

An LLM-based Technique for Detection of Code Clones Across Open Source Microservice Repository Forks

Jamie Kemman

Shepherd University

Shreya Nakum

University of California, Irvine

Alexander Bakhtin

University of Oulu

Matteo Esposito

`esposito@imada.sdu.dk`

University of Southern Denmark

Tomas Cerny

University of Arizona

Valentina Lenarduzzi

University of Southern Denmark

Davide Taibi

University of Southern Denmark

Research Article

Keywords: code clones, microservices, semantic analysis, large language model, retrieval-augmented generation, open source software

Posted Date: October 21st, 2026

DOI: <https://doi.org/10.21203/rs.3.rs-11181478/v1>

License: © ⓘ This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Additional Declarations: No competing interests reported.

An LLM-based Technique for Detection of Code Clones Across Open Source Microservice Repository Forks

Jamie Kemman^{1†}, Shreya Nakum^{2†}, Alexander Bakhtin³,
Matteo Esposito⁴, Tomas Cerny⁵, Valentina Lenarduzzi^{4,3},
Davide Taibi^{4,3}

¹Shepherd University, Shepherdstown, USA.

²University of California, Irvine, USA.

³University of Oulu, Oulu, Finland.

⁴University of Southern Denmark, Vejle, Denmark.

⁵University of Arizona, Tucson, USA.

Contributing authors: jamieruth@proton.me; snakum@uci.edu;
alexander.bakhtin@oulu.fi; esposito@imada.sdu.dk; tcerny@arizona.edu;
lenarduzzi@imada.sdu.dk; taibi@imada.sdu.dk;

[†]These authors contributed equally to this work.

Abstract

Aim: Understanding code reuse and divergence in open-source microservice projects is crucial to assessing software evolution and community-driven software development. As repositories are forked and modified, it becomes difficult to trace how much of the original code persists and whether or not these forks create a unique project or retain the same functionality as the original. In this study, we leveraged Large Language Models to identify code clones and analyzed code clone distribution in open-source microservice projects.

Method: We used a RAG framework based on LLMs to semantically compare code at the service level. This allowed us to detect not only direct clones, but also functionally equivalent segments that have been refactored. In order to identify a feasible LLM pipeline for analyzing clone types and distribution, we tested three candidate LLMs and four prompt techniques on the IBM CodeNet dataset of Type-4 clones. We then used the best-performing pipeline to identify the code clones in pairs of primary and forked open-source microservices projects.

Results: We found the code-specific Codestral-22B LLM with a few-shot continuous prompting technique to be the most accurate in detecting Type-4 code clones, with a precision of 0.74 but a recall of only 0.60. Our analysis of 22 microservice repositories with this technique showed that Type-1 clones are most prevalent, followed by Type-3 clones. Semantic comparison revealed patterns of reuse and divergence not captured by syntax-based tools.

Conclusion: Semantic code analysis using LLMs provides deeper understanding of code evolution in forks of microservice projects. Code-specific LLMs significantly outperformed multimodal models but remain unsuitable for production use due to high false negative rates.

Keywords: code clones, microservices, semantic analysis, large language model, retrieval-augmented generation, open source software

1 Introduction

Open-source software (OSS) has become the foundation of modern software development, allowing for collaborative innovation, rapid prototyping, and widespread reuse. OSS projects are frequently forked, adapted, and restructured to suit other developers’ needs. Understanding code evolution in forked projects is essential to evaluate the original’s complexity and malleability (Rattan et al. (2013)), as between 5% and 20% of software systems contain duplicated code (Roy and Cordy (2007)).

Traditional code clone detection tools like pure textual comparisons, token-based analysis, and abstract syntax tree comparisons can identify exact copies but often fail to detect semantically similar code (Type-4 clones, Rattan et al. (2013)) that has been completely refactored (Svajlenko and Roy (2014)). This limitation is particularly evident in microservice systems (OSS MSS), where code modules can be merged or otherwise reorganized into new services while retaining original functionality.

Many studies have demonstrated that Large Language Models (LLMs) can understand and generate human-like language (Chang et al. (2023)), while also reasoning and explaining machine-readable code. With the growth of LLMs trained specifically for understanding code, there is now a possibility to analyze code similarity at a higher level, as done with CodeBERT (Feng et al. (2020)).

In this work, we investigated the performance of LLMs in detecting code clones and leveraged the best-performing model and prompting technique to assess the distribution of code clones in forks of 22 OSS MSS repositories.

We found that larger and code-specific models demonstrated higher accuracy (Codestral, Devstral), while smaller and general-purpose LLM performed worse (Llama3-8B). In addition, our findings uncovered that LLMs tend to have a high number of false negatives, while few-shot prompts with structured, numerical outputs performed well. Moreover, in OSS MSS repositories, Type-1 and Type-3 clones are more common than expected, while Type-2 and Type-4 clones are less common.

Our contributions are as follows:

- We analyzed the performance of LLMs and embedding models for detecting Type-4 code clones in the CodeNet dataset;
- We identified the best LLM and prompting strategy for detecting code clones;
- We determined that LLM-based code clone detection performs better than mere embedding-based;
- We determined that code-specific LLMs are better at detecting code clones than general-purpose LLMs;
- We identified 22 OSS MSS repositories and 22 respective forks suitable for fork analysis and code clone detection;
- We used the best-performing LLM pipeline for code clone detection to determine the distribution of code clone types specifically in microservice projects.

Paper Structure: Section 2 presents the background and definitions. Section 3 describes the methodology. Section 4 presents the obtained results, and Section 5 discusses them, while Section 6 summarizes related work. Section 7 highlights threats to the validity of our study, and Section 8 draws the conclusion.

2 Background & Definitions

This section gives background research, terminology, and definitions relevant to our work.

2.1 Definitions

In this work, we define **primary repository** as the original repository, as opposed to a **fork**, which is a copy of the repository developed in parallel. We define **file** as collection of code present in a GitHub repository. We define **snippet** and **chunk** to be a section of a file or the entirety of a file within a repository. A snippet and chunk are a part of a file, consisting of up to 20 lines of code in a file.

2.1.1 Code Clone Analysis

In our work, we adopt the definitions of **code clones** from [Rattan et al. \(2013\)](#), who provide a widely adopted classification of code clones based on a synthesis of prior studies focused on software clones and clone detection:

Type-1 Exact copies of the primary repository.

Type-2 Exact copies except for variable names, layout adjustments, and comments.

Type-3 Non-exact copies with line insertions or deletions.

Type-4 Semantic clones with identical function but differing syntax.

Using a standard systematic review process, the authors further classified these articles into the subtopics of clone detection, clone management, semantic clone detections, and model-based clone detection. Although the literature review delves further into specifics of various analysis methods, we found these four basic clone types to easily adapt to the needs of the natural-language prompt engineering used in our work.

2.1.2 Code-Focused Retrieval Augmented Generation (RAG)

Vector embedding is the process of transitioning human-readable raw data into a common multi-dimensional vector space. Data with similar attributes will inherently cluster into neighboring vectors, enabling search agents to quickly retrieve similar results (Günther et al. (2024)).

Embedding models handle the embedding process, and are available in both general and data-specific variants. Sentence embedding models are available for general text embedding application and are capable of capturing the semantic meaning of syntactically disparate text samples (Feng et al. (2020)). The need for accurate semantic embedding of long code sequences, which differ from plain language, has led to the development of multilingual, code-specific model architectures such as CodeBERT.

Once translated to vector space, embedded documents may be stored and queried in similar fashion to traditional database structures. **Retrieval augmented generation (RAG)** is employed when a language model is instructed to query an existing vector database during inference and input the response (Lewis et al. (2021)). When performing inference on a codebase, accessing embedded data allows an LLM to directly assess for semantic shifts while ignoring syntactic differences.

3 Study Design

We designed our empirical study according to the guidelines by Wohlin et al. (2024).

3.1 Goal and research questions

We posed the **goal** of this study as follows: **analyze** the effectiveness of LLMs in code clone detection **with the purpose of** assessing clone type distribution **from the point of view** of microservice architecture.

To achieve this goal, we conducted a two-phase empirical study. First, we evaluated three LLMs on the CodeNet dataset to determine their effectiveness in Type-4 clone detection. Second, we applied the best-performing model to analyze the code clone type distribution in 22 pairs of OSS MSS repositories and their forks.

Correspondingly, we formulated two Research Questions (**RQs**):

RQ₁ How effective are LLMs in identifying code clones?

Prior research uncovered that reasoning and multimodal LLMs outperformed generic and coding LLMs. (Haroon et al. (2025)). To find the effectiveness of LLMs on this problem, we used IBM’s CodeNet Project dataset of 10,000 C++ code pairs labeled as Type-4 clones or non-clones (Puri et al. (2021)).

For this study, we postulated the following **null hypotheses** and the following **alternative hypotheses**:

H_{01} : There is **no** difference in the accuracy of LLMs for each prompt.

H_{02} : There is **no** difference in the accuracy of LLMs and the embedding model.

H_{03} : There is **no** difference in the accuracy of code-specific and multimodal LLMs.

H_{11} : There is a difference in the accuracy of LLMs for each prompt.

H_{12} : There is a difference in the accuracy of LLMs and the embedding model.

H_{13} : There is a difference in the accuracy of code-specific and multimodal LLMs.

Systematically identifying clone types and their incidence rates is the key to any effective clone management suite for developers and maintainers. As languages and frameworks evolve, the ability to detect across the spectrum of clone types and continuously evaluate for change will also lessen the effort required to refactor and apply OSS MSS software to any given use case. Assuming we can successfully construct an LLM framework for identifying code clones in **RQ₁**, we can study specifically the code clones in OSS MSS repositories:

RQ₂ What types of code clones are most common in forks of OSS MSS?

We applied each LLM to the RAG-embedded representations of 22 primary and fork GitHub repository pairs to analyze the code clone types in the fork repository relative to the primary repository. We considered one **null hypothesis** and a corresponding **alternative hypothesis**:

H_{04} : The distribution of code clone types in OSS MSS forks is uniform.

H_{14} : The distribution of code clone types in OSS MSS forks is **not** uniform.

We used a χ^2 goodness-of-fit test for the uniformity of distribution to determine if the code clones' distribution is significantly non-uniform. Moreover, we attempted to uncover relationships between each code clone type using Pearson residuals. Finally, we compared the RAG output on three OSS MSS repositories with human-validated results.

3.2 Study Context

The context of this study includes the repositories used for code clone detection (**RQ₁**), as well as the curated OSS MSS repositories used for clone type distribution analysis specifically in microservices (**RQ₂**).

3.2.1 CodeNet dataset (**RQ₁**)

We used the CodeNet dataset by [Puri et al. \(2021\)](#) for code clone analysis. CodeNet contains 10,000 C++ code pairs, with 5,000 labeled as Type-4 clones and 5,000 as non-clones. Due to computational limitations, we conducted initial testing on subsets of 1,000 pairs for prompt optimization, then validated the best-performing configuration on the complete dataset.

3.2.2 Curated OSS MSS repositories (RQ₂)

We curated a dataset of 22 primary-fork OSS MSS repository pairs from two sources: 17 from [Rahman et al. \(2019\)](#) and 5 from [Yang et al. \(2024\)](#).

Our repository selection criteria included: (1) diversity of programming languages; (2) variety of architectural differences between primary and fork repositories; (3) range of modifications (commits ahead/behind).

For repository pairing, we chose one fork repository per primary repository from the list of publicly visible forks. We then curated fork repositories to obtain a diversity of differences between the primary and the fork.

Among the primary repositories, 18 have Java as the primary language, with the remaining 8 using TypeScript, JavaScript, C#, Go, Python, or Bicep as the primary language. Considering the forks, fork divergence ranged from minimal changes (< 10 commits difference) in 13 of the repositories to large changes in another 13 repositories (10-1938 commit difference), with one case being a complete refactoring, i.e., Go/Python to C# migration.

3.3 Data collection

Data collection included iteratively testing LLMs and prompts for code clone detection (RQ₁) and codebase RAG embedding for the analysis of OSS MSS code clones (RQ₂).

3.3.1 Prompt engineering (RQ₁)

Prompt engineering is a method to design specific input sentences to get a specific output from an LLM ([Liu and Chilton 2022](#)). We employed four prompting techniques and measured the Llama3-8B model performance with each of them on the CodeNet dataset. We chose the Llama3-8B model since it is multimodal and documented in previous studies ([Moumoula et al. 2025](#)). We iteratively developed four prompts to optimize LLM clone detection performance. Each prompt instructed the model to classify code pairs into five categories: Non-clone, Type-1, Type-2, Type-3, or Type-4.

We tested the following prompts in increasing order of guidance given to the LLM:

1. Zero-shot: Basic definitions with yes/no output.
2. Chain-of-Thought: Added reasoning step before classification.
3. Few-shot Discrete: Included examples with yes/no output.
4. Few-shot Continuous: Examples with confidence scores (0-1).

Complete prompt text is available in Appendix A Table A1.

The final prompt (Few-shot Continuous) requested JSON-formatted output with numerical confidence scores for each clone type, which enabled finer-grained analysis than binary classification: {"Type-1": 0.85, "Type-2": 0.20, "Type-3": 0.10, "Type-4": 0.05}.

3.3.2 LLMs for code clone detection (RQ₁)

After finalizing the prompt, we conducted testing with the CodeNet on the three selected LLMs (Table 1). We ran six iterations of testing with the CodeNet dataset. The testing procedure was as follows:

- Codestral-22B: 5 iterations on 1,000-pair subsets, 1 iteration validated on full 10,000;
- Devstral-24B: 2 iterations on 1,000-pair subsets, 4 iterations validated on full 10,000;
- Llama3-8B: 3 iterations (ranging between 1,000-8,000 pairs), 3 iterations validated on full 10k;
- Jina Embedding Baseline: 6 iterations on full 10,000-pair dataset.

The Llama3-8B model is multilingual and contains a long context window, having been trained on about 15 trillion data tokens, and is capable of writing code and solving reasoning problems. (Grattafiori et al. (2024)). The Codestral model was trained on over 80 programming languages and performs as well as Llama3-70B in the Spider benchmark¹. The Devstral model is trained on a dataset of GitHub issues to write functions and code². Both Codestral and Devstral have much larger parameters than Llama3-8B, so they are able to accommodate more knowledge and outperform Llama (Kaplan et al. (2020)).

Table 1: Comparison of LLM characteristics and hyperparameters.

Characteristic	Devstral-24B	Codestral-22B	Llama3-8B	Jina Embeddings V2 (Baseline)
Parameters	24B	22B	8B	161M
Type	Code-specific	Code-specific	Multimodal	Embedding only
Training Focus	GitHub issues, function writing	80+ languages	General + code (15T tokens)	Code-specific
Context Window	32k tokens	32k tokens	8k tokens	8k+ tokens
Temperature	0.8	0.8	0.8	N/A
Seed	0	0	0	N/A
Top-p	0.9	0.9	0.9	N/A
Top-k	50	50	50	N/A

Across each of the three LLMs, we attempted to run six iterations of the CodeNet dataset. We had access to the CSC Mahti³, where we were able to run two of three LLMs, but we had to test the third model locally. Due to these computational limitations, the locally run model had only one iteration on the complete CodeNet dataset, while the remaining five iterations are of size 1000. The other two LLMs had at least three iterations with the complete CodeNet dataset. Across all iterations, we maintained an equal ratio of non-semantic and semantic code clones. Computational limitations prevented full-dataset testing for all models during prompt optimization, but all final comparisons used the complete CodeNet dataset. To answer **RQ₁**, we evaluated model performance using standard information retrieval metrics (Table 2).

Moreover, Appendix Table A2 shows the preliminary results from other models not included in this work, which were tested on a subset of CodeNet that can be analyzed without a supercomputer.

¹<https://mistral.ai/news/codestral/>

²<https://mistral.ai/news/devstral/>

³<https://csc-guide-preview.2.rahtiapp.fi/origin/qiskit-singularity/computing/systems-mahti/>

Table 2: Evaluation metrics.

Metric	Formula
Precision	$\frac{TP}{TP+FP}$
Recall	$\frac{TP}{TP+FN}$
F1 Score	$\frac{2 \times Precision \times Recall}{Precision + Recall}$
Accuracy	$\frac{TP+TN}{TP+TN+FN+FP}$
MSE	$\frac{1}{n} \sum_{i=1}^n (Truth_i - Prediction_i)^2$

3.3.3 Code Embeddings for OSS MSS projects (RQ₂)

Once paired, the primary and fork repositories (Section 3.2.2) are cloned locally in their entirety, embedded, and then broken into 22 individual database files, where each database contains three collections for each embedding model. Each collection holds a vector embedding, code content, and a file ID for each file. Embeddings and associated metadata are chunked during processing and then reassembled into whole representations of individual files. For embedding generation, we use three separate code-specific embedding models that support a different range of programming languages (Table 3):

- **Jina Embeddings V2 Base Code**⁴: 161M parameter BERT derivative trained on the github-code dataset and a proprietary dataset of code question answers. (Günther et al. (2024)).
- **Nomic CodeRankEmbed**⁵: 137M parameter bi-encoder trained on the CoRN-Stack contrastive dataset of 6 programming languages derived from The Stack V2 (Suresh et al. (2025)).
- **CodeSage Large V2**⁶: 1.3B parameter transformer encoder trained across a contrastive dataset of 9 languages derived from The Stack V1 (Zhang et al. (2024)).

To identify code clones, we prompted each LLM with the few-shot (continuous) prompt (Section 3.3.1) and two code snippets, where the former snippet is from the primary repository and the latter from the respective fork. In order to determine the code snippet pairs, we employed cosine similarity between the embeddings of each code chunk in the repositories. Due to the training limitations of our selected embedding models, certain languages present in our OSS MSS repository dataset were not covered, such as HCL, leading to embedding failures. Thus, such code snippets were excluded from analysis.

3.3.4 Human Validation of OSS MSS code clones (RQ₂)

To answer RQ₂, we measured the distribution of code clone types by running the LLM pipeline on our OSS MSS dataset of 22 projects, then chose three projects as a subset for a human validator to make manual comparisons. We limited the scope

⁴<https://huggingface.co/jinaai/jina-embeddings-v2-base-code>

⁵<https://huggingface.co/nomic-ai/CodeRankEmbed>

⁶<https://huggingface.co/codesage/codesage-large-v2>

Table 3: Programming languages supported by embedding models.

Language	Jina V2 Base	Nomic CodeRank	CodeSage Large V2
Assembly	✓	-	-
Batchfile	✓	-	-
C	✓	-	✓
C#	✓	-	✓
C++	✓	-	-
CMake	✓	-	-
CSS	✓	-	-
Dockerfile	✓	-	-
FORTRAN	✓	-	-
Go	✓	✓	✓
Haskell	✓	-	-
HTML	✓	-	-
Java	✓	✓	✓
JavaScript	✓	✓	✓
Julia	✓	-	-
Lua	✓	-	-
Makefile	✓	-	-
Markdown	✓	-	-
Perl	✓	-	-
PHP	✓	✓	✓
PowerShell	✓	-	-
Python	✓	✓	✓
Ruby	✓	✓	✓
Rust	✓	-	-
Scala	✓	-	-
Shell	✓	-	-
SQL	✓	-	-
TeX	✓	-	-
TypeScript	✓	-	✓
Visual Basic	✓	-	-
Total	30	6	9

of human validation to three projects in an attempt to meet time constraints while maintaining a limited variety of languages. The human validation was performed by viewing the online GitHub diff of each project’s primary and fork repositories. Files with no differences were classified as Type-1 clones, while files with differences were reviewed line-by-line and classified accordingly. Files that were novel additions to a fork with no corresponding code in the primary repository were excluded.

3.4 Data Analysis

This section presents the data analysis procedure used to answer the research questions.

For **RQ₁**, before selecting statistical tests, we assessed data distribution normality using the Shapiro-Wilk test ([Shapiro and Wilk \(1965\)](#)). Model accuracy metrics violated normality assumptions ($W = 0.234$, $p = 0.0007$), necessitating non-parametric methods. Prompt accuracy metrics also violated normality assumptions ($W = 0.5739$,

Table 4: Average Prompt Results on CodeNet dataset with Llama3-8B ($\mathbf{RQ}_1$, H_{01}).

Prompt Name	Clone Type	Precision	Recall	F1-score	MSE	Accuracy
Zero-shot	Overall	0.4733	0.01	0.02	0.5	0.5
	Non-clone	0.50	0.99	0.66		
	Clone	0.4733	0.01	0.02		
Chain of Thought	Overall	0.00	0.00	0.00	0.50	0.50
	Non-clone	0.50	1.00	0.67		
	Clone	0.00	0.00	0.00		
Few-shot (Discrete)	Overall	0.00	0.00	0.00		0.52
	Non-clone	0.55	0.90	0.68		
	Clone	0.00	0.00	0.00		
Few-shot (Continuous)	Overall	0.6333	0.1617	0.2567	0.4583	0.5417
	Non-clone	0.5267	0.9117	0.6650		
	Clone	0.6333	0.1617	0.2567		

$p \approx 0$). We thus tested H_{01} , H_{02} , and H_{03} via the Wilcoxon signed-rank test (Wilcoxon (1945)), which is a generic test that does not assume normal data distribution. WT utilizes the absolute difference between paired data and compares the sum of the positive and negative differences, taking the minimum to be the test statistic. In testing each hypothesis (H_{01} - H_{03}), we set our alpha value to 0.05. We decided on this conservative estimate to reduce the chance of false positives when performing multiple statistical tests on the same CodeNet dataset.

For clone type distribution analysis ($\mathbf{RQ}_2$), when testing H_{04} , we used the χ^2 goodness-of-fit test with $\alpha = 0.05$ to assess if the code clone type distribution is uniform in OSS MSS repositories, or if one type of code clones is more common among OSS MSS. The χ^2 goodness-of-fit test is used to determine if the proportions of different types of observations deviate significantly enough from being equally proportionate. Moreover, we computed Pearson residuals (Cordeiro (2004)) to determine the strength of the deviation from uniformity. Furthermore, we computed the Cramer’s V ($\phi_c = 0.311$), which indicated a moderate to large effect size for the deviation of clone type proportions from a uniform distribution (Cramér (1999)).

4 Results

This section presents the results used to answer the $\mathbf{RQ}$ s.

4.1 Performance of LLMs in code clone detection ($\mathbf{RQ}_1$)

In this research question, we wanted to determine whether LLMs are able to successfully perform semantic code clone detection based on a variety of prompts.

We tested H_{01} to assess the best prompt using pairwise Wilcoxon tests across all four prompts. We rejected the null hypothesis (all $p < 0.05$), confirming significant differences in prompt effectiveness. The Few-shot (Continuous) prompt significantly

Table 5: Average Model Results on CodeNet Dataset ($\mathbf{RQ}_1$, H_{02} - H_{03}).

Model	Clone type	Precision	Recall	F1-score	MSE	Accuracy
Devstral-24B LLM (code-specific model)	Overall	0.65	0.5183	0.5767	0.3733	0.6267
	Non-clone	0.6117	0.7317	0.6667		
	Clone	0.65	0.5183	0.5767		
Codestral-22B LLM (code-specific model)	Overall	0.8433	0.7	0.7367	0.3917	0.655
	Non-clone	0.5983	0.7650	0.7050		
	Clone	0.74	0.6	0.64		
Llama3-8B LLM (general model)	Overall	0.6333	0.1617	0.2567	0.4583	0.5417
	Non-clone	0.5267	0.9117	0.6333		
	Clone	0.6333	0.1617	0.2567		
Jina Embeddings (baseline model)	Overall	0.54	0.60	0.57	0.46	0.54
	Non-clone	0.55	0.48	0.51		
	Clone	0.54	0.60	0.57		

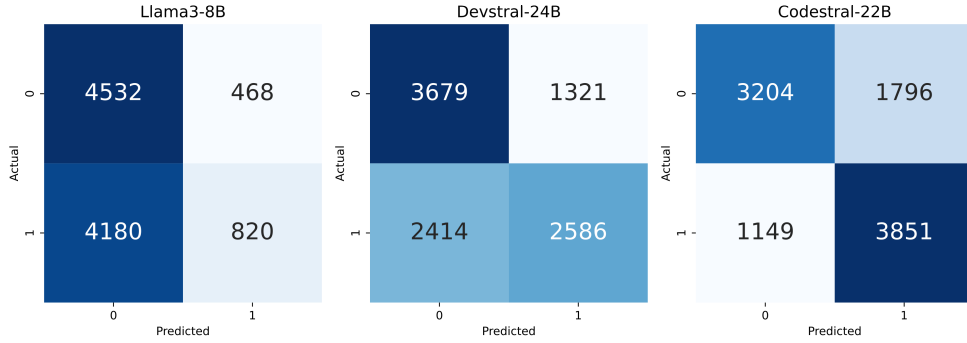


Fig. 1: Confusion matrices for three LLMs on the CodeNet dataset (n=10,000 pairs). All models exhibited high false negative rates (40-85%), with Llama3-8B showing the most conservative classification bias (91% of clones misclassified as non-clones). Codestral-22B achieved the best balance but still produced 40% false negatives.

outperformed all others: vs. Zero-shot ($p = 0.03125$), vs. CoT ($p = 0.03125$), vs. Few-shot Discrete ($W = 159$, $p = 0.03125$). Table 4 shows that the Chain of Thought and Few-shot (Discrete) prompts exhibited poor precision. Nonetheless, **all prompts show fair accuracies, within 0.50 and 0.55.**

The Few-shot (Continuous) prompt **significantly outperforms** all others.

Table 5 shows the average CodeNet metrics for all three LLMs. The models exhibit adequate Accuracy and Precision. However, the Llama3-8B model underperformed, predicting Non-clones far more than clones, leading to a low precision and recall for Clones. In contrast, Devstral and Codestral showed better results for true positive

classification despite also showing a high number of false negatives. Nonetheless, **all models showed high values for Recall in Non-clones, all above 0.73.**

We tested H_{02} to determine if using an LLM is effective. If a naïve embedding model performs with a higher accuracy than an LLM, then an LLM is redundant for this kind of tasks. We tested H_{02} using pairwise Wilcoxon signed-rank tests. We rejected the null hypothesis for Devstral-24B vs. Jina ($p = 0.03125$) and Codestral-22B vs. Jina ($p = 0.03125$), indicating these LLMs significantly outperformed the baseline embedding model. However, we failed to reject H_{02} for Llama3-8B vs. Jina ($p = 1.0$), suggesting no significant difference in performance.

Devstral and Codestral **LLMs achieve better performance** than a naïve embedding approach with Jina.

We tested H_{03} to determine if a generalized LLM could be used for code-specific problems or if a code-specific model performs better. We tested H_{03} using pairwise Wilcoxon tests comparing code-specific models (Devstral, Codestral) against the multimodal model (Llama3-8B). We rejected the null hypothesis in all comparisons: Devstral vs. Llama ($p = 0.03125$), Codestral vs. Llama ($p = 0.03125$).

Code-specific LLMs (Devstral, Codestral) **significantly outperform** the multimodal LLM (Llama3-8B).

Finally, the confusion matrix shows the results of the first iteration of CodeNet testing (Figure 1). We observe that false negatives are common across all 3 LLMs. But, the false negatives were least seen in the Codestral model, which reported adequate performance through its true positive and true negative rates. Despite this, Codestral performed questionably in terms of false positives, as that rate was the highest among all three models; although Codestral performed well overall, having more false positives than false negatives was unsatisfactory. However, in conjunction with Table 5 and the WT results, the larger parameter models of Codestral and Devstral performed better than Llama.

Codestral-22B achieved the **highest accuracy** (65.5%) and was superior to both Devstral-24B (62.7%, $p = 0.03125$) and Llama3-8B (54.2%, $p < 0.05$).

We therefore selected Codestral-22B for OSS MSS repository analysis.

4.2 OSS MSS Code Clone Distribution (RQ_2)

This research question analyzed the distribution of code clone types across forks of OSS MSS repositories and determined the most common type.

Figure 2 shows the distribution of code clone types for a selection of forks. Based on this subset, Type-1 and Type-3 clones are most common among forks of the surveyed repositories. Moreover, we can look at how different embedding models led to different

code clone distribution results across all repositories in Figure 3. In this figure, the second and third embedding models were trained on adjacent versions of the same dataset, leading to similar predictions. However, the first embedding model was not, leading to different predictions overall. Nomic CodeRankEmbed and CodeSage Large V2, trained on adjacent versions of The Stack dataset, produced similar distributions with high Type-1 detection. Jina Embeddings V2, trained on different data (github-code), detected fewer Type-1 clones but more Type-3 and Type-4, suggesting training data significantly influences semantic similarity assessment

We tested H_{04} using a χ^2 goodness-of-fit test. We strongly rejected the null hypothesis ($\chi^2 = 44666$, $df = 3$, $p < 2.2 \times 10^{-16}$), confirming that clone type distribution is not uniform with a large effect size (Cramér’s $V = 0.311$), indicating substantial practical significance. Type-1 clones occurred 69% more frequently than under uniform distribution, while Type-2 clones occurred 73% less frequently (Table 6).

There is a **statistically significant difference** in code clone distribution for surveyed OSS MSS forks.

Additionally, based on the Pearson residuals (Table 6), we can see that Type-1 had a large positive residual (+135.15) and Type-3 had a positive residual (58.93), showing that they are predicted more than expected in OSS MSS repositories. Furthermore, Type-2 had a notably large negative residual (-142.27), while Type-4 had a negative residual (-51.82), indicating underrepresentation compared to uniform distribution. Together, these residuals show that Type-1 and Type-3 clones are over-represented, while Type-2 and Type-4 clones are under-represented, relative to what would be expected if all clone types occurred with equal frequency. The Type-1/Type-2 imbalance accounts for the majority of the overall χ^2 statistic, with the Type-3/Type-4 imbalance contributing a smaller, secondary effect.

Finally, Table 7 compares the Codestral-22B code clone outputs per each embedding model with the human validation on three OSS MSS repository pairs. Here, the Nomic and CodeSage models’ counts were closer to the human than the Jina Embedding model’s count. The code-specific version of Jina Embeddings V2 was pretrained on the github-code dataset, which contains no exact duplicates and ignores whitespace, and therefore would contain few Type-1 clones, if any (CodeParrot (2022); Günther et al. (2024)). This may account for the large disparity between Codestral’s analysis using the Jina embedding model as compared to the human validator.

Type-1 and Type-3 clones were **most common** in OSS MSS repositories.

5 Discussion

This section discusses the implications of the results. We provide take-away messages for each key finding.

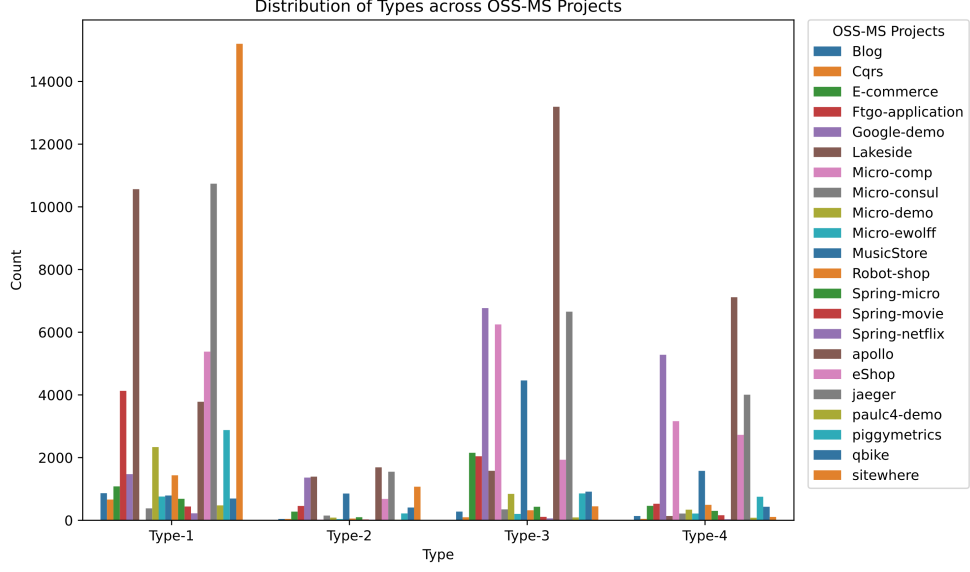


Fig. 2: Distribution of code clone types across selected OSS MSS fork pairs (RQ_2).

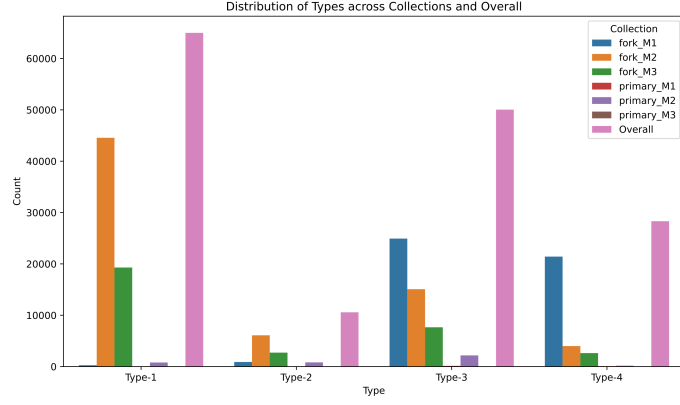


Fig. 3: Clone type distribution across all 22 repository pairs using the Devstral-24B (RQ_2).

5.1 LLM Accuracy for code clone detection

We found that LLMs were at least as accurate as, or more accurate than the baseline embeddings model. Nonetheless, according to Table 5, LLMs still did not achieve excellent accuracy. In context of identifying code clones, we prefer to have adequate precision and recall in both Non-clone and Clone classes. However, the models tended to perform better in identifying Non-clones than Clones and tend to produce many

Table 6: Observed counts, expected counts and Pearson residuals for the code clone types in OSS MSS projects ($\mathbf{RQ}_2$).

	Type-1	Type-2	Type-3	Type-4
Observed	64993	10572	50041	28316
Expected	38480.5	38480.5	38480.5	38480.5
Pearson Residuals	135.15	-142.27	58.93	-51.82

Table 7: Human Validation on OSS MSS Repositories ($\mathbf{RQ}_2$)

Blog Microservices				
	Type-1	Type-2	Type-3	Type-4
Human	688	0	0	0
Jina Embeddings V2 Base Code	2	1	253	117
Nomic CodeRankEmbed	440	21	14	8
CodeSage Large V2	423	22	11	12
Ewolfo/Microservice				
	Type-1	Type-2	Type-3	Type-4
Human	655	0	15	0
Jina Embeddings V2 Base Code	1	0	150	183
Nomic CodeRankEmbed	369	23	27	13
CodeSage Large V2	388	16	27	20
Spring Cloud Movie Recommendations				
	Type-1	Type-2	Type-3	Type-4
Human	391	0	7	0
Jina Embeddings V2 Base Code	14	10	78	149
Nomic CodeRankEmbed	206	7	15	9
CodeSage Large V2	220	10	17	4

false negatives (Figure 1). Code-specific Mistral AI models (Devstral and Codestral) performed the best, significantly better than Meta’s general-purpose Llama model.

Our findings align with Moumoula et al. (2025), who reported false negative rates of 55-70% for Starcoder2-15b-instruct and Starchat- β on CodeNet as those models tended to consider most pairs as Non-clones, as well as Dou et al. (2023) where authors automated code clone detection through various LLMs, prompts, and thresholds and found that larger models performed better than smaller ones.

Key Takeaway: LLMs efficiently identify non-clones but exhibit 40-85% false negative rates for Type-4 semantic clones, limiting their reliability for clone detection in production.

5.2 Code Clones in OSS MSS

We found that Type-1 and Type-3 were the most common code clones in OSS MSS repositories’ forks. However, the embedding models trained on fewer languages led to higher Type-1 clone detection than the embedding model that was trained on more languages (Figure 3). Moreover, the Jina embeddings model tended to deviate significantly from human-validated counts for a subset of three OSS MSS repositories’ forks, as seen in Table 7.

In our study, we analyzed the apollo OSS MSS, similar to Mo et al. (2021), who found that OSS MSS repositories’ forks primarily consist of Type-3 clones. In fact, with our results, the apollo fork displayed the highest Type-3 clone count out of all the analyzed OSS MSS repositories.

Key Takeaway: Several independent ways to analyzed code indicate that OSS MSS forks primarily contain Type-3 clones.

5.3 Industrial Implications

In this section, we discuss industrial-specific implications. Prior studies (Haroon et al. (2025)) found that LLMs struggle to understand code, as its understanding is tied to lexicographic features, leading to semantics of code being overlooked. With the advent of Model Context Protocol and the adoption of Vibe Coding as a more widespread practice in software development, it will become crucial for agentic development suites to accurately traverse and manage each layer of a codebase’s software abstractions (Steghöfer and Borg (2026)). Automatic semantic analysis tasks, such as the refactoring of code and test smells, will be paramount to avoiding long-term maintainability issues as AI-driven workflows are integrated into existing systems (Melo et al. (2026); Steghöfer and Borg (2026)). Our findings highlight the risk of automated LLM code clone detection, which directly relates to these concerns.

Key Takeaway: Further evaluation and more fine-grained approaches for code clone detection with LLMs are required, as currently the false negative rate prevents their use in automated pipelines.

6 Related Work

Concerning code clones in microservices projects, Mo et al. (2021) identified an important distinction between code clones that occur within one microservice and clones that appear across multiple services in a system. During the development of novel microservice systems and especially during the deconstruction of monolithic software into microservices, there is a risk of propagating code clones across services. Cross-service clones present a potential systemic maintenance challenge compared to clones that exist only within individual services. Services with clones that require co-modifications to reflect the same change across services end up being coupled in a way that impedes the desired *high cohesion-low coupling* nature of MSS systems

(d’Aragona et al. (2024); Liljas et al. (2026)). Using the TXL-based NiCad static analysis tool, which produces results for Type-1, 2, and 3 clones only, Mo et al. surveyed eight OSS MSS projects across five releases per project. The results showed that 28.6-60.0% of services held co-modified clones, with Type-3 clones accounting for 60-100% of clone makeup.

Table 8: Comparison of related works in code clone detection

Study	Technique	Clone Type	Contribution	This Work
Mo et al. (2021)	Deep learning	Type-4	Large-scale functional clone benchmark (multi-language)	Applied to real OSS microservice repositories
Abdelfattah et al. (2023)	CCG and ML	Type-1, Type-3	Component-level microservice analysis (Java)	End-to-end RAG + LLM pipeline
Moumoula et al. (2025)	LLM prompting	Type-1, Type-4	Prompt-based semantic clone reasoning (multi-language)	Cross-lingual evaluation with statistical testing

Abdelfattah et al. (2023) presented a microservice-specific component-level approach to clone detection using a combination of Component Call Graphing (CCG) and supervised machine learning. CCGs map internal and external call sequences of individual microservice components in order to capture dependencies and control flow across an entire system. In this process, standard call graphs are statically generated from source code and then annotated using either generalized or language-specific parsing to derive and link the component type and properties of each method depicted in the graph. Once quantified, the CCGs of individual components form a visualization of an entire microservice. External calls are then detected in the same manner and used to link the endpoints of separate microservices. Abdelfattah et al. extracted numerical representations of component semantics through similarity comparison of CCG pairs and passed the resulting similarity vectors through a logistic regression classifier to obtain binary classification of semantic clone/non-clone pairs. When tested on the train-ticket OSS MSS benchmark⁷, this method produced correct classifications for 24 out of 27 Type-4 clones. Despite the language-agnostic properties of graph-based semantic analysis, Abdelfattah et al. relied on Java-specific parsing tools for initial

⁷<https://github.com/FudanSELab/train-ticket>

CCG generation in their analysis. Thus, the applicability of this approach to polyglot use cases common in MSS systems remains limited.

More recently, [Moumoula et al. \(2025\)](#) investigated the performance of LLMs in predicting whether or not two compared code snippets are Type-4 clones. Through testing various prompts and thresholds on five LLMs, baseline models, and embedding models with the XLCoST and CodeNet datasets, the authors found that LLMs are able to detect cross-lingual clones on samples in XLCoST but struggled to accomplish the same on CodeNet. This was due to the prevalence of more complex Type-4 clone pairs in CodeNet. Moreover, the best performing model was GPT-3.5-Turbo, a multimodal, general-purpose model not trained specifically for code comprehension.

7 Threats To Validity

In this section, we will discuss threats to the validity of our study. We have categorized these threats as Construct, Internal, External, and Conclusion validity as defined by [Wohlin et al. \(2024\)](#).

Construct Validity concerns how our approach to data collection and processing may affect our results. To mitigate the threat of arbitrarily chosen data or misapplication of tools, we identified OSS MSS data sources published in prior studies ([Rahman et al. \(2019\)](#); [Yang et al. \(2024\)](#)) and utilized embedding and language models with open documentation or development literature. Additionally, the CodeNet dataset labels only Type-4 clones, providing no ground truth for Type-1, Type-2, or Type-3 detection. This limits our ability to validate LLM performance across the full clone spectrum using benchmark data.

Internal Validity addresses the accuracy of our study in minimizing unknown variables and drawing true causal relationships between data. Although we attempted to collect a language-diverse pool of samples, our overall sample size of 22 repository pairs is still relatively small, with Java-based projects accounting for half of the set. In addition, the high variability of project architectures, clone presence, and clone typing among separate repository pairs provides a potential for unknown variables to affect LLM analyses. In order to cultivate a baseline accuracy to compare our LLM analysis techniques against, we turned to a combination of the CodeNet dataset for ground truth Type-4 semantic problems ([Puri et al. \(2021\)](#)) and Jina AI’s V2 Base Code embedding model for a baseline non-generative semantic analysis of ground truth data ([Günther et al. \(2024\)](#)). Prompt structure variation between testing phases may have introduced systematic bias. We tested four prompts on Llama3-8B before selecting the best for all models, potentially favoring Llama’s architecture. However, code-specific models still outperformed Llama with this prompt, suggesting the bias did not affect our primary conclusions.

External Validity asks whether our results have the potential to generalize outside of our study. This study focuses specifically on the analysis of OSS MSS projects and therefore could fail to generalize to code clone analysis of non-microservice projects. The 22-repository dataset exhibits strong Java bias (69.2% Java-primary projects), limiting generalizability to ecosystems dominated by other languages. Our findings may not extend to JavaScript-based (Node.js) or Python-based (Django,

Flask) microservice architectures. Future work should address more programming languages, as polyglotism is considered one of the key advantages of microservices architecture.

Conclusion Validity concerns ensuring a significant statistical relationship between the treatment of data and the resulting outcomes. We adhere to an established practice of checking the data distribution, and use non-parametric statistical tests since none of the analyzed data appeared normally distributed.

8 Conclusion

This study evaluated the feasibility of using Large Language Models for semantic code clone detection in open-source microservice repositories. We assessed four prompt techniques and three LLMs on the CodeNet dataset and applied the best-performing model to analyze 22 repository-fork pairs of OSS MSS projects to assess the most common clone types.

Key Findings:

- **LLM Effectiveness.** Code-specific LLMs (Codestral-22B, Devstral-24B) significantly outperformed multimodal models (Llama3-8B) and baseline embedding approaches. However, even the best model achieved only 65.5% accuracy with 40% false negatives for Type-4 clones.
- **Clone Distribution.** Analysis of the set of 22 OSS MSS projects showed Type-1 and Type-3 clones as most common.
- **Human Validation.** Direct comparison to human validation on a subset of three projects did loosely corroborate this distribution, indicating Type-1 as most common within the subset with Type-3 indicated to a lesser degree between the Nomic and CodeSage embedding models. The Jina embedding model’s results may reflect training limitations, as they deviated significantly from human validation and were heavily skewed to Type-3 and Type-4.

Future Directions:

- Evaluate larger models (70B-175B parameters) to determine if false negative rates decrease with scale.
- Develop architectures combining LLM semantic analysis with AST-based verification to achieve production-grade precision and recall.
- The CodeNet dataset is limited to C++ code pairs and Type-4 clones, so creating a multi-language, human-annotated datasets covering all clone types would enable comprehensive model evaluation.

This work demonstrates that while LLMs offer powerful semantic capabilities for code analysis, current models lack the reliability required for critical clone management tasks. Researchers and practitioners should view LLMs as augmentation tools rather than replacements for established clone detection methods.

Supplementary information. Not applicable

Acknowledgements. This work has been funded in part by the National Science Foundation under grant number 2409933, the University of Arizona, and the University of Oulu. We used the Finland CSC Mahti supercomputer for our data analysis, testing, and results.

Declarations

The CodeNet dataset is available at <https://dax-cdn.cdn.appdomain.cloud/dax-project-codenet/1.0.0/readme.html>.

Data Availability Statement. We provide our replication package, i.e., source code and experiments on Zenodo ⁸.

Appendix A Extended Data

These are our prompts and previous data results of models not used in the paper.

References

- Abdelfattah AS, Rodriguez A, Walker A, et al (2023) Detecting semantic clones in microservices using components. SN Comput Sci 4(5). <https://doi.org/10.1007/s42979-023-01910-1>, URL <https://doi.org/10.1007/s42979-023-01910-1>
- Chang Y, Wang X, Wang J, et al (2023) A survey on evaluation of large language models. URL <https://arxiv.org/abs/2307.03109>, arXiv:2307.03109
- CodeParrot (2022) github-code. URL <https://huggingface.co/datasets/codeparrot/github-code>
- Cordeiro GM (2004) On pearson’s residuals in generalized linear models. Statistics & probability letters 66(3):213–219
- Cramér H (1999) Mathematical methods of statistics, vol 9. Princeton university press
- d’Aragona DA, Li X, Janes A (2024) Understanding the causes of microservice logical coupling: an exploratory study. In: Proceedings of the 1st International Workshop on New Trends in Software Architecture. Association for Computing Machinery, New York, NY, USA, SATrends ’24, p 14–17
- Dou S, Shan J, Jia H, et al (2023) Towards understanding the capability of large language models on code clone detection: A survey. URL <https://arxiv.org/abs/2308.01191>, arXiv:2308.01191
- Feng Z, Guo D, Tang D, et al (2020) CodeBERT: A pre-trained model for programming and natural languages. In: Cohn T, He Y, Liu Y (eds) Findings of the

⁸<https://doi.org/10.5281/zenodo.22994636>

- Association for Computational Linguistics: EMNLP 2020. Association for Computational Linguistics, Online, pp 1536–1547, <https://doi.org/10.18653/v1/2020.findings-emnlp.139>, URL <https://aclanthology.org/2020.findings-emnlp.139/>
- Grattafiori A, Dubey A, Jauhri A, et al (2024) The llama 3 herd of models. URL <https://arxiv.org/abs/2407.21783>, arXiv:2407.21783
- Günther M, Ong J, Mohr I, et al (2024) Jina embeddings 2: 8192-token general-purpose text embeddings for long documents. arXiv <https://doi.org/10.48550/arXiv.2310.19923>
- Haroon S, Khan AF, Humayun A, et al (2025) Assessing the impact of code changes on the fault localizability of large language models. arXiv <https://doi.org/10.48550/arXiv.2504.04372>
- Kaplan J, McCandlish S, Henighan T, et al (2020) Scaling laws for neural language models. URL <https://arxiv.org/abs/2001.08361>, arXiv:2001.08361
- Lewis P, Perez E, Piktus A, et al (2021) Retrieval-augmented generation for knowledge-intensive nlp tasks. URL <https://arxiv.org/abs/2005.11401>, arXiv:2005.11401
- Liljas V, Bakhtin A, Esposito M, et al (2026) Software coupling metrics. an industrial evaluation at cadmatic. In: 2026 IEEE 23rd International Conference on Software Architecture (ICSA), IEEE, accepted for the proceedings
- Liu V, Chilton LB (2022) Design guidelines for prompt engineering text-to-image generative models. In: Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems. Association for Computing Machinery, New York, NY, USA, CHI '22, <https://doi.org/10.1145/3491102.3501825>
- Melo R, Simões P, Gheyi R, et al (2026) Agentic lms: Hunting down test smells. IEEE Software 43(1):32–40. <https://doi.org/10.1109/MS.2025.3621356>
- Mo R, Zhao Y, Feng Q, et al (2021) The existence and co-modifications of code clones within or across microservices. In: Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). Association for Computing Machinery, New York, NY, USA, ESEM '21, <https://doi.org/10.1145/3475716.3475784>, URL <https://doi.org/10.1145/3475716.3475784>
- Moumoula MB, Kaboré AK, Klein J, et al (2025) The struggles of llms in cross-lingual code clone detection. Proceedings of the ACM on Software Engineering 2(FSE):1023–1045. <https://doi.org/10.1145/3715764>, URL <http://dx.doi.org/10.1145/3715764>

- Puri R, Kung DS, Janssen G, et al (2021) Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. In: Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2), <https://doi.org/10.48550/arXiv.2105.12655>, URL <https://openreview.net/forum?id=6vZVBkCDrHT>
- Rahman M, Panichella S, Taibi D (2019) A curated dataset of microservices-based systems. In: Joint Proceedings of the Inforte Summer School on Software Maintenance and Evolution, vol 2520. CEUR-WS, pp 1–9, URL https://github.com/davidetaibi/Microservices_Project_List
- Rattan D, Bhatia R, Singh M (2013) Software clone detection: A systematic review. *Information and Software Technology* 55(7):1165–1199. <https://doi.org/https://doi.org/10.1016/j.infsof.2013.01.008>, URL <https://www.sciencedirect.com/science/article/pii/S0950584913000323>
- Roy C, Cordy J (2007) A survey on software clone detection research. *School of Computing TR* 2007-541
- Shapiro SS, Wilk MB (1965) An analysis of variance test for normality (complete samples). *Biometrika* 52(3/4):591. <https://doi.org/10.2307/2333709>, URL <https://www.jstor.org/stable/2333709?origin=crossref>
- Steghöfer JP, Borg M (2026) An abstraction is worth a thousand vibes. *IEEE Software* 43(1):13–16. <https://doi.org/10.1109/MS.2025.3621786>
- Suresh T, Reddy RG, Xu Y, et al (2025) Cornstack: High-quality contrastive data for better code retrieval and reranking. *arXiv* <https://doi.org/10.48550/arXiv.2412.01007>
- Svajlenko J, Roy CK (2014) Evaluating modern clone detection tools. In: Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution. IEEE Computer Society, USA, ICSME '14, p 321–330, <https://doi.org/10.1109/ICSME.2014.54>, URL <https://doi.org/10.1109/ICSME.2014.54>
- Wilcoxon F (1945) Individual comparisons by ranking methods. *Biometrics Bulletin* 1(6):80–83. URL <http://www.jstor.org/stable/3001968>
- Wohlin C, Runeson P, Höst M, et al (2024) *Experimentation in Software Engineering*, Second Edition. Springer, <https://doi.org/10.1007/978-3-662-69306-3>
- Yang W, Song B, Xing Y, et al (2024) A feature dataset of microservices-based systems. In: *International Conference on Service Science (ICSS)*, Springer, pp 73–87, https://doi.org/10.1007/978-981-97-5760-2_6, URL <https://github.com/yang66-hash/microservice-catalog>

Zhang D, Ahmad W, Tan M, et al (2024) Code representation learning at scale.
In: The Twelfth International Conference on Learning Representations (ICLR'24),
<https://doi.org/10.48550/arXiv.2402.01935>, URL <https://openreview.net/forum?id=vfzRRjumpX>

Table A1: Clone detection prompts

Prompts			
Type	Name	Content	JSON Values
Zero-shot	Prompt 1	<pre> """ Definitions: _Type-1:_Identical_except_for_whitespace,_comments,_layout. _Type-2:_Identical_except_for_variable/function_names_(plus_Type-1_differences). _Type-3:_Similar,_but_with_some_statements_added/removed/modified_(plus_Type-2_differences). _Type-4:_Syntactically_different_but_functionally_identical_(same_outputs_for_same_inputs). Analyze_the_following_two_code_snippets_and_determine_whether_they_are_clones,_regardless_of_the_programming_language._Respond_with_'yes'_if_the_code_snippets_are_clones_or_'no'_if_not_for_each_category_of_Type-1,_Type-2,_Type-3,_and_Type-4. """ </pre>	yes/no
	Prompt 2	<pre> """ {Prompt_1_Definitions} Analyze_the_following_two_code_snippets_for_code_clone_detection,_regardless_of_the_programming_language._You_should_first_report_which_lines_of_code_are_more_similar._Then_based_on_the_report,_please_answer_whether_these_two_codes_are_a_clone_pair._The_response_should_be_'yes'_or_'no'_for_each_Type. """ </pre>	yes/no
Few-shot	Prompt 3	<pre> """ You_are_a_code_similarity_expert._Analyze_the_following_code_pair_step_by_step_for_clone_detection. {Prompt_1_Definitions} Examples: Type-1:_int_a=5;_vs_ int_a=5;_//_set_a' Type-2:_int_a=5;_vs_ int_b=5;' Type-3:_int_a=5;_print(a);_vs_ int_a=5;' Type-4:_for(int_i=0;i<5;i++)sum+=i;_vs_ sum=_sum_of_first_n(5);' Step-by-step: 1._Are_the_outputs_always_identical_for_all_inputs?(Type-4) 2._Are_they_identical_except_for_whitespace/comments?(Type-1) 3._Are_names_changed_but_structure_identical?(Type-2) 4._Are_there_statement-level_edits?(Type-3) Explain_your_reasoning_for_each_type_and_provide_a_'yes'_or_'no'_for_each_category. """ </pre>	yes/no
	Working	<pre> """ You_are_a_code_similarity_expert._Analyze_the_following_code_pair_step_by_step_for_clone_detection. {Prompt_1_Definitions} {Prompt_3_Examples_and_Steps} Explain_your_reasoning_for_each_type_and_provide_a_confidence_score_(0-1). Respond_ONLY_with_a_JSON_object_with_the_following_keys:_Type-1,_Type-2,_Type-3,_Type-4,_each_with_a_confidence_score_between_0_and_1. """ </pre>	score 0-1

Table A2: Model Results Not Used.

Model Name	Clone Type	Precision	Recall	F1 Score	MSE	Accuracy
starcoder2-15b-instruct-v0.1	Overall	0.42	0.34	0.38	0.52	0.48
	Non-clone	0.51	0.60	0.55		
	Clone	0.42	0.34	0.38		
DeepSeek-R1-Qwen3-8B	Overall	0.56	0.55	0.54		0.57
	Non-clone	0.57	0.75	0.65		
	Clone	0.55	0.35	0.43		