ZC3 - Cross Code Clone Detection: A Replication Study

Tanveer Reza
Ryane Lekbir Hilmi
Yehia Metwally
Jerome Kithinji

Friday April 18th, 2025

Abstract

Detection of code clones is a critical task in software maintenance and evolution, where developers implement similar functionality across programming languages to support multiple platforms. Traditional clone detection methods struggle with Type-4 code clones. The paper aims to explore these challenges by validating and training the ZC3 [10] model for cross-language code classification and retrieval tasks using a supervised fine-tuning setup. Comparing it with C4 [19], a pre-trained transformer model enhanced with contrastive learning to improve the representation of functionally similar code snippets while distinguishing them from non-clones. The results of the replication show that ZC3 [10] performs well on seen data using the XLCost [23] dataset, where it scored 98.22% Query to Candidate and 97.79% Candidate to Query aligning with the original paper. As well as on Unseen dataset using GPT-CloneBench [2], scoring 75.9% for Query to Candidate and 78.6% from Candidate to Query on the MAP(Mean average precision) score basis. We then extend our research to train and test the ZC3 [10] model on Atcoder [4] dataset against the C4 [19] model. Where we tested it for classification, the results shows that C4 [19] outperformed the ZC3 [10] with ZC3 [10] scoring 84% against 91% for C4 [19] on the F1-Score basis. We trained and tested the ZC3 [10] for cross-language code clones and Semantic code clones on GPT-CloneBench [2] where the results where surprisingly low scoring 51% for cross language clones and 23% for semantic clones.

Keywords: Code Clone Detection, ZC3, C4, supervised fine-tuning, Unseen dataset, MAP, F1-score, Classification, GPT-CloneBench, XLCost, Semantic Code Clones, Candidate to Query, Query to Candidate.

1 Introduction

Code clones are duplicated segments of code found within a software project. These clones can appear either intentionally when developers reuse code across different parts of a project to avoid rewriting it, or accidentally when similar code is copied and pasted without proper code structure. Although code clones can improve efficiency, they also introduce risks like higher maintenance costs, increased technical debt, and bugs, especially when updates in one part of the code aren't reflected in others [8]. Code clones are generally classified into four types:

- **Type-1** (exact clones): are identical copies with only minor differences such as whitespace or comments;
- **Type-2** (renamed clones): are structurally identical but include renamed variables or functions;
- **Type-3** (near-miss clones): contain added, removed, or modified statements while maintaining structural similarity;
- **Type-4** (semantic clones): perform the same function but use different logic, algorithms, or structures [20].

In this paper, we focus on **Type-4** code clones, which are semantically similar but syntactically different code fragments, and pose significant challenges in software maintenance and evolution [21]. Detecting these clones is crucial because they can lead to inconsistent bug fixes, increased maintenance costs, and hindered code comprehension if left unmanaged [20]. Recent studies have highlighted the importance and complexity of identifying Type-4 clones [21]. Wei and Li (2017) proposed a supervised deep learning approach that leverages lexical and syntactical information to detect functional clones, highlighting that traditional clone detection methods struggle with capturing deep semantic similarities in code (IJCAI) [20]. Xue et al. (2022) introduced SEED, a semantic graph-based detection model specifically designed for Type-4 clones, demonstrating that graph-based deep learning techniques are necessary to model the underlying logic and structure of code rather than just surface-level similarities (arXiv) [21]. These studies reinforce the need to focus on Type-4 clone detection, as it represents the most challenging and impactful clone type that requires advanced AI-driven approaches to improve software quality, maintainability, and security [21].

Detecting code clones is a critical task in software maintenance, with studies showing that between 7% and 23% of codebases contain clones, depending on the software system and industry. Clone detection can generally be divided into two categories: monolingual clone detection, which focuses on identifying clones within the same programming language, and cross-language clone detection, which involves detecting functionally equivalent code across different programming languages [19]. However, the majority of

the clone detection literature is confined to monolingual clone detection.

Cross-language code clone detection is used in modern software development, where applications are often implemented across multiple programming languages to support various platforms [14]. Unlike monolingual detection, cross-language clone detection requires identifying semantically equivalent code despite differences in syntax, structure, and naming conventions. Identifying functionally similar code across languages enhances code reuse, facilitates maintenance, and improves software quality [14]. Recent studies have demonstrated the challenges and advancements in this area [14]. Hu et al. proposed a method using graph neural networks to improve code representation learning, highlighting the difficulty of aligning semantically equivalent code across different languages. Nafi et al. introduced CLCDSA, which detects cross-language clones using syntactic features, proving that traditional clone detection methods struggle with deep semantic differences between languages [14]. Furthermore, the application of large language models has recently shown promising results in handling cross-language similarity detection [4]. These studies emphasize the growing need for more robust, scalable, and automated methods in cross-language clone detection, but there is still a lack of effective, scalable solutions, justifying further research in this area [14].

This paper does a replication study of the paper "ZC3: Zero Shot Cross Language Code Clone Detection" [10]. ZC3 is a novel method for detecting cross-language code clones without the need for parallel data or direct language mappings. By leveraging domain-aware learning and cycle consistency learning, ZC3 can identify semantically equivalent code (Type 4 clones) across different programming languages. The results show that ZC3 outperforms existing unsupervised models for cross-language clone detection, with particularly strong performance in Java. This highlights ZC3's effectiveness in detecting cross-language clones without requiring costly parallel datasets. We make the following contributions:

We validate the results of ZC3 on XLCost, test it on unseen dataset on GPTCloneBench for retrieval tasks. We compare ZC3 [10] and C4 [19] on our selected datasets to evaluate the performance of zero-shot learning (ZC3) [10] against the supervised fine-tuning approach in C4 for classification tasks. We train and evaluate the ZC3 on GPTCloneBench dataset to test its performance for cross language clones and semantic clones classification on new dataset.

Keywords: Cross-language code clone, Zero-Shot, Supervised fine-tuning, LLM, GraphCodeBERT, UniXCoder, ZC3.

2 Background

2.1 Zero-Shot

Zero-shot learning (ZSL) is a machine learning approach where a model is trained on a set of seen classes and is then expected to recognize unseen classes without being explicitly trained on them. Instead of relying on labeled examples, ZSL leverages auxiliary information such as semantic attributes, natural language descriptions, or embeddings from related tasks to infer similarities between known and unknown classes [22].

2.2 Supervised fine-tuning

Supervised Contrastive Learning (SCL) is a training framework that enhances representation learning by encouraging models to group similar instances closer together in a high-dimensional space while pushing dissimilar ones apart. Unlike traditional supervised learning that relies on classification loss (cross-entropy), SCL uses contrastive loss to learn better feature embeddings [9].

2.3 C4 Contrastive Cross-Language Code Clone Detection model

The **C4** (Contrastive Cross-Language Code Clone Detection) [19] paper introduces a supervised deep-learning approach to detect cross-language code clones, which are functionally similar code fragments written in different programming languages. The motivation behind this research is the increasing prevalence of cross-language software development, where developers implement similar functionality across different programming languages to support multiple platforms. Traditional clone detection methods struggle with Type-IV clones (semantically similar but syntactically different), making automated detection crucial for software maintenance and evolution. The paper aims to address this challenge by leveraging **CodeBERT** [5], a pre-trained transformer model, and enhancing it with contrastive learning to improve the representation of functionally similar code snippets while distinguishing them from non-clones. The study evaluates **C4** against state-of-the-art baselines using the **CLCDSA** [14] dataset, demonstrating substantial improvements in *Precision (0.94)*, *Recall* (*0.90*), and *F1-score (0.92)* compared to previous methods. The results highlight that contrastive learning significantly enhances the detection of cross-language clones.

2.4 Summary of ZC3:

The paper proposes a novel approach to detect cross-language code clones without parallel data. It creates a contrastive snippet prediction by utilizing domain-aware learning and cycle consistency learning. Basi-

cally, the method wants to retrieve semantically equivalent code clones (Type 4 clones) from a large codebase after being given a specific program as a query. It uses CodeBert as the base for design because it is pre-trained on large amounts of unlabelled multilingual data and can map the representation of different languages to a unified space [5]. The research scope, methods, and findings from this paper are discussed below.

2.4.1 Research Questions

- RQ1: How does ZC3 perform compared to the SOTA (state-of-the-art) unsupervised cross-language code clone baselines? Compare ZC3 with SOTA baselines across four datasets.
- RQ2: What is the performance of ZC3 in the unseen programming languages?
- RQ3: How does ZC3 perform on the monolingual code clone detection?

2.4.2 Datasets

CSNCC : Dataset for adversarial code clone. Contains function-level programs in Ruby, python, and Java. Taken 500 problems from a GitHub repository. **CodeJam**: Google's programming competition dataset with 256 problems and solutions in Python and Java **AtCoder**: Atcoder programming in Japan. Programs in Python and Java. **XLCOST:** Used to construct an isomorphic embedded space. Contains 7 different programming languages with several consecutive snippets containing if-else, for statements. Data structure and algorithm functions. **BigCloneBench:** For monolingual clone detection **POJ-104:** For monolingual clone detection, judges the correctness of submitted code. 500 student-written C programs

2.4.3 LLMs for comparison

- **CodeBert** : is a pre-trained transformer model for code understanding, designed to generate high-quality code embeddings.
- **GraphCodeBert** : is a pre-trained model for code understanding that incorporates both code syntax and data flow
- **UnixCoder** : is a pre-trained model that understands and generates code by combining text and code structure
- **CodeBertMCC** : CodeBert finetunes on monolingual code clone pairs in java and python.
- **CodeBertCSP** : CodeBert post trained with contrastive snippet prediction and fine-tuned like CodeBertMCC.
- **CodeBertDAL** : The model can be regarded as an ablation model where the cycle consistency learning is removed from our ZC3.
- **CodeBERTFT** : for monolingual code clone detection, trained on **BigCloneBench** and **POJ-104**

Metrics: MAP (Mean Average Precision)

2.4.4 Results of ZC3

- **RQ1:** ZC3 achieves the best results among all baselines.
- **RQ2:** ZC3 still performs better, but results for JAVA are better than Python.
- **RQ3:** ZC3 performs well on the monolingual clone detection scenario, and its performance is close to the counterpart of the supervised method, but its performance is slightly lower than **CodeBERT**

3 Related Work

Recent research in code clone detection has increasingly focused on leveraging LLMs to address the challenges of detecting Type-4 code clones. These studies explore diverse approaches, ranging from instruction tuning and fine-tuning with functionally equivalent datasets to graph-based representations, each offering insights into model generalizability, semantic understanding, and practical integration.

Almatrafi et al. [3] investigated the use of instruction-tuned large language models, specifically GPT-3.5 and GPT-4, for detecting semantic code clones, particularly Type-4 clones that are functionally similar but structurally different. The authors applied few-shot prompt engineering to adapt these general-purpose LLMs to the clone detection task without requiring extensive retraining. One of the key contributions of their work is the integration of the model into the IntelliJ IDEA environment, enabling real-time detection and automated refactoring of clones within an IDE. Their experimental results showed that GPT-4 achieved an *F1-score* of *0.93* on Type-4 clones, outperforming GPT-3.5 and traditional detection techniques such as token-based or AST-based methods. This highlights the strong semantic reasoning abilities of instruction-tuned LLMs. However, the study also identifies limitations, including the models' sensitivity to prompt design, the black-box nature of GPT-4, cost and API access constraints, and a limited evaluation scope that focuses primarily on semantic clone detection. Overall, their work emphasizes the growing potential of LLMs in practical code clone detection and supports the use of instruction-tuned models in real-world development tools.

Inoue and Higo [7] investigate how fine-tuning large language models (LLMs) on a dataset of functionally equivalent but structurally different Java methods can enhance Type-4 code clone detection. They introduce the FEMP dataset, specifically curated for semantic clone detection, and apply efficient fine-tuning techniques such as LoRA and ZeRO to models like CodeLlama and LLaMA2. Their experiments demonstrate that fine-tuning significantly improves performance, with GPT-3.5 achieving an *F1-score* of *0.87* and CodeLlama exhibiting a well-balanced enhancement in both precision and recall. The study also highlights that off-the-shelf LLMs tend to misclassify non-clones due to a lack of semantic understanding, an issue mitigated through targeted training. However, the authors note limitations including the dataset's restriction to Java and its exclusive focus on Type-4 clones, which may impact the generalizability of their findings. Despite these constraints, the work underscores the importance of supervised learning and task-specific datasets.

Mehrotra et al. [11] propose RUBHUS, a semi-supervised deep learning framework designed to improve cross-language code clone detection by incorporating enriched code representations and graph neural networks (GNNs). RUBHUS constructs flow-enriched Abstract Syntax Trees (ASTs) by adding semantic edges such as last-read, last-write, and compute-from relationships, capturing deeper program logic across languages. These enriched ASTs are encoded using a GNN, and the resulting embeddings are optimized via mutual information maximization to retain semantic structure. A binary classifier then determines clone similarity between code pairs. The model is evaluated on datasets from AtCoder and CodeChef, demonstrating superior performance over several state-of-the-art baselines including **C4**, **GraphCodeBERT**, **CLCDSA**, and **PCCD**. Specifically, **RUBHUS** achieves *F1-scores* up to *97%* on Type-4 (semantic) clones and generalizes effectively between statically and dynamically typed languages. However, the framework relies on language-specific AST parsers, introduces additional preprocessing and training complexity, and is evaluated primarily on competitive programming code, which may limit its applicability to broader software domains. Despite these limitations, the work offers a compelling alternative to traditional and LLM-based clone detectors by emphasizing the importance of graph-based semantic representations in cross-language clone detection.

These studies helped us shape our approach by showing the benefits of instruction tuning and fine-tuning with semantic data, as well as the challenges general-purpose models face with unseen data. Based on this, we wanted to test ZC3's performance on different datasets and compare its performance with C4, so we can observe the difference between the two models with different approaches when tested on the same dataset and the same setup.

4 Paper Design

4.1 Research questions

- RQ1: How valid are the results of ZC3 with the XLCOST dataset and how it performs on unseen datasets?
- RQ2: How do **ZC3** and**C4** perform for cross code clone detection?
- RQ3: How **ZC3** performs on new dataset for classification?

4.2 Research questions approach

- RQ1: We want to validate the findings of **ZC3** with the **XLCost** dataset [23] and measure its generalizability with the **GPTCloneBench** dataset [2].
- RQ2: We want to measure the robustness of **ZC3** for classification tasks. **ZC3** and **C4** offer two distinct approaches to detecting functionally similar code across different programming languages.**ZC3** is a zero-shot learning model that does not require labeled clone pairs, making it highly generalizable to unseen languages without retraining. In contrast, **C4** is a supervised model that uses contrastive learning and fine-tuning on labeled clone pairs to enhance clone detection accuracy. ZC3 is designed to detect clones across languages it has never seen before, making it ideal for scalable and adaptive programming environments where labeled data is scarce. However, its performance may be less precise compared to a fine-tuned model. **C4** benefits from contrastive learning, allowing it to more effectively differentiate between similar and dissimilar code, but it requires extensive labeled training data, limiting its adaptability to new languages. The premise of this research question arises from the **ZC3** paper itself, which states that [19] *"**C4** uses a large amount of parallel multilingual data for contrastive learning, which is a competitive method for supervised cross-language clone detection"*. Given this, we aim to directly compare **ZC3** and **C4** to evaluate their performance against each other, particularly in terms of their ability to detect cross-language code clones and semantic clones. To keep the results fair, we will use the **AtCoder** dataset [15], which was previously applied by both papers to present their results and achieved a high score.

- RQ3: To extend the research, we will train and evaluate the **ZC3** model on the **GPT-Clonebench** dataset [2] and test its performance for classification.

  We selected **GPTCloneBench** [2] over **BigCloneBench**, which was previously used in the original study for generalization, based on insights from *On the Use of Deep Learning Models for Semantic Clone Detection* [15], a study that outlines several key limitations of **BigCloneBench**. The paper highlights that **BigCloneBench** is primarily based on structural similarity and lacks purely semantic clones, which means that it does not include code pairs that are functionally similar but structurally different. Furthermore, the definition of semantic clones is ambiguous. The dataset suffers from class imbalance due to its design principles, making it less suitable for training and evaluating machine learning-based clone detection models.

  Instead, the paper [15] recommends **SemanticCloneBench** and **GPTCloneBench** as they contain more diverse, realistic, and challenging semantic clones, making them a better reflection of real-world clone detection scenarios. By using this dataset, we aim to determine how **ZC3** performs for classification tasks on new datasets.

4.3 Evaluation Metrics:

We used MAP score as evaluation metrics to validate and compare the performance of **ZC3** with **XLCost** dataset [23] **(RQ1)**. We used *Precision*, *Recall*, and *F1-scores* to assess how **ZC3** and **C4** perform on **AtCoder** [4] and **GPT-CloneBench** [2] datasets to measure its robustness and generalizability **(RQ2, RQ3)** [5] [6] [16].

Mean Average Precision (MAP) is an effective metric for evaluating ranking models, as it considers both the relevance of results and their position in the ranked list, assigning higher scores to models that rank relevant items earlier.

F1, **precision**, and **recall** metrics are widely recognized for providing a comprehensive evaluation of classification models, particularly in balancing the trade-off between correctly identifying true clones and minimizing false positives.

Precision measures the proportion of correctly identified clones among all positive predictions, ensuring that the model does not mistakenly classify non-clones as clones.

Recall evaluates the model's ability to detect actual clones, preventing the omission of relevant results.

F1-score, the harmonic mean of Precision and Recall, provides a balanced assessment when there is a trade-off between the two. The **C4** paper [19] itself validates the use of these metrics, demonstrating their effectiveness in assessing clone detection performance across programming languages. This approach guarantees a structured and comparative evaluation of the models, reinforcing the validity of our research.

4.4 Datasets:

The **XLCost** [23] dataset contains a diverse set of code snippets across seven programming languages, including **Python**, **Java**, **C++**, **C**, **JavaScript**,**Go**, and **Ruby**. It focuses on capturing structurally and semantically similar code that solves the same logic patterns, such as loops and conditional statements. Each example includes aligned code snippets implementing equivalent functionality, making it ideal for evaluating cross-language code representation and clone detection. We selected **XLCoST** as it serves as the primary training dataset for the **ZC3** model, making it a strong and consistent foundation for our replication study and performance comparison.

AtCoder [4] dataset comprises of a large collection of competitive programming code submissions from the **AtCoder** [4] platform, primarily written in **Python** and **Java**, which are commonly used in cross-language clone detection studies. The dataset includes solutions to various algorithmic problems and captures functionally similar code implementations across languages, making it well-suited for evaluating code similarity and clone detection tasks. Each submission is labeled with metadata such as problem ID, language, and execution status. We selected **AtCoder** [4] because it is one of the training datasets used in the **C4** [19] model and demonstrated strong results in cross-language clone detection, making it a good benchmark for performance comparison.

GPT-CloneBench[2] is a dataset introduced alongside advancements in generative models. It builds on SemanticCloneBench[1] by using a GPT-based model to generate additional semantic clone pairs.**GPT-CloneBench**[2] is an essential resource for evaluating code clone detection models, particularly for identifying semantic and cross-language clones. It offers a large-scale, diverse benchmark with over *37,000* true semantic clone pairs and nearly *21,000* cross-language clone pairs across **Java**, **C**, **C#**, and **Python**, making it ideal for assessing both intra and inter-language clone detection capabilities. Unlike traditional clone datasets, **GPT-CloneBench** [2] leverages generative AI techniques to enhance coverage, capturing subtle functional similarities that conventional syntactic-based benchmarks may miss. By incorporating it into our research, we can provide a more comprehensive evaluation of **ZC3**'s [10] effectiveness in detecting complex code clones, ensuring its robustness in real-world software engineering applications [17][18][15].

4.5 Implementation Details

4.5.1 Preprocessing Data

To prepare the datasets for training and evaluating the ZC3 [10] model, we developed a set of preprocessing scripts to convert raw code files into a clean and standardized JSON Lines (JSONL) format. We specifically designed this formatting process to fit the input requirements of the ZC3 [10] model, which expects each line to represent a single code example in JSON form, containing fields such as `index` (a unique identifier), `label` (to group related clones), `domain_label` (0 for source, 1 for clone or translation), `description` (programming language), and `code` (the actual snippet).

For the training data, each label was required to have at least four code examples: two with `domain_label: 0` and two with `domain_label: 1`. We wrote our own custom programs, which is included in our replication package, to achieve this.

For evaluation, we formatted the query and candidate datasets using the same JSONL structure, where query entries were assigned `domain_label: 0`, and candidate entries were assigned `domain_label: 1`. The `query` file contains code snippets written in one programming language, serving as input queries for the retrieval or classification task. The `candidate` file, on the other hand, contains code snippets written in a different programming language or consists of semantic clones, forming the pool of potential matches for each query data.

All training, query, and candidate data files, custom programs and modified model files used in our experiments have been made publicly available to support transparency and replication in future research.

4.5.2 LLM setup

We trained the ZC3 [10] model for cross-language code classification and retrieval tasks using a contrastive learning setup. The model was trained over 3 epochs, with a training batch size of 8 and an evaluation batch size of 16. Input sequences were limited to a maximum length of *512* tokens, and training was performed using a learning rate of *2e-5*, with gradient clipping set to 1.0 to maintain stability. The model was initialized with pretrained weights from *CodeBERT (Microsoft/codebert-base)*, and tokenization was handled using the *RoBERTa* tokenizer (Roberta-base) to ensure consistency with the underlying model architecture.

Training data consisted of examples from the **XLCost, AtCoder, and GPTCloneBench** [23][2] [4] datasets, which are included in our code package. Evaluation was conducted on both retrieval and classification tasks in a cross-language setting, using the specified QUERY files as the query set and CANDIDATE files as the candidate set.

To monitor training progress, we enabled evaluation during training and saved model checkpoints every 50 steps. For reproducibility, the random seed was fixed at 123456.

4.5.3 Fine tuning for F1 score

To compute the *F1-score*, we first generated vector representations for both query and candidate samples using the trained model. These vectors were then used to compute cosine similarity scores between all query-candidate pairs. Based on these similarity scores, we identified the top-most similar candidate for each query and compared its label to the true query label to determine whether the prediction was correct.

Using these predicted and true labels across the dataset, we calculated:

Precision: the proportion of correctly predicted positive instances among all predicted positives,

Recall: the proportion of correctly predicted positives among all actual positive instances.

The *F1-score* was then computed as the harmonic mean of precision and recall, offering a balanced metric that accounts for both false positives and false negatives. This approach transforms the ranking-based evaluation into a classification-oriented metric, better reflecting the system's performance in top-1 matching tasks.

5 Results

5.1 RQ1 Results

5.1.1 Replication and Validation of ZC3

Query → Candidate	**Candidate → Query**
Original results from paper with XLCost	
96.96%	*96.92%*
Replication Results with XLCost	
98.22%	*97.79%*

Table 1: Replication Results for XLCost

The model was evaluated on the **XLCOST** [23] dataset, comprising *3,997* data points for evaluation and *33,000* for training. Training was conducted over 3 epochs. For the query language used was **Python**, the candidate languages considered were **Java**, **C++**, **C**, and **JavaScript**. As shown in *Table 1* 1 We were able to validate the results of the paper. Our results are slightly higher and that could be due to overfitting and data leakage as we trained and evaluated it again on the same dataset used.

5.1.2 Generalization on an Unseen Dataset

Query → Candidate	Candidate → Query
75.9%	*78.6%*

Table 2: Results for GPTCloneBench

To assess the model's generalization capability, we evaluated it on the unseen **GPTCloneBench** [2] dataset, which contains *7,046* data points. The dataset spans multiple programming languages: **Java**, **Python**, **C**, and **C#**. The model was trained on the **XLCost** [23] dataset and evaluated on **GPTCloneBench** for 3 epochs, consistent with prior experiments. Despite not being exposed to this dataset during training, the model achieved a good score, as shown in *Table 2* 2. These results suggest that while performance on unseen data is comparatively lower than on seen datasets, the model still has a reasonable ability to capture cross-language semantic similarity and perform effective code mapping across unfamiliar contexts. **We conducted the *evaluation 3* times for both datasets and presented the average MAP score in the tables.**

5.2 RQ2 Results

Approach	Recall	Precision	F1 Score
ZC^3	0.82	0.86	0.84
C4	0.90	0.93	0.91

Table 3: Performance comparison of ZC^3 and C4 with AtCoder dataset

To evaluate the relative performance of different model architectures, we conducted a comparative analysis of the **C4** [19] and **ZC3** [10] models using the **AtCoder** [4] dataset. The evaluation set comprised 34,789 instances, with *77,000* data points used for training. The experiments were conducted with two programming languages: **Python** and **Java**, with both models trained for 3 epochs under identical conditions. As shown in *Table 3* 3 the **C4** [19] model outperformed the **ZC3** [10] model. This indicates that the **C4** [19]architecture demonstrates superior performance in capturing semantic alignment and language translation within the **AtCoder** [4] dataset. While the **ZC3** [10] is performing incredibly good for classification on *F1-score* basis not far behind **C4** [19] model.

5.3 RQ3 Results

Approach	Recall	Precision	F1 Score
Cross Code	0.61	0.48	0.51
Semantic	0.74	0.17	0.23

Table 4: Performance comparison of ZC^3 and C4 with AtCoder dataset

5.3.1 ZC3 performance on new dataset for cross-language code clone classification

The **ZC3** [10] model was trained on the **GPTCloneBench** [2] dataset to assess its effectiveness in detecting cross-language code clones. The training set consisted of *62,000* examples, while evaluation was conducted on a separate set of *7,046* samples. The dataset includes code in **Java**, **Python**, **C**, and **C#**, and the model was trained for 3 epochs. The result shown in *Table 4* 4 suggests that there remains considerable room for improvement in generalizing code clone detection in multilingual settings. The high *recall* indicates that the model is capable of identifying the majority of clones across different programming languages, showing its strength in generalizing across language boundaries. However, the lower precision suggests that many of the predicted clones are incorrect, highlighting a tendency to over-predict matches where there may be none. The moderate *F1-score* reflects a trade-off between these strengths and weaknesses. Overall, while the model shows promise in recognizing semantic similarities across languages, it still needs improvement in filtering out false positives to enhance the reliability of its predictions.

5.3.2 ZC3 performance on new dataset for semantic clone classification

To evaluate the **ZC3** [10] model's capability in identifying semantic code clones, we trained it on the **GPTCloneBench** [2] dataset, which includes *62,000* training samples and *7,044* evaluation instances spanning **Java**, **Python**, **C**, and **C#**. The model was trained for 3 epochs. As shown in *Table 4* 2 the high recall indicates that the model is able to identify a large portion of actual semantic clones across languages, demonstrating sensitivity to potential matches. However, the low precision reveals that many of these predicted matches are incorrect, resulting in a high number of false positives. The low *F1-score* reflects this imbalance between correctly retrieved and accurately predicted clones. These results suggest that while **ZC3** [10] is effective at finding clone candidates, it struggles to distinguish true semantic matches from unrelated code, limiting its generalization for classification problems.

6 Discussion

6.1 Challenges faced

In conducting our study, we encountered a few challenges. Firstly, the original paper did not specify any data preprocessing steps, which meant we had to implement code solutions to format the datasets according to the input structure expected by the **ZC3** [10] model. Manually adapting these datasets to the **ZC3** [10] model's required **JSONL** format was time-consuming and required careful validation. Additionally, the original implementation lacked *F1-score* computation. For that reason, we had to extend the model to include this ourselves.

On the hardware side, each training and evaluation run took around 10–12 hours, likely because we were only using one GPU. Moreover, the original model was not directly compatible with our GPU setup. We had to make environment-level modifications, install specific dependencies, and tweak parts of the code to ensure successful execution. These challenges highlight the importance of clear documentation and adaptable implementations when creating replication packages for large language models.

6.2 Takeaways and Lessons Learned

6.2.1 Takeaways

This replication study revealed four key takeaways that may inform future research in this area.

First, our evaluation confirms that the ZC3 [10] model excels in cross-language code retrieval tasks, particularly when evaluated on the XLCOST [23] dataset. This solidifies the model's effectiveness in retrieval tasks, which aligns with its primary design purpose.

Second, despite being primarily designed for retrieval, **ZC3** [10] demonstrates notable versatility by performing well in classification tasks as well. It achieved an *F1-score* of *84%*, which, while slightly lower than the *91%* achieved by the dedicated classification model **C4** [19], highlights **ZC3**'s [10] capacity to adapt to a broader range of tasks.

Third, **ZC3** [10] exhibits promising generalization capabilities, particularly evident when it performed well on the unseen **GPTCloneBench** [2] dataset. This suggests that the model can effectively detect cross-language code clones across diverse environments, reinforcing its adaptability.

Fourth, while **ZC3** [10] shows strong performance in many cases, its effectiveness declines in more complex and new scenarios. Specifically, its performance suffers when evaluated on Type-4 code clones and completely new cross-language datasets. These results indicate that the model's robustness and ability to generalize to more intricate and varied clone detection tasks still require further refinement.

6.2.2 Lessons Learned

It is essential to thoroughly examine the replication package provided by the original authors, ensuring that all datasets, model files, and preprocessing steps are correctly included and documented. Attempting replication without verifying these elements can lead to misinterpretations or failures unrelated to the model itself.

Additionally, reaching out to the original authors was a valuable strategy for us. Early communication helped clarify ambiguities and saved significant time that would otherwise be spent troubleshooting undocumented assumptions or environment-specific configurations.

Furthermore, we strongly recommend that researchers test their own replication packages by attempting to re-run their experiments independently. This practice helps identify potential gaps in documentation or setup scripts before the package is shared publicly, ultimately increasing transparency and reproducibility for the broader research community.

Finally, it is important to ensure that the model and codebase are compatible across different computing environments. If full compatibility cannot be achieved, it is good practice to explicitly note such limitations in the replication package to help future users set up their environment correctly and avoid frustration.

In addition to the technical insights gained through replication, this study provided us with valuable hands-on experience in working with LLM models. We learned how to preprocess raw datasets into standardized formats, fine-tune large language models, and evaluate their performance on both seen and unseen tasks. As this was our first exposure to this area of research, the project served as a practical introduction to key concepts in machine learning and software engineering, deepening our understanding of real-world challenges in reproducibility, model generalization, and evaluation.

7 Threats to validity

7.1 Internal Validity

Codebase Modifications: In order to adapt the **ZC3** [10] model to run efficiently on our GPU environment, we had to modify the original codebase. These changes, while necessary for hardware compatibility, could potentially have introduced unintended side effects that might have affected the model's performance. Any alteration to the underlying code, even minor, can influence how the model processes

data, executes training, and ultimately achieves its results. Therefore, the modifications may not be perfectly aligned with the original experimental setup, which could impact the reproducibility and comparison of results.

Number of Epochs: The number of training epochs used in our experiments was fewer than what was specified in the original paper. This decision, while practical for our resource constraints, may have led to premature stopping of the model training before it reached its full potential. As a result, the model may not have fully converged, which could have affected the overall performance. Training for a longer period could potentially yield better results, especially for the **GPT-CloneBench** [2] dataset, and the choice of epochs might limit the reliability of our findings.

Preprocessing Bias: We applied specific formatting and preprocessing steps to the **GPT-CloneBench** [2] dataset in order to make it compatible with our model. However, these preprocessing choices might have introduced biases that influenced the model's performance. In particular, changes in the way code snippets were tokenized, normalized, or structured could lead to differences in how the model interprets the data, potentially affecting its ability to generalize across different tasks. The preprocessing steps may have inadvertently favored certain types of inputs or structures, which could affect the model's performance in ways not originally intended by the authors of the dataset.

7.2 Construct Validity

Change in Evaluation Metric: We altered the evaluation metric from Mean Average Precision (MAP) to *F1-score*. This shift may have influenced the results, as *F1-score* emphasizes a balance between precision and recall, potentially highlighting different aspects of the model's performance compared to MAP, which focuses more on ranking accuracy.

7.3 External validity

Limited Dataset Availability: The availability of open-source datasets is a significant constraint. With limited public datasets, the generalization of our findings to other real-world scenarios is challenged. Our evaluation may not fully capture the diversity of cross-language code retrieval and classification tasks.

8 Conclusion

This study examined the effectiveness of the ZC3 [10] model in detecting cross-language and semantic code clones through supervised fine-tuning and empirical evaluation. By replicating and extending the original ZC3 [10] setup, we validated its performance on the XLCoST [23] dataset and further assessed its generalization to unseen datasets such as **GPT-CloneBench** [2]. While ZC3 [10] demonstrated strong performance on seen and unseen data for retrieval, its accuracy declined on more complex semantic and cross-language tasks, particularly in classification settings where the C4 [19] model outperformed it. These findings underscore the trade-offs between zero-shot and fine-tuned approaches, highlighting that while zero-shot models offer greater scalability and adaptability, fine-tuned models like C4 [19] currently deliver higher precision in identifying functionally similar code.

9 Replication Package

We added all the scripts, preprocessing programs, datasets and models used in a Github repository [12] and a Zenodo repository [13] for any future replications.

10 Credits

- Conceptualization
  - Tanveer Reza, Jerome Kithinji, Yehia Metwally, Hilmi Ryane Lekbir
- Data curation and analysis
  - Tanveer Reza, Jerome Kithinji, Yehia Metwally, Hilmi Ryane Lekbir
- Experiments
  - Tanveer Reza, Jerome Kithinji, Yehia Metwally, Hilmi Ryane Lekbir
- Validation
  - Tanveer Reza, Jerome Kithinji, Yehia Metwally, Hilmi Ryane Lekbir
- Visualization
  - Tanveer Reza, Jerome Kithinji, Yehia Metwally, Hilmi Ryane Lekbir
- Writing
  - Hilmi Ryane Lekbir
- Review and editing
  - Tanveer Reza, Jerome Kithinji, Yehia Metwally

References

[1] Farouq Al-Omari, Chanchal K Roy, and Tonghao Chen. Semanticclonebench: A semantic code clone benchmark using crowd-source knowledge. In *2020 IEEE 14th International Workshop on Software Clones (IWSC)*, pages 57–63. IEEE, 2020.

[2] Ajmain I Alam, Palash R Roy, Farouq Al-Omari, Chanchal K Roy, Banani Roy, and Kevin A Schneider. Gptclonebench: A comprehensive benchmark of semantic clones and cross-language clones using gpt-3 model and semanticclonebench. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–13. IEEE, 2023.

[3] Afnan A. Almatrafi, Fathy A. Eassa, and Sanaa A. Sharaf. Code clone detection techniques based on large language models. *IEEE Access*, 12:40301–40315, 2024.

[4] Micheline Bénédicte Moumoula, Abdoul Kader Kabore, Jacques Klein, and Tegawendé Bissyande. Large language models for cross-language code clone detection. *arXiv e-prints*, pages arXiv–2408, 2024.

[5] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[6] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[7] Ryutaro Inoue and Yoshiki Higo. Improving accuracy of llm-based code clone detection using functionally equivalent methods. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. IEEE, 2024.

[8] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *2009 IEEE 31st International Conference on Software Engineering*, pages 485–495. IEEE, 2009.

[9] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. Supervised contrastive learning. *Advances in neural information processing systems*, 33:18661–18673, 2020.

[10] Jia Li, Chongyang Tao, Zhi Jin, Fang Liu, and Ge Li. Zc 3: Zero-shot cross-language code clone detection. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 875–887. IEEE, 2023.

[11] Rakesh Mehrotra, Pankaj Chauhan, Sayan Nandi, and Naouel Moha. Improving cross-language code clone detection via code representation learning and graph neural networks. In *Proceedings of the 46th International Conference on Software Engineering (ICSE)*. ACM, 2024.

[12] Yehia Metwally and Jerome Kithinji. Code clone detection using zc3 llm model: Replication package. `https://github.com/yehiametwally55/Code-clone-detection-using-ZC3-LLM-model`, 2024. Accessed: YYYY-MM-DD.

[13] Yehia Metwally and Jerome Kithinji. yehiametwally55/code-clone-detection-using- zc3-llm-model: Zc3 replication package, April 2025. `https://doi.org/10.5281/zenodo.15243978`.

[14] Kawser Wazed Nafi, Tonny Shekha Kar, Banani Roy, Chanchal K Roy, and Kevin A Schneider. Clcdsa: cross language code clone detection using syntactical features and api documentation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1026–1037. IEEE, 2019.

[15] Subroto Nag Pinku, Debajyoti Mondal, and Chanchal K Roy. On the use of deep learning models for semantic clone detection. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 512–524. IEEE, 2024.

[16] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*, 2021.

[17] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training.(2018), 2018.

[18] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[19] Chenning Tao, Qi Zhan, Xing Hu, and Xin Xia. C4: Contrastive cross-language code clone detection. In *Proceedings of the 30th IEEE/ACM international conference on program comprehension*, pages 413–424, 2022.

[20] Huihui Wei and Ming Li. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *IJCAI*, pages 3034–3040, 2017.

[21] Zhipeng Xue, Zhijie Jiang, Chenlin Huang, Rulin Xu, Xiangbing Huang, and Liumin Hu. Seed: Semantic graph based deep detection for type-4 clone. In *International Conference on Software and Software Reuse*, pages 120–137. Springer, 2022.

[22] Ziming Zhang and Venkatesh Saligrama. Zero-shot learning via semantic similarity embedding. In *Proceedings of the IEEE international conference on computer vision*, pages 4166–4174, 2015.

[23] Mingjie Zhu, Anirudh Jain, Karthik Suresh, Raghu Ravindran, Shreyas Tipirneni, and Chandan K. Reddy. XLCoST: A benchmark dataset for cross-lingual code intelligence. *arXiv preprint arXiv:2206.08474*, 2022.