more accurate representation of the strengths and weaknesses of the proposed approach.

Section 5

Results

5.2 Results for Validation Subset

The validation subset was used to evaluate the effectiveness of the proposed cross-ecosystem method mapping methodology under multiple repository conditions, including repositories with strong functional similarity, repositories with little or no semantic overlap and repositories of varying size and complexity. Ground truth mappings were manually identified through direct inspection of the repositories, method implementation, and repository description.

Table 5.1 presents the average evaluation of metrics across all repositories included in the validation subset.

Accuracy	Precision	Recall	Specificity	FPR	F1-Score	MRR
49%	33%	85%	40%	58%	44%	39%

Table 5.1: Average Evaluation Scores for Validation Subset

The validation subset demonstrates several important patterns regarding the behavior of the proposed methodology. First, repositories with clear cross-ecosystem semantic alignment and smaller codebases produce significantly strongest results. Smaller repositories generally contain fewer unrelated methods and clearer semantic correspondences, allowing both the embedding stage and LLM stage to operate more effectively.

Second, repositories with weak or nonexistent semantic overlap exposed a major limitation of the methodology. In these cases, the LLM frequently failed to return zero results even when no valid semantic mappings existed, resulting in extremely high false positive rates and very low specificity.

Third, larger and more complex repositories introduced substantial variability in performance. Complex repositories often contain diverse implementation styles and partial semantic overlaps that complicated both embedding- cosine similarity and LLM reasoning. As repository complexity increases, precision and specificity generally decrease due to a growing number of unrelated but semantically weak candidate mappings.

Overall, the results indicate that the proposed methodology performs better in smaller repositories and in repositories where the Python and JavaScript implementations share similar functionality. In these cases, semantic correspondences are clearer, resulting in higher accuracy, precision, and specificity. In contrast, larger repositories and repositories with limited functional similarity produced significantly more false-positive matches, as the embedding and LLM stages frequently identified weak contextual similarities as valid mappings. Although the methodology consistently achieved high recall when correct candidates were passed to the LLM, reducing false positives and improving semantic rejection remains a major challenge

5.1.1 Analytical Results for Validation Subset

The repository appfirst/statsd_clients produced a relatively strong semantic correspondence between Python and JavaScript implementations. The embedding and cosine similarity stage successfully passed correct candidates for 15 out of 18 methods. During the LLM evaluation stage, 13 (out of 15) correct mappings were successfully identified. However, the LLM also generated a large number of false positive matches. These results indicate that the methodology was highly effective at identifying existing semantic correspondences but struggled significantly to reject incorrect candidates. The perfect recall demonstrates that no valid mappings were missed, while the extremely low specificity reveals severe overprediction behavior by the LLM stage.

TP	TN	FP	FN	Accuracy	Precision	Recall	Specificity	FPR	F1-Score	MRR
13	0	19	0	25%	25%	100%	0%	100%	40%	63%

Table 5.2: appfirst/statsd_clients Repository Results

In the bcoe/karait repository the embedding and cosine similarity stage successfully passed correct candidates for 5 out of the 6 evaluated methods. The sixth method did not have a real semantic mapping, but candidates were still incorrectly passed. The LLM correctly identified the five valid mappings while also producing results for the unmatched method. Similar to the previous repository, the methodology achieved excellent recall but completely failed to reject unrelated candidates.

TP	TN	FP	FN	Accuracy	Precision	Recall	Specificity	FPR	F1-Score	MRR
5	0	8	0	38%	38%	100%	0%	100%	55%	83%

Table 5.3: bcoe/karait repository Results

The wq.app repository represented one of the weakest performing cases in the validation subset. The embedding stage correctly filtered several methods by preventing candidates from passing the similarity threshold, but in all remaining cases unrelated candidates still pass to the LLM stage. The LLM consistently failed to return zero results and instead produced incorrect semantic matches for every candidate set. These results demonstrate a complete failure of the semantic validation stage in repositories where embedding stage does not provide sufficiently accurate candidate filtering.

TP	TN	FP	FN	Accuracy	Precision	Recall	Specificity	FPR	F1-Score	MRR
0	3	9	0	25%	No TP	No TP	25%	75%	No Recall	0

Table 5.4: wq.app Repository Results

The baurca/configable repository produced more balanced results. From 19 evaluated methods, seven semantic mappings were expected. The embedding stage correctly prevented unrelated candidates in five cases, while seven additional methods incorrectly passed unrelated candidates above the similarity threshold. The LLM correctly identified five valid mappings.

Compared to previous repositories, this repository demonstrates slightly improved specificity because the methodology correctly rejected some unrelated methods. However, the large number of false positives still significantly reduced overall precision.

TP	TN	FP	FN	Accuracy	Precision	Recall	Specificity	FPR	F1-Score	MRR
5	11	14	0	53%	26%	100%	44%	56%	41%	33%

Table 5.5: baurca/configable Repository Results

The simplediff repository produced the strongest overall performance within the validation subset. All four expected semantic mappings were successfully identified during the embedding stage and correctly validated by the LLM. Additionally, the methodology generated relatively few false positives while correctly rejecting several unrelated candidates. These results indicate that the methodology performs particularly effectively in smaller repositories with clear and well-aligned cross-language functionality. The high specificity and relatively high precision demonstrate improved ability to reject incorrect candidates in simpler repository structures.

TP	TN	FP	FN	Accuracy	Precision	Recall	Specificity	FPR	F1-Score	MRR
4	5	2	0	81%	66%	100%	71%	28%	79%	100%

Table 5.6: simplediff Repository Results

The petli/codplayer repository demonstrates one of the most challenging evaluation scenarios. The repository contained 164 methods for which no semantic mappings were expected between the Python and JavaScript implementations. Nevertheless, the embedding stage still passed several candidates above the similarity threshold, and the LLM consistently failed to reject them. These results highlight a major weakness of the methodology when applied to complex and large repositories with no functional similarity. The LLM frequently interpreted weak semantic relationships as potential matches, generating excessive false positives predictions.

TP	TN	FP	FN	Accuracy	Precision	Recall	Specificity	FPR	F1-Score	MRR
0	136	54	0	71%	No TP	No TP	71%	28%	No Recall	0

Table 5.7: petli/codplayer Repository Results

The aws_kinesis_agg repository represented a more complex medium sized evaluation scenario. The repository contained 25 evaluated methods. During the embedding stage 10 methods correctly did not pass any candidate because no valid mappings existed, 2 methods incorrectly failed to pass valid candidates, 4 methods incorrectly passed candidates despite having no semantic match, 3 methods passed candidates that were not the correct mappings and 6 methods successfully passed correct semantic candidates. During the LLM stage 5 out of 6 correct candidates were correctly identified, several methods produced additional false positives, some unrelated candidates were correctly rejected, and some correct mappings were missed. Unlike previous repositories, this repository demonstrated reduced recall due to the presence of false negatives indicating that correct mappings were sometimes lost during evaluation process.

TP	TN	FP	FN	Accuracy	Precision	Recall	Specificity	FPR	F1-Score	MRR
5	13	11	6	51%	31%	45%	54%	45%	36%	43%

Table 5.8: aws_kinesis_agg Repository Results

The repository martinthomson/encrypted-content-encoding also represents a complex evaluation scenario. A total of 13 semantic mappings were expected across 23 evaluated methods. The embedding stage successfully passed correct candidates for all 13 valid mappings, while unrelated candidates also passed the threshold for nine methods. One method correctly produced no candidates. During the LLM stage 7 correct mappings were successfully identified, 5 incorrect candidate groups were correctly rejected, 6 correct mappings were incorrectly rejected, and several unrelated candidates still produced false negative results. This repository highlights the trade-off between precision and recall. While the methodology could reject several incorrect candidates, it is also incorrectly rejected many valid semantic mappings, reducing recall substantially.

TP	TN	FP	FN	Accuracy	Precision	Recall	Specificity	FPR	F1-Score	MRR
7	9	12	6	47%	36%	53%	42%	57%	42%	59%

Table 5.9: martinthomson/encrypted-content-encoding Repository Results

The romantolkachyov/django-bridge repository contained only three evaluated methods and no expected semantic mappings. Two methods correctly produced no candidates. However, one method incorrectly passed candidates to the LLM, which subsequently produced a false semantic mapping. This repository again demonstrated the difficulty of forcing the LLM to confidently return zero matches in repositories without semantic overlap.

TP	TN	FP	FN	Accuracy	Precision	Recall	Specificity	FPR	F1-Score	MRR
0	3	2	0	60%	No TP	No TP	60%	40%	No Recall	0

Table 5.10: romantolkachyov/django-bridge Repository Results

Finally, the 5monkeys/socket repository showed strong recall but poor precision. The repository contained three expected semantic mappings, 2 out of 3 were correctly identified by both the embedding and LLM stages. However, eight unrelated methods also passed the similarity threshold and generated additional false positive predictions by LLM. These results indicate that while the methodology successfully preserves valid mappings, it remains highly susceptible to overprediction in repositories containing many semantically weak candidates.

TP	TN	FP	FN	Accuracy	Precision	Recall	Specificity	FPR	F1-Score	MRR
2	11	16	0	44%	11%	100%	40%	59%	19%	13%

Table 5.11: 5monkeys/socket Repository Results

5.2 Results for Test Subset

The testing subset confirmed the main observations identified during the validation stage. Repositories with strong functional similarity generally produced higher recall and stronger overall performance because correct semantic mappings passed the threshold and were easier for the LLM to identify. Smaller and medium sized repositories with clearer functionality also tended to achieve better precision and specificity. Additionally, the MRR results indicate that when the methodology successfully identified the correct semantic mapping, the correct candidate was commonly ranked in the first position of the candidate list.

Table 5.12 presents the average evaluation metrics across the ten randomly selected repositories of the testing subset

Accuracy	Precision	Recall	Specificity	FPR	F1-Score	MRR
48%	37%	93%	38%	60%	51%	43%

Table 5.12 Average Evaluation Scores for Test Subset

In contrast, repositories with little or no semantic overlap consistently produced extremely high false positive rates. In these scenarios, the LLM frequently failed to confidently reject unrelated candidates and instead generated incorrect semantic matches. Large repositories additionally increased the number of unrelated methods and semantically weak candidate mappings, causing precision and specificity to decrease substantially despite maintaining high recall.

The testing subset reinforces the conclusion that the proposed methodology is highly effective at preserving valid semantic mapping once they pass the threshold but still faces significant challenges regarding semantic rejection and false positive generation, particularly in large or functionally unrelated repositories.

5.2.1 Analytical Results for Test Subset

The alphacep/vosk-api repository produced considerably stronger results. Out of the 19 evaluated methods, 16 valid semantic matches existed. The embedding stage successfully passed the correct candidate for all 16 methods. Additionally, one method without mapping

correctly produced no candidates while the two remaining methods incorrectly passed candidates above the threshold. The LLM successfully identified all 16 valid mappings while also correctly labeling several unrelated candidates as non-Type-IV clones. This repository demonstrates that the methodology performs better when strong semantic alignments exist between the Python and JavaScript implementations.

TP	TN	FP	FN	Accuracy	Precision	Recall	Specificity	FPR	F1-Score	MRR
16	14	25	0	54%	39%	100%	35%	64%	56%	86%

Table 5.13: alphacep/vosk-api Repository Results.

The bokeh/jupyter_bokeh repository represented a scenario where no semantic mappings existed between the evaluated methods. Out of nine methods, the embedding stage correctly rejected candidates only once, while unrelated candidates incorrectly passed the threshold for the remaining methods. During the LLM stage, only two of the incorrect candidate groups were correctly rejected. The remaining methods generate several false positives. These results further confirm that the methodology struggles substantially when repositories contain no actual semantic overlap.

TP	TN	FP	FN	Accuracy	Precision	Recall	Specificity	FPR	F1-Score	MRR
0	6	8	0	42%	No TP	No TP	42%	57%	No Recall	0

Table 5.14: bokeh/jupyter_bokeh Repository Results

The repository chokobole/JavaSrotobuf_bytes contained 13 evaluated methods with 12 expected semantic mappings. The embedding and cosine similarity stage performed strongly, successfully passing the correct candidate for all 12 methods while correctly rejecting the single unrelated method. However, the LLM stage only successfully identified seven of the valid mappings. In the remaining cases, the model either failed to identify the correct mapping or generated incorrect semantic associations producing many false positives. These results indicate that even when embeddings provide highly accurate candidate mappings, the LLM stage may still generate substantial semantic noise.

TP	TN	FP	FN	Accuracy	Precision	Recall	Specificity	FPR	F1-Score	MRR
7	3	24	0	29%	22%	100%	11%	88%	36%	58%

Table 5.15: chokobole/JavaSrotobuf_bytes Repository Results

The ciscous/intersight-rest repository produced one of the strongest performances within the testing subset. The repository contained nine methods and nine expected mappings. The embedding stage successfully passed the correct candidate in every case. The LLM then correctly identified eight out of the nine mappings while also correctly rejecting most unrelated candidates. These results show that the methodology can achieve very strong performance when repositories contain highly aligned functionality and relatively clear semantic correspondences.

TP	TN	FP	FN	Accuracy	Precision	Recall	Specificity	FPR	F1-Score	MRR
8	6	2	0	87%	80%	100%	75%	25%	88%	88%

Table 5.16: ciscous/intersight-rest Repository Results

The commonsmachinery/libcredit repository represented a large and more complex evaluation scenario. Out of 61 evaluated methods, 42 semantic mappings existed. The embeddings and cosine similarity stage successfully passed correct candidates in 32 cases, correctly rejected candidates for 7 unrelated methods, and failed to pass valid mappings in two cases. Several unrelated candidates also incorrectly candidates also incorrectly passed the similarity threshold. During the LLM stage, 28 out of the correctly passed mappings were correctly identified while four valid mappings were missed. Additionally, many unrelated candidates generated false positive predictions, although some were correctly labeled as non-Type-IV clones. These results illustrate how repository complexity and size substantially increase false positive generation even when recall remains high.

TP	TN	FP	FN	Accuracy	Precision	Recall	Specificity	FPR	F1-Score	MRR
28	32	61	3	48%	31%	90%	34%	65%	46%	50%

Table 5.17: commonsmachinery/libcredit Repository Results

The django-partialaajx repository represented another scenario without semantic overlap. The repository contained 19 methods with no expected matchings. The embedding stage correctly rejected candidates for only four methods, while incorrect candidates passed the threshold for the remaining methods. The LLM correctly returned zero results only twice and labeled only one additional method as non-Type-IV clone. Like previous repositories without semantic overlap, the methodology struggled heavily with semantic rejection.

TP	TN	FP	FN	Accuracy	Precision	Recall	Specificity	FPR	F1-Score	MRR
0	7	35	0	16%	No TP	NO TP	16%	83%	No recall	0

table 5.18: django-partialaajx Repository Results

The FENIX repository produced balanced results with heavily high recall. The repository contained 14 evaluated methods and 8 valid mappings. The embedding stage successfully passed the correct candidates for all eight mappings. From the remaining six methods without mappings, only two correctly produced no candidates. The LLM successfully identified all eight valid mappings while also producing several false positives and true negatives. These results indicate that although the methodology consistently preserves valid mappings, false positive reduction remains difficult in repositories with partial semantic overlap.

TP	TN	FP	FN	Accuracy	Precision	Recall	Specificity	FPR	F1-Score	MRR
8	10	14	0	56%	36%	100%	41%	58%	52%	58%

Table 5.19: FENIX Repository Results

The uonet-request-signe repository contained seven methos and five semantic mappings. Embedding and cosine similarity stage passed candidates for all methods including methods without actual mapping. During the LLM stage four out of the five valid mappings were correctly identified. Compared to several previous repositories, this repository achieved slightly improved specificity and reduced false positive generation.

TP	TN	FP	FN	Accuracy	Precision	Recall	Specificity	FPR	F1-Score	MRR
4	5	6	0	60%	40%	100%	45%	54%	57%	71%

Table 5.20: uonet-request-signe Repository Results

The FireEye repository contained 10 extracted methods, from which only 3 methods were expected to have valid semantic mappings. During the embedding and cosine similarity stage, the correct candidates successfully passed the threshold for 2 of the 3 mapped methods. While several incorrect candidates were also forwarded to the LLM stage. The LLM correctly identified the two valid mappings but struggled to reject unrelated candidates producing a large number of false positives. preserving correct mappings once they passed embedding stage, but had difficulty filtering semantically unrelated candidates.

TP	TN	FP	FN	Accuracy	Precision	Recall	Specificity	FPR	F1-Score	MRR
2	6	12	1	38%	14%	66%	33%	66%	22%	25%

Table 5.21: FireEye Repository Results

Finally, the text_selector repository represented one of the weakest performing testing cases. The repository contained 11 methods with no expected semantic mappings. The first stage correctly rejected candidates for seven methods but incorrectly passed unrelated candidates for four methods. However, the LLM failed on every occasion to return zero results or label candidates as non-Type-IV clones. Consequently, the repository produced only false positives.

TP	TN	FP	FN	Accuracy	Precision	Recall	Specificity	FPR	F1-Score	MRR
0	7	6	0	53%	No TP	No TP	53%	46%	No Recall	0

Table 5.22: text_selector Repository Results

Section 6

Discussion

6.1 Interpretation of Results and Limitations

This thesis investigates whether embeddings and LLMs can effectively identify semantically similar methods across Python and JavaScript ecosystems. The experimental results demonstrate that the proposed methodology is generally capable of preserving correct semantic mappings once the candidates successfully pass the embedding and cosine similarity stage. This behavior is reflected in the consistently high recall values observed across many repositories in both the validation and testing subsets.

The results indicate that the embedding stage was highly effective at capturing semantic relationships between methods implemented in different programming languages. In many repositories, the correct semantic mappings were successfully included within the Top 10 cosine similarity candidates. This suggests that transformed-based embeddings can successfully encode semantic characteristics of source code even when syntax and implementation patterns differ significantly between ecosystems.

Furthermore, the MRR results indicate that when correct semantic mappings were successfully identified, they were frequently ranked in the first candidate position. Smaller repositories generally produced higher MMR values because the reduced number of unrelated methods allowed the embedding and cosine similarity stage to generate cleaner candidates, while the LLM was more frequently able to identify the correct semantic mapping in the first ranking position.

The strong recall observed throughout the experiments demonstrates that the embedding model is generally successful at preserving semantically relevant candidates before the LLM stage. In many repositories, once the correct candidate passes the similarity threshold, the LLM was able to correctly identify the semantic correspondence. This behavior confirms that embedding-based semantic representations can serve as an effective filtering mechanism for cross-language method mapping.

However, despite the strong recall performance, the experiments also revealed important weaknesses regarding semantic rejection and false positive reduction. Although the LLM frequently identified the correct mappings, it also often returned additional incorrect matches. This behavior significantly reduced precision and specificity across many repositories. The problem became especially visible in repositories where methods shared partial contextual structural or naming similarities despite implementing different functionality.

One important observation was that the LLM stage behaved more efficiently as a semantic confirmation mechanism rather than a semantic rejection mechanism. When correct candidates are available, the model often succeeds in recognizing them as valid Type-IV semantic clones. However, when unrelated methods pass the cosine similarity threshold, the LLM frequently attempted to force semantic relationships between methods that did not actually implement equivalent functionality.

This behavior may be related to the nature of modern Large Language Models. LLMs are trained to identify patterns and semantic associations across large datasets, which makes them particularly effective at reasoning about possible similarities. However, this same capability can also lead to overprediction, especially when methods share common terminology, framework usage, or structural patterns. In several repositories utility methods, configuration methods, wrapper functions and helper functions produced semantically weak similarities that were incorrectly interpreted as valid mappings.

The experiments also showed that embeddings alone were insufficient for fully reliable semantic mapping. Although the embedding stage frequently preserved correct mappings, cosine similarity occasionally allowed unrelated methods to pass the threshold. This demonstrates that vector similarity alone cannot perfectly capture functional equivalence, especially in repositories containing many utility or infrastructure related methods.

At the same time, experiments confirm that the combination of embeddings and LLM semantic reasoning is considerably stronger than using embeddings and cosine similarity alone. The LLM was sometimes capable of eliminating incorrect candidates that have passed similarity threshold. In repositories with clear semantic alignment between ecosystems, the combination of embeddings and LLM reasoning produced strong overall results.

Another important observation concerns the relationship between repository structure and semantic complexity. Repositories with highly modular architectures or utility heavy implementations tended to produce a larger number of false positives because many methods shared overlapping contextual characteristics without implementing identical functionality. In contrast, repositories with direct and clearly aligned implementations generally produce much stronger results.

The experiments also demonstrated that repository size and functional similarity strongly influenced the quality of the results. Smaller repositories generally produce better performance compared to larger repositories. Small repositories usually contain fewer unrelated methods, clearer implementation structures, and stronger semantic alignment between ecosystems. As a result, the embedding stage generated cleaner candidate sets, and the LLM stage was less likely to generate incorrect mappings.

Some repositories demonstrated this behavior by achieving higher precision specificity and overall accuracy. In contrast, larger repositories introduced significantly more complexity. Large repositories often contain helper methods, wrappers, utility functions, and partially related implementations that increase the probability of weak semantic similarities between unrelated methods. Consequently, the embedding stage passed frequently unrelated candidates, increasing false positive generation during the LLM stage. As repository complexity increased, precision and specificity decreased because the methodology struggled to confidently reject unrelated methods.

Another important factor was the level of functional similarity between ecosystems. Repositories where Python and JavaScript implementation shared similar objectives generally achieved stronger results because semantically equivalent methods were easier to identify. In repositories with little or no semantic overlap, the methodology frequently generated very large

numbers of false positives because weak similarities were incorrectly interpreted as semantic mappings. The LLM frequently failed to return zero results or label unrelated methods as non-Type-IV semantic clones and instead attempted to generate correspondences between unrelated methods.

Overall, the results demonstrate that the proposed methodology performs effectively when repositories contain strong semantic alignment and moderate complexity, but it still struggles with semantic rejection, false positive reduction and semantic ambiguity in large or functionally unrelated repositories.

6.2 Importance of the Results

The results of this thesis are important because they demonstrate that semantic method mapping across programming ecosystems is possible using modern embedding models and LLMs. As software systems increasingly span multiple ecosystems and programming languages, identifying semantically equivalent functionality becomes an increasingly important problem in software engineering.

Modern software development rarely occurs within a single programming ecosystem. Large scale applications frequently combine backend services, frontend systems, APIs, cloud services, and automation tools written in different languages. Consequently, developers often need to understand how similar functionality is implemented across ecosystems. The proposed methodology contributes directly to this problem by demonstrating that semantic correspondences between methods can be identified.

One important practical implication involves software migration and interoperability. Developers frequently port libraries and frameworks from one ecosystem to another. Developers must manually identify equivalent functionality across implementations. This process is often time-consuming and error-prone, particularly in large repositories. The proposed methodology could assist developers by automatically identifying methods that implement similar functionality across ecosystems, reducing manual analysis effort during migration and maintenance processes.

Another important application involves cross-language software maintenance. Many projects maintain parallel implementations across ecosystems to support multiple developer communities. As these projects evolve independently, developers often attempt to maintain semantic consistency between ecosystems, but this process becomes increasingly difficult over time. Automatic semantic method mapping could help developers identify missing functionalities, locate equivalent implementations, and compare changes across ecosystems.

The results are also highly relevant to software reuse and developer productivity. Developers working on multi-language systems often need to understand how similar functionality is implemented in different ecosystems. Automatically identifying semantic correspondences between methods can simplify repository exploration.

Additionally, the methodology may contribute to semantic code search systems. Traditional code search techniques often rely heavily on textual similarity or keyword matching. However, semantically equivalent methods implemented in different languages may use completely different syntax and naming conventions. Embedding-based semantic representation combined with LLM reasoning provides a way to search for functionality rather than exact textual similarity. This could improve cross-language code retrieval systems substantially.

The results are also important from an academic perspective because they demonstrate both the strengths and limitations of modern AI-based software analysis techniques. The experiments show that embeddings and LLMs can successfully identify semantic correspondences, but they also reveal that semantic reasoning alone is insufficient for fully reliable mapping. This suggests that future systems will likely require hybrid approaches combining embeddings, semantic reasoning, static analysis, dynamic analysis, and execution-based validation.

6.3 Threats to Validity

Several threats to validity should be considered when interpreting the experimental results of this thesis.

First, the evaluation process relied partially on manual semantic inspection and human judgment when constructing ground-truth mappings. Although careful analysis was performed, semantic equivalence between methods may sometimes be subjective, particularly in repositories containing complex utility methods, wrapper functions, or partial functional overlap. Different evaluators could potentially interpret semantic similarity differently in certain cases. Additionally, developer experience and familiarity with different programming languages may vary across evaluators, which could further influence the consistency and reliability of cross-language semantic mappings.

Another important threat concerns the dataset size used for manual evaluation. Although repositories with varying sizes, complexities, and functional similarities were selected, the validation and testing subsets still represent only a portion of possible cross-ecosystem repositories. Consequently, the results do not generalize to all ecosystems, repository structures, or software domains.

The methodology focuses specifically on the Python and JavaScript ecosystems. While these ecosystems provide strong evaluation scenarios (see Section 4.1) for cross-language semantic analysis, the behavior of the methodology may differ when applied to other programming languages, such as Java, C#, Rust, Go, or C++. Differences in syntax, paradigms, and repository organization may influence embedding quality and LLM reasoning behavior.

Another important limitation concerns the use of a fixed cosine similarity threshold. Different repositories contain different levels of semantic diversity and structural complexity. A fixed threshold may therefore incorrectly reject valid semantic mappings or incorrectly allow unrelated candidates to pass the LLM evaluation stage. This behavior directly influences both false positives and false negatives in the experiments.

Additionally, the LLM evaluation stage introduced several limitations. The experimental results showed that the LLM behaved more effectively as a semantic confirmation mechanism rather than a semantic rejection mechanism. In many cases, when unrelated methods passed the embedding threshold, the LLM attempted to force semantic relationships between methods that did not actually implement equivalent functionality. This behavior significantly increased false positive generation and reduced precision and specificity.

Another threat to validity concerns the non-deterministic behavior of Large Language Models. LLM responses may occasionally vary depending on prompt phrasing, candidate ordering, or contextual interpretation. Although structured prompts and JSON-based outputs were used to improve consistency, some variability may still influence the results.

Moreover, different LLMs may produce substantially different semantic reasoning behavior, ranking quality, and false positives depending on their architecture, training data, reasoning capabilities, and code understanding performance. Consequently, the experimental results observed in this thesis may vary when using alternative LLMs or future generations of code-oriented language models.

Repository size and complexity also strongly influenced the performance of the methodology. Evaluation metrics in smaller repositories were stronger because they contained fewer methods and clearer semantic alignments. In contrast, larger repositories introduced significantly more semantic ambiguity, partially related implementations and weak similarities, which increased the number of false positives and reduced precision.

Finally, the methodology focuses mainly on semantic similarity at the method level and does not fully consider runtime behavior, repository evolution, or deeper architectural relationships between implementations. As a result, some semantically equivalent behaviors may remain difficult to identify using only embeddings, cosine similarity and static reasoning. Future approaches could integrate dynamic analysis and execution-based validation to capture deeper behavioral similarities between methods

Section 7

Conclusion

7.1 Conclusion

This thesis investigated the problem of semantic method mapping across Python and JavaScript packages using embeddings and LLMs. The objective was to examine whether semantic similarities between methods implemented in different programming languages can be identified despite syntactic and structural differences.

To achieve this, a complete methodology was developed including dataset collection, repository filtering, method extraction, embedding generation using UniXcoder, cosine similarity, and LLM-based semantic evaluation. The methodology aimed to identify possible Type-IV semantic clones between methods from different ecosystems.

The experimental results demonstrated that embedding-based representations are effective at preserving semantic relationships between methods across programming languages. The LLM evaluation stage was also capable of identifying meaningful semantic correspondences between Python and JavaScript methods.

However, the experiments also revealed some important limitations. The methodology frequently generated false positives, especially in large repositories and repositories with weak functional similarity between ecosystems. The results additionally showed that repository size and complexity strongly influence semantic mapping quality.

Overall, the results suggest that embeddings combined with LLMs provide a promising approach for cross-ecosystem method mapping and semantic software analysis. Furthermore, the proposed methodology may also contribute to future research areas.

7.2 Future Work

One important direction for future work involves integrating execution-based validation techniques into the methodology. The current approach relies mainly on embeddings, cosine similarity, and LLMs. Method execution with automatically generated test cases can be used to compare runtime behavior directly. Such behavioral validation could significantly reduce false positives by ensuring that candidate methods not only appear semantically similar but also produce equivalent outputs and execution behavior. However, the quality of automatically generated tests cases may significantly influence the reliability of execution-based validation, since incomplete or weak test coverage could fail to capture important behavioral differences between methods.

Another promising direction involves improving both the embedding stage and LLM evaluation stage. Future research could investigate alternative embedding models that combine structural, syntactic, and semantic information. Additionally, stronger reasoning LLMs, fine-tuned code models, or advanced prompt-engineering strategies may improve semantic rejection and reduce overprediction behavior. Moreover, adaptive thresholding strategies could also be explored to dynamically adjust cosine similarity thresholds according to repository size and complexity.

Finally, future work could extend the methodology to additional programming ecosystems. Evaluating more ecosystems would provide a broader understanding regarding the scalability and generalization ability of semantic method mapping approaches. Furthermore, the proposed methodology could potentially be adapted as a semantic validation framework for evaluating AI-generated code translations. Since the methodology focuses on semantic equivalence rather than syntactic similarity, it could help determine whether translated methods preserve functionality of the original implementation across programming languages. In this context, the evaluation would not focus on line-by-line translation similarity, but on whether the translated implementation preserves the same overall functionality and semantic behavior with the original method.

Bibliography

[1] C.K. Roy and J.R. Cordy, “A Survey on Software Clone Detection Research,” Technical Report No. 2007-541, School of Computing, Queen’s University, Ontario, Canada, 2007.

[2] A. Eagal, K.T. Stolee, and J.-P. Ore, “Analyzing the Dependability of Large Language Models for Code Clone Generation,” The Journal of Systems & Software, vol. 230, 2025.

[3] Z. Zhang and T. Saber, “Assessing the Code Clone Detection Capability of Large Language Models,” in Proceedings of the International Conference on Code Quality (ICCQ), 2024.

[4] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, “Comparison and Evaluation of Clone Detection Tools,” IEEE Transactions on Software Engineering, vol. 33, no. 9, pp. 577-591, 2007.

[5] X. Qian and E.A. AlOmar, “Can Large Language Models Identify and Refactor Code Clones? An Empirical Study,” The Journal of Systems & Software, vol. 234, 2025.

[6] A.K. Mohammed, D. Fallah, S.A. Azize, B. Al-Attar, A.H. Ahmad, N. Divekar, N.B. Pokale, S.A. Mahdi, and R. Sekhar, “Cross-Language Semantic Code Clone Detection in Large-Scale Software Repositories Using Transformer-Based Contrastive Learning,” in Proceedings of the IEEE Conference, 2025.

[7] Y. Liang, R. Ying, T. Taniguchi, G. Lyu, and Z. Cui, “HyClone: Bridging LLM Understanding and Dynamic Execution for Semantic Code Clone Detection,” arXiv preprint arXiv:2508.01357, 2025.

[8] M. Khajezade, J.J.W. Wu, F.H. Fard, G. Rodríguez-Pérez, and M.S. Shehata, “Investigating the Efficacy of Large Language Models for Code Clone Detection,” in Proceedings of the ACM/IEEE Conference, 2024.

[9] M.B. Moumoula, A.K. Kabore, J. Klein, and T. Bissyande, “Large Language Models for Cross-Language Code Clone Detection,” in Proceedings of the ACM/IEEE Conference, 2025.

[10] A.S. Alshabib, S. Mahmood, and M. Alshayeb, “A Systematic Literature Review on Cross-Language Source Code Clone Detection,” Computer Science Review, vol. 58, p. 100786, 2025.

[11] P.R. Roy, A.I. Alam, F. Al-Omari, B. Roy, C.K. Roy, and K.A. Schneider, “Unveiling the Potential of Large Language Models in Generating Semantic and Cross-Language Clones,” in Proceedings of the IEEE/ACM Conference, 2024.

[12] C. Teyton, J.-R. Falleri, and X. Blanc, “Automatic Discovery of Function Mappings between Similar Libraries,” in Proceedings of the IEEE Conference, 2013.

[13] E. Constantinou, A. Decan, and T. Mens, “Breaking the Borders: An Investigation of Cross-Ecosystem Software Packages,” in Proceedings of the IEEE International Conference, 2018.

[14] H. Li and C.-P. Bezemer, “Bridging the Language Gap: An Empirical Study of Bindings for Open Source Machine Learning Libraries Across Software Package Ecosystems,” Empirical Software Engineering, vol. 29, p. 153, 2024.

[15] X. Gu, H. Zhang, D. Zhang, and S. Kim, “DeepAM: Migrate APIs with Multi-modal Sequence to Sequence Learning,” in Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), 2017.

[16] R. Pandita, R.P. Jetley, S.D. Sudarsan, and L. Williams, “Discovering Likely Mappings between APIs using Text Mining,” in Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), 2015.

[17] H. Zhong, S. Thummalapenta, and T. Xie, “Exposing Behavioral Differences in Cross-Language API Mapping Relations,” in Fundamental Approaches to Software Engineering (FASE), Lecture Notes in Computer Science, vol. 7793, pp. 130-145, Springer, 2013.

[18] K. Kannee, R.G. Kula, S. Wattanakriengkrai, and K. Matsumoto, “Intertwining Communities: Exploring Libraries that Cross Software Ecosystems,” in Proceedings of the

IEEE/ACM International Workshop on Software Engineering for Systems-of-Systems and Software Ecosystems, 2023.

[19] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, “Mining API Mapping for Language Migration,” in Proceedings of the International Conference on Software Engineering (ICSE), Cape Town, South Africa, pp. 195-204, 2010.

[20] J. Huhtamäki, R.C. Basole, K. Still, M.G. Russell, and M. Seppänen, “Visualizing the Geography of Platform Boundary Resources: The Case of the Global API Ecosystem,” in Proceedings of the 50th Hawaii International Conference on System Sciences (HICSS), 2017, pp. 5305-5314.

[21] N.D.Q. Bui, Y. Yu, and L. Jiang, “SAR: Learning Cross-Language API Mappings with Little Knowledge,” in Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE ’19), Tallinn, Estonia, 2019, pp. 796-806.

[22] UniXcoder, “UniXcoder: Unified Cross-Modal Pre-training for Code Representation,” Microsoft Research, 2022. Available: https://github.com/microsoft/CodeBERT

[23] Ollama, “Ollama Documentation,” Ollama. Available: Ollama Documentation

[24] Hugging Face, “Transformers Documentation,” Hugging Face Documentation. Available: Transformers Documentation

[25] CLOC, “Count Lines of Code (CLOC) Documentation,” CLOC Official Documentation. Available: CLOC Documentation

[26] Abstract Syntax Trees (AST), “Python AST Documentation,” Python Software Foundation. Available: Python AST Documentation

[27] Babel Team, “Babel Parser Documentation,” Babel Official Documentation. Available: Babel Parser Documentation