

An Empirical Study of LLM-Based Code Clone Detection

Wenqing Zhu*
zhuwqing1995@ertl.jp
Nagoya University
Nagoya, Japan

Norihiro Yoshida
norihiro@fc.ritsumei.ac.jp
Ritsumeikan University
Osaka, Japan

Eunjong Choi
echoi@kit.ac.jp
Kyoto Institute of Technology
Kyoto, Japan

Yutaka Matsubara
yutaka@ertl.jp
Nagoya University
Nagoya, Japan

Hiroaki Takada
hiro@ertl.jp
Nagoya University
Nagoya, Japan

Abstract

Large language models (LLMs) have demonstrated remarkable capabilities in various software engineering tasks, such as code generation and debugging, because of their ability to translate between programming languages and natural languages. Existing studies have demonstrated the effectiveness of LLMs in code clone detection. However, two crucial issues remain unaddressed: the ability of LLMs to achieve comparable performance across different datasets and the consistency of LLMs' responses in code clone detection. To address these issues, we constructed seven code clone datasets and then evaluated five LLMs in four existing prompts with these datasets. The datasets were created by sampling code pairs using their Levenshtein ratio from two different code collections, CodeNet and BigCloneBench. Our evaluation revealed that although LLMs perform well in CodeNet-related datasets, with o3-mini achieving a 0.943 F1 score, their performance significantly decreased in BigCloneBench-related datasets. Most models achieved a high response consistency, with over 90% of judgments remaining consistent across all five submissions. The fluctuations of the F1 score affected by inconsistency are also tiny; their variations are less than 0.03.

CCS Concepts

• **Software and its engineering** → *Software maintenance tools*.

Keywords

Code Clone Detection, Large Language Model, Benchmark Testing

ACM Reference Format:

Wenqing Zhu, Norihiro Yoshida, Eunjong Choi, Yutaka Matsubara, and Hiroaki Takada. 2025. An Empirical Study of LLM-Based Code Clone Detection. In . ACM, New York, NY, USA, 11 pages. <https://doi.org/XXXXXXX.XXXXXXX>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, Washington, DC, USA

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/2018/06
<https://doi.org/XXXXXXX.XXXXXXX>

1 Introduction

Code clones are pairs of identical or similar code fragments. These clones are generally created when developers reuse functionality by copying and pasting existing source code [1, 3, 19, 21]. They can be classified into syntactic and semantic clones [19]. Syntactic clones have high syntactic similarity, while semantic clones may have low syntactic similarity but identical functionality. Detecting code clones has traditionally been treated as a classification problem [8, 20, 22]. Traditional code clone detectors generally employ threshold-based similarity measurements to identify code fragments as code clones. While these detectors effectively identify syntactic clones, they struggle to detect semantic clones because the same functionality can be implemented using different syntax.

In recent years, the Large Language Model (LLM) has demonstrated notable performance across various software engineering tasks, including code generation, debugging, and comprehension [14, 16, 18, 23, 25, 26, 29]. LLM has the potential for accurate code clone detection as it can capture deeper relationships between code clones, even when their syntax differs. LLM-based code clone detectors allow developers to identify code clones using zero-shot or few-shot prompting. In this process, developers provide a prompt that includes a pair of code fragments and ask the model to evaluate their similarity. The model then generates an output, which is interpreted as a Boolean value either directly or with the help of a discriminator to determine whether the code fragments are clones.

Previous studies have demonstrated the effectiveness of LLMs in code clone detection [6, 9]. Khajezade et al. demonstrated the effectiveness of *GPT-3.5-turbo* with their proposed prompts in detecting code clones, particularly for Java-Java and Java-Ruby clone pairs [9]. Dou et al. evaluated the performance of LLMs with prompt engineering techniques and found that *GPT-3.5-turbo* and *GPT-4* outperform traditional code clone detectors in identifying semantic code clones [6]. While these studies highlight the potential of LLMs for code clone detection, a deeper understanding of their capabilities in detecting code clones is necessary. In particular, evaluating their generalization ability and response consistency is crucial to ensure the practical application.

Regarding generalization, while Deep Learning (DL)-based clone detectors perform well in detecting semantic clones [7, 12, 32], they struggle to generalize across diverse datasets [4]. Since LLMs are trained on significantly larger datasets than DL models, they may offer better generalization. However, this potential still requires thorough investigation. Similarly, response consistency is a critical

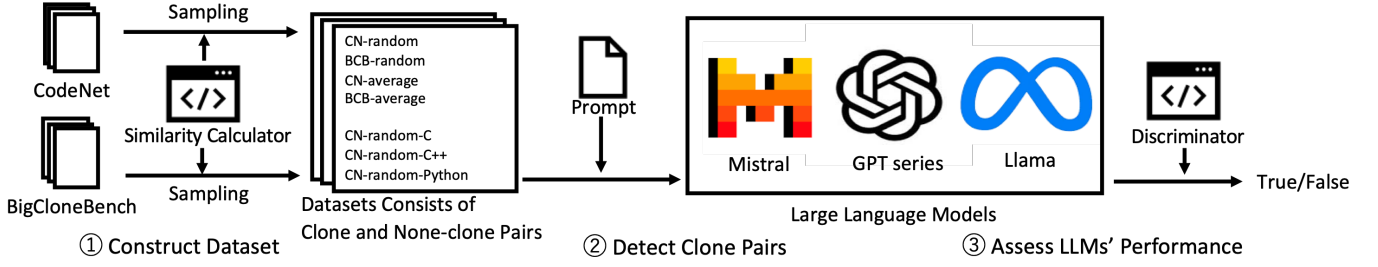


Figure 1: Experiment Process

factor. LLMs are designed for contextual generation, generating varied outputs rather than identical responses to the same input. However, for code clone detection, judgments for each candidate must remain stable and deterministic. Therefore, investigating the response consistency of LLM in detecting code clones is necessary for reliable detection. Despite their importance, these issues have not been thoroughly investigated.

To address these issues, this study investigates LLM-based code clone detectors to answer the following two Research Questions (RQs):

RQ1: How accurately does each LLM detect code clones across different datasets?

RQ2: How consistent are the responses from each LLM when presented with identical input?

To answer these RQs, we constructed seven datasets in various programming languages (four in Java and one each in Python, C, and C++) by selecting clone and non-clone pairs from two commonly used code clone evaluation datasets *CodeNet* (CN) [17] and *BigCloneBench* (BCB) [24]. In these datasets, all code pairs were labeled according to their syntactic similarity. For Java, the datasets were divided into two types based on similarity distribution to ensure a fair comparison of performance across datasets. Furthermore, we selected five LLMs (i.e., *o3-mini*, *GPT-4o*, *GPT-4o-mini*, *Llama 3.1*, and *Mistral*) and evaluated their performance in code clone detection. The prompts were chosen based on existing studies [6, 9]. The evaluation metrics included recall, precision, F1 score, and response consistency rate. Each model-prompt combination was evaluated in different temperature settings (a parameter to control the randomness of LLM's output) and attempt times. The evaluation results show that LLMs demonstrated superior performance on CN-derived datasets than on BCB, and most models exhibited high response consistency with minimal performance variations.

The contributions of this study are listed below:

- All models demonstrated significantly higher performance on datasets derived from CN, with certain model-prompt combinations achieving F1 scores above 0.9. However, on datasets derived from BCB, most model-prompt pairs could not accurately detect clones effectively in low-similarity ranges
- All LLMs, except *Llama 3.1*, exhibited high response consistency, with response inconsistencies having minimal impact on F1 scores across models. Additionally, prompt selection

typically affected response consistency more than the temperature settings of the LLMs.

- The temperature value of LLM has a small influence on code clone detection on both response consistency and F1 score.
- For the replicability of this study, we have published seven datasets covering four languages¹. Besides, our scripts for accessing LLMs for clone detection and analyzing the results are also included.

2 Related Work

Traditionally, code clone detection has been treated as a classification problem. Traditional code clone detectors typically transform the input code into intermediate representations and compare the syntactic similarity based on a predefined threshold to identify potential clones [10, 15, 20, 22, 27, 34]. They effectively detect syntactic clones but struggle with semantic clones because code fragments that implement the same functionality may exhibit significant syntactic variation [22, 24, 28, 30, 35].

To evaluate code clone detectors, several code clone datasets have been developed to measure the accuracy and performance of code clone detectors [2], [17], [24], [35], [33]. One such dataset, *BCB*, is constructed from commonly implemented Java functions mined from open-source software. It also provides similarity data for clone pairs [24]. Other datasets are constructed using correct code submissions for identical problems from programming competitions. For example, Google Code Jam [33] provides Java submissions, and CN [17] encompasses submissions across 55 different programming languages. More recently, *SemanticCloneBench* was constructed by collecting semantic clones from Stack Overflow answers [2]. Similarly, Zhu et al. [35] published code clone datasets supporting four programming languages, including 14 problems from CN. This dataset contains clone pairs with precomputed Levenshtein ratios to reduce computational time.

Over the past decade, DL-based code clone detectors have shown promise in detecting semantic clones by capturing deeper relationships between code clones beyond syntax [7, 12, 28, 30, 32]. For example, Li et al. introduced *CCLearner*, a clone classifier trained on tokenized data using a deep neural network model [12]. Zhang et al. proposed *ASTNN*, which segments abstract syntax trees into sequences of subtrees and encodes them into vectors for code clone detection [32]. Feng et al. proposed *CodeBERT*, a BERT-based

¹All our datasets, programs, and results can be accessed by: <https://zenodo.org/records/15019503>

Table 1: Detail of Datasets Used in This Work

Language	Dataset	Code pairs in each similarity range												Sum		RQ
		[0,0.2)		[0.2,0.4)		[0.4,0.6)		[0.6,0.8)		[0.8,1.0)		[1.0,1.0]				
		TC	FC	TC	FC	TC	FC	TC	FC	TC	FC	TC	FC	TC	FC	
Java	CN-random	140	56	140	399	140	242	140	3	140	0	-	-	700	700	RQ1,2
	BCB-random	140	56	140	399	140	242	140	3	140	0	-	-	700	700	RQ1
	CN-average	140	140	140	140	140	140	140	0	140	0	140	0	840	420	RQ1
	BCB-average	140	140	140	140	140	140	140	140	118	118	140	0	818	678	RQ1
C	CN-random-C	140	78	140	368	140	251	140	3	118	0	-	-	700	700	RQ1
C++	CN-random-C++	140	85	140	453	140	159	140	3	118	0	-	-	700	700	RQ1
Python	CN-random-Python	140	49	140	395	140	254	140	2	118	0	-	-	700	700	RQ1

CN: CodeNet BCB: BigCloneBench TC: clones FC: non-clones

Table 2: Prompts evaluated in this study

ID	Prompt	Ref.
<i>P0</i>	Do code 1 and code 2 solve identical problems with the same inputs and outputs? Answer with yes or no and no explanation.	[9]
<i>P1</i>	Please analyze the following two code snippets to assess their similarity and determine if they are code clones. Provide a similarity score between 0 and 10, where a higher score indicates more similarity. Additionally, identify the type of code clone they represent and present a detailed reasoning process for detecting code clones. The response should be 'yes' or 'no'.	[6]
<i>P2</i>	Please provide a detailed reasoning process for detecting code clones in the following two code snippets. Based on your analysis, respond with 'yes' if the code snippets are clones or 'no' if they are not.	
<i>P3</i>	Please analyze the following two code snippets for code clone detection. 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'.	

pre-trained model designed to understand and generate both programming and natural languages, with performance improvements achievable through the pretraining-finetuning paradigm[7]. However, several studies have shown that these models exhibit limited generalizability [5, 13]. For instance, Choi et al. found that when CCLearner, ASTNN, and CodeBERT were fine-tuned on one code clone dataset and tested on different datasets, their F1 and MCC-scores dropped significantly [5]. This limitation hinders the practical applicability of these models.

LLM-based code clone detectors have emerged as promising solutions for detecting semantic clones. Dou et al. proposed several prompts and evaluated various LLMs' clone detection performance using BCB[6]. Their results showed that *GPT-3.5-turbo* and *GPT-4* outperformed traditional techniques in detecting semantic clones. Khajezade et al. compared *GPT-3.5-turbo* with three pre-trained model-based methods (i.e., CodeBERT, RoBERTa, GraphCodeBERT) and found that *GPT-3.5-turbo* achieved the highest recall in Java-Java detection and the highest F1 score in Java-Ruby detection[9]. Despite these advancements, research has yet to explore LLM-based clone detectors' generalization and response consistency, which are crucial for their practical application.

3 Research Questions

As explained in Section 2, although LLM-based code clone detectors have shown promising results, their generalization across different datasets and response consistency remain underexplored. To address these issues, this study aims to answer the following two RQs:

RQ1: How accurately does each LLM detect code clones across different datasets?

This RQ aims to examine LLMs' accuracy in consistently detecting code clones across different datasets. Unlike traditional DL-based code clone detectors that struggle with generalization, LLMs may offer better generalization because they are trained on a much broader and more diverse corpus, which includes code from various sources, languages, and contexts. This potential advantage could significantly enhance their real-world applicability.

RQ2: How consistent are the responses from each LLM when presented with identical input?

This RQ aims to investigate the consistency of results produced by LLM-based code clone detectors when given the same input multiple times. Due to their non-deterministic nature, LLMs can generate different outputs for the same input. Ensuring high response consistency is crucial for practical applications of LLM-based clone

detection. Moreover, reproducibility is essential in software engineering research, as many tasks depend on consistent and reliable clone detection.

4 Methodology

This section presents the evaluation methodology used to answer the two RQs discussed in Section 2. Figure 1 illustrates the overall experimental process. First, datasets containing both clone and non-clone code pairs were constructed across four programming languages. Next, each code pair was formatted into a predefined prompt and submitted to the LLMs for code clone detection. Finally, the performance of the LLMs was evaluated using various performance metrics.

4.1 Clone Datasets

Motivation for Constructing Datasets: Existing code clone datasets have two significant limitations: they lack diversity in programming languages beyond Java and do not account for syntactic similarity when selecting clone pairs. We constructed seven diverse code clone datasets to address these limitations and answer RQ1 and RQ2. These datasets cover a range of programming languages and syntactic similarities, ensuring they better reflect the diverse types of code clones developers encounter in practice. The primary motivations behind constructing these datasets were to incorporate syntactic similarity as a key factor influencing the complexity of code clone detection and ensure fair comparisons across LLMs by controlling the distribution of syntactic similarity within the datasets. By carefully curating clone pairs with varying degrees of similarity, we could report F1 scores for the LLMs across different similarity levels, providing valuable insights into their performance.

Dataset Construction Methodology: Rather than random sampling, we selected clone pairs based on their similarity using the Levenshtein ratio[31] on identifier-normalized token sequences, which is defined by the following equation:

$$LevRatio(A, B) = \frac{Max(|A|, |B|) - LevTS(A, B)}{Max(|A|, |B|)} \quad (1)$$

Here, $LevTS(A, B)$ represents the Levenshtein distance between two token sequences, A and B , with identifiers normalized [11]. We chose this similarity measurement for two key reasons. It enables the creation of balanced datasets by ensuring an even distribution of clone pairs across different similarity levels. The Levenshtein ratio ensures that semantic clones are spread across the similarity range, whereas syntactic clones tend to cluster near 100% similarity. Second, it was also adopted by Zhu et al. [35] to categorize code pairs within the *CN*. By using the same established approach, we enhanced the reliability and consistency of our similarity measurements across a wide range of levels, thus strengthening the robustness of our evaluation.

Constructed Datasets: We constructed seven datasets, in which *CN-random* is the primary dataset for most experiments. The other six datasets allow for comparisons of LLM performance across different datasets, similarity ranges, and programming languages.

- ***CN-random*** consists of 700 clone pairs and 700 non-clone pairs in Java, which were randomly sampled from 14 subsets of Zhu et al.’s dataset [35]. Clone pairs originate from code

submissions to the same programming competition problems, while non-clone pairs were sampled from different problems. Clones in this dataset were distributed across 20% similarity intervals, with 10 pairs randomly selected from each range based on the similarity distribution. Non-clone pairs were evenly sampled from different subsets to ensure diversity.

- ***BCB-random*** was constructed from *BCB* to have identical similarity distributions with *CN-random*, ensuring comparable distributions across datasets. Since *BCB* uses a different similarity measure than that of *CN*, clone pairs in *BCB* generally have lower similarity values. To address this problem, we adjusted the similarity ranges to match those used in *CN*, recalculated the similarity for each code pair, and re-sampled any pairs that did not satisfy the required similarity threshold.
- ***CN-average*** was constructed to compute F1 scores within each similarity range, including an equal number of clone and non-clone pairs in each range. Initially, we focus on constructing a single dataset for all experiments, avoiding the necessity of separating sampling strategies. However, obtaining sufficient non-clones for similarity ranges above 60% was challenging. Therefore, for similarity ranges below 0.6, we sampled 140 clone and non-clone pairs. For ranges above 0.6, we sampled only 140 clone pairs. Recall metrics are reported exclusively for the ranges that contain only clone pairs.
- ***BCB-average***, similar to *CN-average*, includes an equal number of clones and non-clones pairs in higher similarity ranges (above 0.6). This reflects the higher proportion of high-similarity non-clones in *BCB*.
- ***CN-random-C*, *CN-random-C++*, *CN-random-Python*** were constructed using the same sampling strategy with *CN-random*, but include clone pairs in C, C++, and Python, respectively. Similar to *CN-random*, clones in these datasets were distributed across 20% similarity intervals, with 10 pairs randomly selected from each range. Non-clone pairs were sampled evenly from different subsets to maintain diversity.

Table 1 presents the number of clone and non-clone pairs collected within each Levenshtein ratio range in all the constructed datasets. As depicted in this table, *CN-random* and *BCB-random* have identical distributions, whereas *CN-average* and *BCB-average* contain an equal number of clones and non-clones pairs across as many similarity ranges as possible.

4.2 Models and Prompting

This study evaluated five LLMs, including a reasoning model named *o3-mini*, two models from the GPT-series (*GPT-4o*, and *GPT-4o-mini*), and two open-source LLMs, *Llama 3.1* and *Mistral*.

- ***o3-mini*** is one of the latest reasoning models of OpenAI. Reasoning models produce a long internal chain of thought before their response and aim at more complex tasks such as coding and scientific reasoning. *o3-mini* was selected to investigate the performance of the reasoning model in clone detection.

- **GPT-4o** is one of the latest and most advanced models in the GPT series. This study uses *GPT-4o-2024-08-06*, which is the most recent available at the time. We selected this model to investigate whether the advancements in the model can enhance performance in code clone detection.
- **GPT-4o-mini** is the lightweight version of *GPT-4o*. Evaluating this model allows us to assess whether the mini version exhibits performance degradation in the code clone detection task. *GPT-4o-mini* points to *GPT-4o-mini-2024-07-18*. We used this model to examine the trade-off between model size and performance, particularly in code clone detection.
- **Llama 3.1:8B²** is an open-source LLM released by Meta. We chose this model because it provides a lower-cost alternative to proprietary models, and understanding its performance can provide insights into whether open-source models can match or outperform commercial alternatives. The specific version in this study is *Llama 3.1:8B*, which has 8 billion parameters.
- **Mistral:7B³** is another open-source LLM released by Mistral AI, but with a smaller parameter size (7 billion parameters). *Mistral* was chosen to evaluate whether smaller open-source models can still perform well compared with larger commercial models. The version used in this study is *Mistral:7B*.

We utilized Batch API⁴ of Open AI to access their models, while *Llama 3.1* and *Mistral* were run locally using oLlama⁵. All models were evaluated in their original, non-fine-tuned form.

We employed four distinct prompts to conduct code clone detection with these LLMs, as detailed in Table 2. The first prompt, *P0*, was identified as the optimal prompt in the study by Khajezade et al. [9]. The remaining prompts, *P1*, *P2*, and *P3*, are sourced from the study by Dou et al.,. In their original study, *P3* is a prompt with relatively high recall, *P2* is a prompt with relatively high precision, and *P1* is a prompt that balances recall and precision [6]. Prompt *P0* asks LLMs to judge if two codes solve identical problems, while the other three prompts ask LLMs to assess the code pair's similarity to make the judgment.

4.3 Performance Evaluation for Code Clone Detection

In this study, the performance of LLMs for code clone detection was evaluated based on their ability to provide binary responses, specifically 'yes' or 'no'. To automate the collection of results of code clone detection, each selected prompt, described in Section 4.2, was designed to instruct the LLMs to return only these two responses. However, LLMs occasionally deviated from this instruction by including additional text, such as reasoning, in their responses. To address this problem, a discriminator program was implemented to process and convert the responses into Boolean results. The discriminator accepts variations such as "Result: Yes" or "Yes, ..." but excludes more complex responses, even if they contain the correct answer, to avoid recognition problems.

The temperature parameter (*temp*) is a critical factor in determining the randomness of LLMs' output. Lower *temp* values generally lead to more deterministic and consistent responses, whereas higher *temp* values introduce greater variability in the output. In RQ1, we set the *temp* value as 0.3 to ensure consistency with the existing study [9]. As a relatively low value, setting *temp* to 0.3 is considered to balance answer consistency and detection ability. In RQ2, we measured LLMs' response consistency rate and F1 score for various *temp* settings. Because the effect of *temp* depends on the model's architecture, *temp* values are incomparable across different LLMs. Besides, *o3-mini* does not support the temperature parameter.

For performance evaluation, the study uses the commonly adopted metrics of recall, precision, and F1 score. Recall is calculated as $\frac{TP}{TP+FN}$, whereas precision is defined as $\frac{TP}{TP+FP}$. F1 score (harmonic mean of recall and precision) is computed as $\frac{2 \times \text{Recall} \times \text{Precision}}{\text{Recall} + \text{Precision}}$. Here, *TP*, *FP*, *TN*, and *FN* represent the true positives, false positives, true negatives, and false negatives, respectively. Specifically, *TP* refers to the intersection of true clones and detected clones, *FP* is the intersection of false clones and detected clones, *TN* refers to the intersection of false clones and non-detected clones, and *FN* is the intersection of true clones and non-detected clones.

To address RQ2, we evaluate the response consistency rate (RCR) of LLMs for code clone detection. This metric reflects the models' ability to provide consistent outputs for identical inputs across repeated submissions. The response consistency rate for *i*-th submission RCR_i is defined in Equation 2.

$$RCR_i = \frac{1}{N} \sum_{j=1}^N \prod_{k=2}^i \mathbf{1}\{r_k(j) = r_1(j)\}, \mathbf{1}\{A\} = \begin{cases} 1, & A \text{ is true} \\ 0, & \text{otherwise} \end{cases} \quad (2)$$

Here, $r_k(j)$ represents the judgment for the *j*-th code pair in the *k*-th submission. Given that there are *N* code pairs in the test dataset, RCR_i is defined as the proportion of cases in that submission where the judgment has remained unchanged with the first submission.

5 Answers to the RQs

5.1 RQ1: How accurately does each LLM detect code clones across different datasets?

To address this RQ, we conducted experiments on our constructed datasets to evaluate the code clone detection performance of various LLMs. First, we analyzed the performance on *CN-random*, the main dataset used in almost all experiments, to assess LLMs' ability to detect clones in a dataset with diverse similarity ranges. To further investigate the impact of dataset characteristics, we compared the detection performance of LLMs on *CN-random* and *BCB-random*, which share identical similarity distributions but originate from different code collections. This comparison helped us examine whether models maintain consistent performance across datasets. Furthermore, we assessed the performance on datasets covering four programming languages from *CN* to determine whether models generalize well across different languages. Finally, we compared LLM's performance on *CN-average* and *BCB-average* to analyze detection accuracy across each similarity range.

Results on the Main Dataset: *CN-random*

The results of code clone detection on *CN-random* are listed in Table 3. Among all models, *o3-mini* achieved the highest overall

²Model ID in oLlama platform: 42182419e950

³Model ID in oLlama platform: f974a74358d6

⁴<https://platform.openai.com/docs/guides/batch>

⁵<https://oLlama.com/>

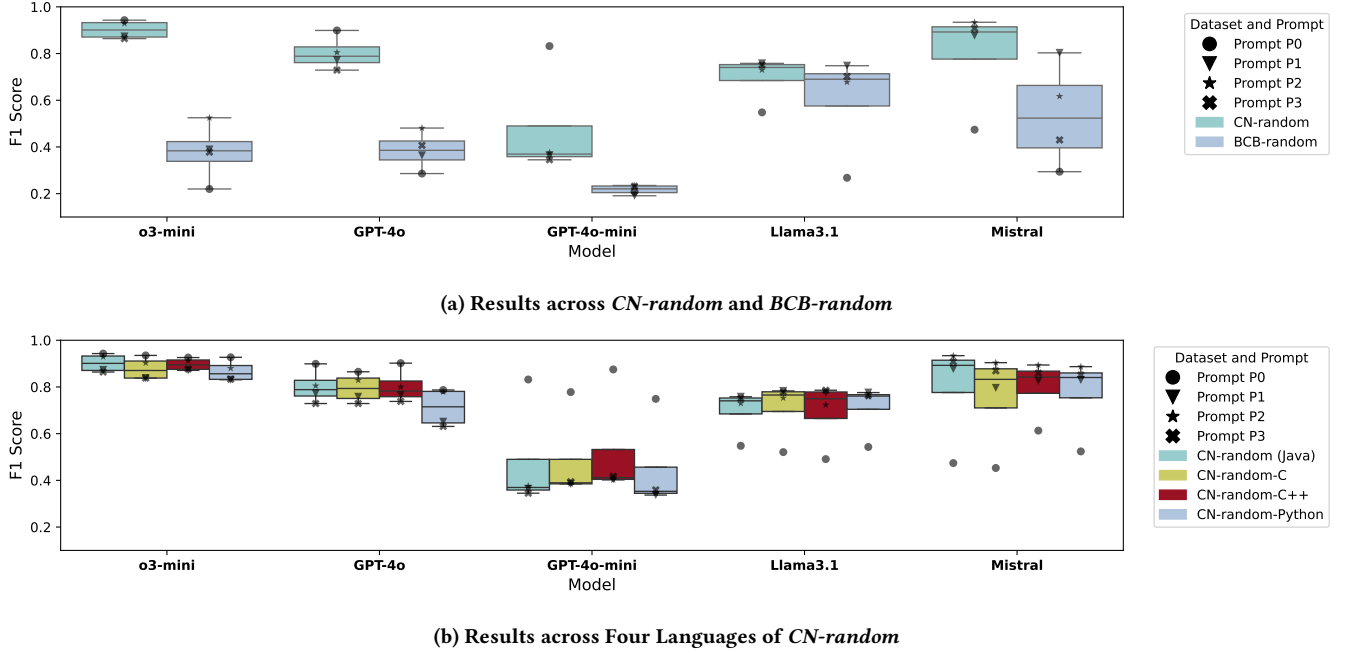


Figure 2: F1-Score of Each Model and Prompt for Different Datasets

F1 score of 0.943 when using prompt *P0* from Table 2. *Mistral* also achieved a 0.934 F1 score in prompt *P2*. The *GPT-4o* and *GPT-4o-mini* also performed well, with their best F1 scores of 0.899 and 0.832, respectively. In contrast, *Llama 3.1* exhibited the lowest performance, with a maximum F1 score of 0.758. Additionally, we observed the following two findings:

- **Performance variation across prompts for the same model:** For all five models, the difference between the highest and lowest F1 scores across the four prompts was substantial, with the slightest difference being 0.17. *GPT-4o-mini* and *Mistral* revealed even more significant differences, with gaps of 0.49 and 0.46, respectively.
- **Performance variation across models for the same prompt:** Prompt *P0*, which achieved F1 scores above 0.8 with the GPT-series models, yielded only 0.548 and 0.474 F1 scores with *Llama 3.1* and *Mistral*. In contrast, prompt *P2*, which produced the highest overall F1 score with *Mistral* (0.934), achieved only 0.376 with *GPT-4o-mini*.

These results highlight that the choice of prompt significantly impacts model performance, suggesting that selecting the optimal model-prompt combination is more crucial than simply choosing the most powerful model.

Comparison Between *CN-random* and *BCB-random*

To evaluate the impact of dataset differences on clone detection, we compared model results between *CN-random* and *BCB-random*. Figure 2(a) presents a comparison of the F1 score for *CN-random* and *BCB-random*, revealing that all models achieved considerably lower F1 scores on *BCB-random* than *CN-random*. For each model, the F1 score across all prompts on *BCB-random* was consistently lower than on *CN-random*. To quantify these differences, Table 4

presents the F1 score differences between the two datasets for each prompt. The most significant relative drop was observed with *o3-mini*, where the average F1 score decreased by 0.52. In contrast, only four model-prompt combinations had F1 score differences smaller than 0.1. The smallest relative drop was observed with *Llama 3.1*, with an average F1 score decrease of only 0.1 across all prompts. These results demonstrated that most model-prompt combinations could not achieve the same level of performance on *CN-random* and *BCB-random*, indicating the challenge of achieving consistent performance across different code collections, even when similarity distributions are aligned.

Comparison Among *CN-random* in Four Languages

Figure 2(b) presents the results of each LLM for four languages of *CN-random*. The four boxes from left to right for each model correspond to Java, C, C++, and Python. Unlike the cross-dataset comparison, all models exhibited similar performance across the four programming languages. This indicates that the models can generalize well across different programming languages within the *CN-random* datasets, maintaining stable performance. Thus, language-specific variations within *CN* have a more negligible impact on performance than the differences between *CN* and *BCB*.

Comparison Between *CN-average* and *BCB-average*

Figure 3 presents recall, precision, and F1 scores across different similarity ranges, solid lines representing *CN-average* results, and dashed lines representing *BCB-average* results. In this figure, different prompts are distinguished by color and marker type. For *CN-average*, F1 scores are reported for similarity ranges lower than 0.6, whereas for *BCB-average*, the ranges are reported for ranges lower than 1.0. In the other ranges, only recall is reported due to the limited number of non-clones available. Furthermore, when all

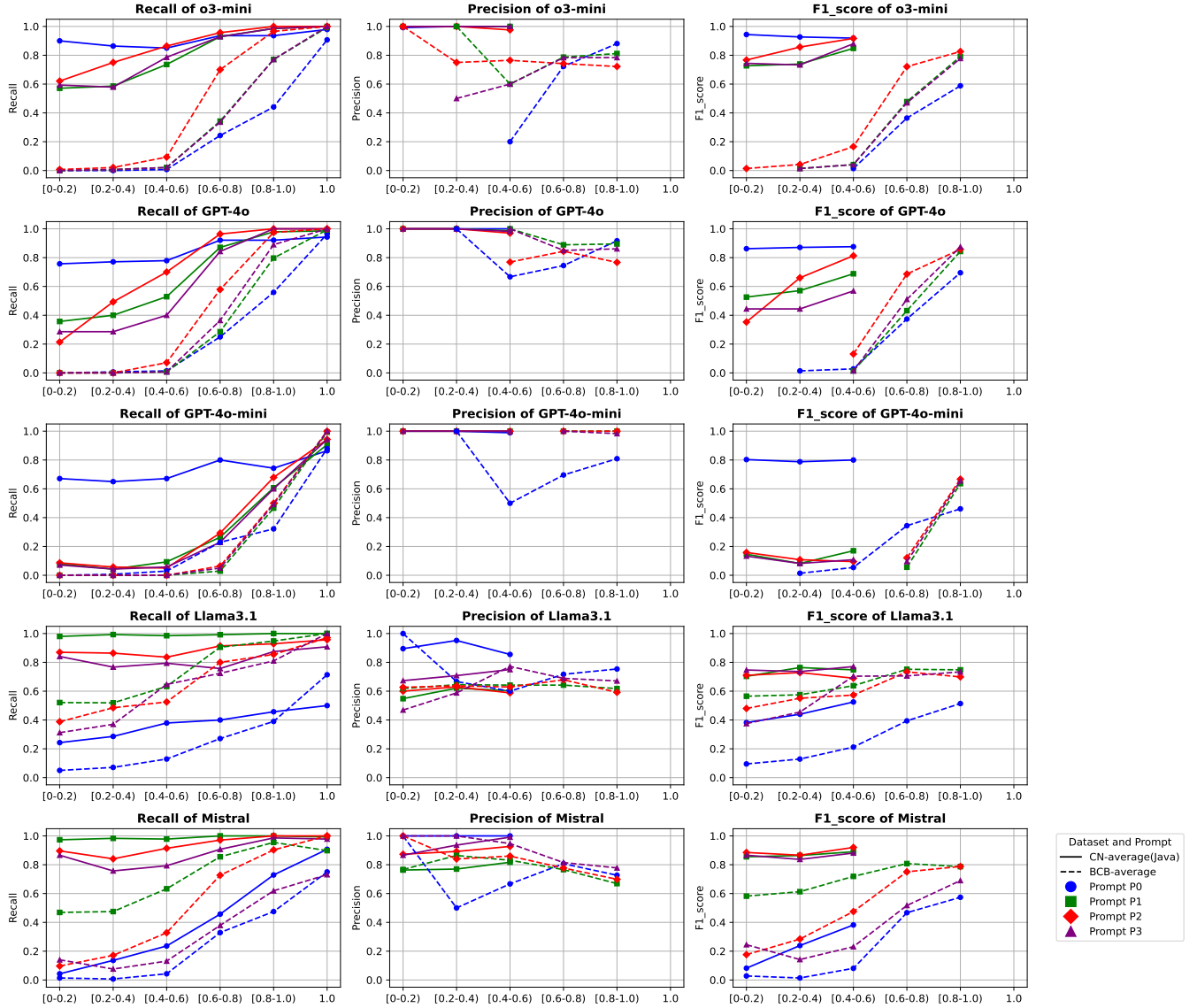


Figure 3: Recall, Precision, and F1-Score of Each Model and Prompt in Each Similarity Range

candidate code pairs were classified as non-clones, precision and F1 score could not be calculated (e.g., *o3-mini* with prompt *P0* in the second range of *BCB-average*).

The results indicate that higher similarity code clones are easier to detect, aligning with expectations. Most model-prompt combinations exhibit higher recall for code pairs with higher similarity. Furthermore, all models demonstrate significantly higher recall for *CN-average* compared to *BCB-average*. In areas where code similarity is below 0.8, nearly all model-prompt combinations show a noticeably higher recall for *CN-average* than for *BCB-average*. However, an exception is observed in *GPT-4o-mini*, where other prompts yield similar recall across both datasets in all regions except for Prompt *P0*. It is also noteworthy that some prompt-model combinations fail to achieve 1.0 recall even in regions where similarity is 1.0

(T1 and T2 clones). For instance, Prompt *P0* does not attain 1.0 recall in this region across any model. This may be due to Prompt *P0* emphasizing functional equivalence in code clones, leading to Type-2 clones with different function names to be misclassified as non-clones. Additionally, recall for Prompt *P0* varies significantly across models, with recall on *GPT-4o* nearly twice that on *Llama 3.1*. This result suggests that selecting the most suitable prompt is essential for achieving optimal performance across different models.

Another notable observation is that non-clones with high similarity are more likely to be misclassified as clones. However, the precision of various models across high- and low-similarity code pairs does not show as large a disparity as recall does. Each model demonstrates cases where precision is higher in regions of higher similarity. Overall, the models maintained relatively high precision,

Table 3: Results of Each Prompt and Model for *CN-random*

Model	PromptId	Recall	Precision	F1
<i>o3-mini</i>	<i>P0</i>	0.891	1.000	0.943
	<i>P1</i>	0.774	1.000	0.873
	<i>P2</i>	0.873	0.993	0.929
	<i>P3</i>	0.760	1.000	0.864
<i>GPT-4o</i>	<i>P0</i>	0.817	1.000	0.899
	<i>P1</i>	0.630	0.998	0.772
	<i>P2</i>	0.677	0.994	0.805
	<i>P3</i>	0.574	0.998	0.729
<i>GPT-4o-mini</i>	<i>P0</i>	0.714	0.996	0.832
	<i>P1</i>	0.221	1.000	0.363
	<i>P2</i>	0.231	1.000	0.376
	<i>P3</i>	0.209	1.000	0.345
<i>Mistral</i>	<i>P0</i>	0.311	0.995	0.474
	<i>P1</i>	0.990	0.787	0.877
	<i>P2</i>	0.938	0.929	0.934
	<i>P3</i>	0.862	0.958	0.908
<i>Llama3.1</i>	<i>P0</i>	0.384	0.954	0.548
	<i>P1</i>	0.997	0.611	0.758
	<i>P2</i>	0.889	0.619	0.730
	<i>P3</i>	0.794	0.713	0.751

Table 4: Mean difference of F1 score for *CN-random* and *BCB-random*(Difference of F1 score: F1 score for *CN-random* - F1 score for *BCB-random*)

Model	Difference of F1 score				Mean
	<i>P0</i>	<i>P1</i>	<i>P2</i>	<i>P3</i>	
<i>o3-mini</i>	0.72	0.48	0.40	0.49	0.52
<i>GPT-4o</i>	0.61	0.41	0.32	0.32	0.42
<i>GPT-4o-mini</i>	0.62	0.17	0.14	0.11	0.26
<i>Llama 3.1</i>	0.28	0.01	0.05	0.05	0.10
<i>Mistral</i>	0.18	0.07	0.32	0.48	0.26
Mean	0.44	0.17	0.22	0.26	

with all model-prompt combinations achieving at least 0.6 precision in the most challenging area. Additionally, in comparable areas, most model-prompt combinations achieve precision for *CN-average* that is equal to or greater than for *BCB-average*. F1 scores correlate positively with code similarity, primarily due to their dependence on recall. Within comparable similarity regions, all model-prompt combinations exhibit significantly higher F1 scores for *CN-average* than for *BCB-average*.

To further examine model performance, Table 5 summarizes the highest recall and precision achieved by each model. The highest recall and precision in each range for a given model may not be achieved by the same prompt. For *CN-average*, *Llama 3.1* and *Mistral* achieve recall above 0.97 even in similarity ranges lower than 0.6, considerably outperforming the GPT-series in these ranges. *Mistral* demonstrated slightly higher precision than *Llama 3.1*, making it the best-performing model on *CN-average*. Similarly, on *BCB-average*, although *Llama 3.1* achieved a slightly higher recall, *Mistral* demonstrated a more pronounced advantage in precision. As a result, *Mistral* emerged as the best-performing model for *BCB-average*, demonstrating a strong balance between recall and precision.

The answer to RQ1: None of the models achieved comparable detection capability across datasets derived from different code collections.

5.2 RQ2: How consistent are the responses from each LLM when presented with identical input?

We evaluated the response consistency of each LLM by submitting the same input multiple times using the *CN-random* dataset. In common with most DL models, LLMs exhibit non-determinism, often yielding variable outputs for identical inputs. This behavior is particularly sensitive to the input’s position relative to the model’s decision boundaries for that task. When the input is situated far away from decision boundaries, the resulting output is typically stable and consistent. Conversely, inputs that lie near a boundary are more likely to produce divergent or inconsistent results. In clone detection, such inconsistencies reduce reproducibility, undermining the reliability of detection results and limiting the model’s practical applicability. For this experiment, we submitted the same prompt five times per model and recorded the results. Each submission was conducted independently to avoid mutual influence. Additionally, we tested two *temperature* (*temp*) settings: 0.3 (the lowest value) and the maximum value of each model.

Figure 4 shows the proportion of responses that remained consistent with the first submission across the second, third, fourth, and fifth trials for each model-prompt combination. As shown in the figure, *GPT-4o* and *GPT-4o-mini* demonstrated the highest consistency, maintaining a rate above 90% in nearly all cases. *Mistral* exhibited considerable variation in consistency depending on the prompt, with prompt *P1* showing notably lower consistency than the others, typically lower than 0.8. *Llama 3.1*, on the other hand, exhibited the lowest consistency. By the second submission, at least 20% of responses differ from the first trial across all prompts. By the fifth submission, at least 40% of responses were inconsistent, with prompt *P3* showing extreme instability, reaching 80% inconsistency.

To analyze the factors influencing response consistency, we examine how prompt selection and *temp* settings affected consistency variation. Table 6 presents the results. The average consistency rate range affected by *temp* was calculated by finding the difference in consistency between *temp 0* and the maximum *temp* for each prompt submission and subsequently averaging the results. The average consistency rate range affected by the prompt was calculated by determining the range of consistency rates across four

Table 5: Max Recall and Precision in Each Similarity Range.
For similarity ranges lower than 0.8, almost all LLMs' max recall for *CN-average* is considerably higher than that for *BCB-average*.

Model	Dataset	Recall						Precision				
		[0,0.2)	[0.2,0.4)	[0.4,0.6)	[0.6,0.8)	[0.8,1)	[1,1]	[0,0.2)	[0.2,0.4)	[0.4,0.6)	[0.6,0.8)	[0.8,1)
<i>o3-mini</i>	<i>CN-average</i>	0.900	0.864	0.864	0.957	1.000	1.000	1.000	1.000	1.000	-	-
	<i>BCB-average</i>	0.007	0.021	0.093	0.700	0.966	1.000	1.000	1.000	0.765	0.787	0.881
<i>GPT-4o</i>	<i>CN-average</i>	0.757	0.771	0.779	0.921	0.964	1.000	1.000	1.000	1.000	-	-
	<i>BCB-average</i>	0	0.007	0.071	0.579	0.975	1.000	NaN	1.000	1.000	0.890	0.917
<i>GPT-4o-mini</i>	<i>CN-average</i>	0.671	0.650	0.671	0.800	0.743	0.943	1.000	1.000	1.000	-	-
	<i>BCB-average</i>	0	0.007	0.029	0.229	0.500	1.000	NaN	1.000	0.500	1.000	1.000
<i>Llama 3.1</i>	<i>CN-average</i>	0.980	0.993	0.985	0.992	1.000	1.000	0.895	0.952	0.855	-	-
	<i>BCB-average</i>	0.521	0.519	0.647	0.905	0.948	1.000	1.000	0.667	0.722	0.717	0.754
<i>Mistral</i>	<i>CN-average</i>	0.973	0.983	0.978	1.000	1.000	1.000	1.000	1.000	1.000	-	-
	<i>BCB-average</i>	0.469	0.475	0.633	0.855	0.955	1.000	1.000	1.000	0.947	0.815	0.778

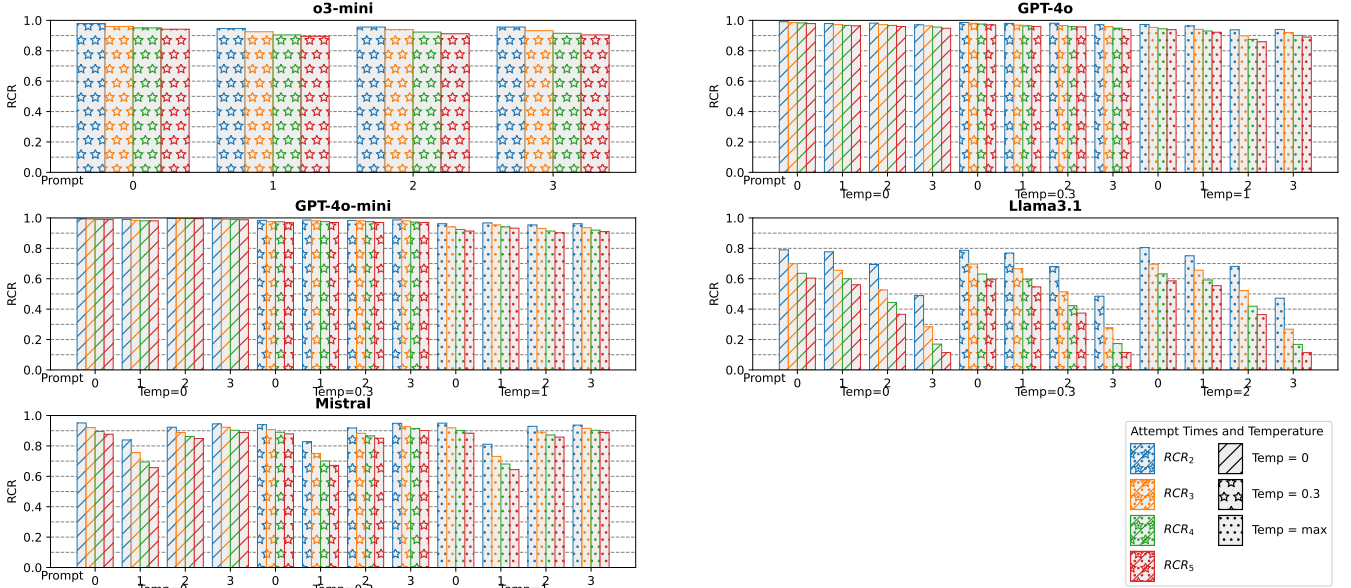


Figure 4: Response Consistency Rate (RCR) of Each Model

prompts and averaging the results. Overall, *temp* had minimal impact on consistency, with fluctuations generally below 0.08 across all models. However, prompt choice significantly influenced consistency, particularly for *Llama 3.1* and *Mistral*, which showed large variations exceeding 0.4 across different prompts. For most models, while changes in *temp* had little effect on consistency, choosing the right prompt played a critical role in maintaining stability.

Moreover, we calculated the range of F1 scores achieved by each model under various *temp* settings to assess how inconsistent responses affected overall model performance. These results are presented in Table 7. For all models, F1 score fluctuations were lower than 0.032, and the degree of F1 score variation did not significantly differ across *temp* settings. This result indicates that *temp* settings

do not significantly affect the performance of LLMs in code clone detection.

One exception in this study was *o3-mini*, which does not allow *temp* configuration. As a result, we submitted each prompt five times under its default settings. *o3-mini* maintained a response consistency rate above 0.9 for all four prompts by the fifth submission. Moreover, the differences between prompts were minimal, with response consistency rates fluctuating by only 0.040. Similarly, the variation in F1 scores across different prompts was slight, with an average F1 fluctuation range of just 0.012. These results

indicate that *o3-mini* is a highly stable model in code clone detection, maintaining consistent responses regardless of prompt choice.

The answer to RQ2: Most models' response consistency is higher-than-expected, and their inconsistency did not result in a significant fluctuation in the detection capability overall.

6 Discussions

6.1 Toward Real-world Applications of LLM-based Clone Detection

This section discusses the challenges of using LLM-based clone detection in practical applications. In Section 5, we evaluated five LLMs' accuracy and response consistency. Although each LLM exhibits high detection accuracy in CN-related datasets, the accuracy of each model in BCB-related datasets declined considerably. These results indicate that the high accuracy of these LLMs for competitive programming data cannot be reflected in open-source software. Competitive programming data has clear labels compared to open-source software, rendering learning easy for large models. In this study, we did not use fine-tuning or prompt-tuning. In future research, we should actively use these technologies to improve the accuracy of clone code detection in open-source software by large models. Regarding answer consistency, our experimental results revealed pronounced differences between models. When choosing an LLM, we should focus on the difference in answer consistency. Furthermore, the cost of using LLMs should be considered. The cost of LLMs can be categorized into time and economic costs. Regarding running time, LLMs typically require very high computing resources; otherwise, the response speed decreases considerably. Communication time costs also occur when using commercial models through APIs. Economically, this also includes the cost of using commercial models. In practice, existing clone detection technologies should be actively used to reduce the number of comparisons through LLM.

6.2 Threats to Validity

This study incorporated competitive programming data to construct datasets as the original research, where accepted code for the same problem is considered a clone. However, this ground truth may fail when using LLMs for code clone detection. For instance, competitive programming problems typically specify input data ranges and formats. For a problem with single-line input specifications, whether or not the submitted code handles multiple lines from the standard input does not affect the judgment. However, an LLM, unaware of such specifics, can classify code supporting multi-line input differently from code that does not, labeling them as non-clones. From the LLM's perspective, this judgment is accurate but would be counted as a false negative in evaluations. Therefore, the actual performance on CN-related datasets could be slightly better than the observed results.

To automatically recognize detection results from LLMs' responses, current prompts instruct the LLM to output only "yes" or "no", ignoring the reasoning or explanations it would typically provide. This approach sacrifices information and does not fully use the capabilities of LLMs. For example, the issue mentioned in the previous paragraph cannot be addressed. However, automated

Table 6: Average Response Consistency Rate Range (ω) Effected by *temp* and Prompt

Model	Effected by <i>temp</i>		Comp.	Effected by prompt	
	Prompt	ω		<i>temp</i>	ω
<i>GPT-4o</i>	<i>P0</i>	0.032	$\approx$	0	0.025
	<i>P1</i>	0.031		0.3	0.024
	<i>P2</i>	0.078		1	0.060
	<i>P3</i>	0.047			
<i>GPT-4o-mini</i>	<i>P0</i>	0.056	$\approx$	0	0.012
	<i>P1</i>	0.035		0.3	0.003
	<i>P2</i>	0.071		1	0.023
	<i>P3</i>	0.059			
<i>Llama 3.1</i>	<i>P0</i>	0.003	<	0	0.418
	<i>P1</i>	0.010		0.3	0.415
	<i>P2</i>	0.011		2	0.424
	<i>P3</i>	0.298			
<i>Mistral</i>	<i>P0</i>	-0.003	<	0	0.180
	<i>P1</i>	0.020		0.3	0.185
	<i>P2</i>	-0.007		1	0.198
	<i>P3</i>	0.003			
<i>o3-mini</i>	-	-	-	-	0.040

**Table 7: Average F1 score Range of each Model
Lowest *temp*: 0 ; Highest *temp*: 2 for *Llama 3.1* and 1 for the others.**

Model	Range of F1 score		
	Lowest <i>temp</i>	0.3	Highest <i>temp</i>
<i>GPT-4o</i>	0.012	0.014	0.012
<i>GPT-4o-mini</i>	0.009	0.009	0.021
<i>Llama 3.1</i>	0.027	0.026	0.032
<i>Mistral</i>	0.012	0.015	0.010
<i>o3-mini</i>	0.012		

result recognition remains a necessary function for code clone detectors. Improvements in this area will depend on further research advancements.

Fine-tuning has been widely demonstrated to considerably improve the performance of LLMs on specific tasks, including code clone detection. However, we chose not to fine-tune the models in this study for the following reasons. Fine-tuning LLMs using data from BigCloneBench would improve their detection accuracy on *BCB-random* and *BCB-average*. However, this study focused on evaluating whether the clone detection capabilities of LLMs can generalize to a broader range of datasets and, ultimately, to the entire codebase in the real world. Without establishing the representativeness of BigCloneBench for the entire coding domain or creating and evaluating models fine-tuned on datasets distinct from BigCloneBench and CodeNet to determine whether they maintain comparable accuracy on other datasets, the significance of achieving higher accuracy on BCB-related datasets would remain limited.

7 Conclusion

To investigate the generalization ability and response consistency of LLMs in code clone detection, a topic that has not been extensively explored in previous research, we constructed seven code clone datasets and then evaluated five LLMs on their detection performance. Our evaluation revealed two key findings. First, the LLMs demonstrated superior performance on datasets derived from CN compared to BCB, where their detection capabilities showed a notable decline. Second, most models exhibited high response consistency with minimal performance variations, suggesting that LLM-based code clone detection results are generally reproducible. Future studies should focus on the following areas: exploring whether fine-tuning can enhance LLMs' detection performance across multiple datasets and whether LLMs can achieve similar performance for relatively less common programming languages.

References

- [1] Raihan Al-Ekram, Cory Kapser, Richard Holt, and Michael Godfrey. 2005. Cloning by accident: an empirical study of source code cloning across software systems. In *2005 International Symposium on Empirical Software Engineering*, 2005. IEEE, 10–pp.
- [2] Farouq Al-Omari, Chanchal K Roy, and Tonghao Chen. 2020. Semanticclonebench: A semantic code clone benchmark using crowd-source knowledge. In *2020 IEEE 14th International Workshop on Software Clones (IWSC)*. IEEE, 57–63.
- [3] Brenda S Baker. 1993. A program for identifying duplicated code. *Computing Science and Statistics* (1993), 49–49.
- [4] Eunjong Choi, Norihiro Fuke, Yuji Fujiwara, Norihiro Yoshida, and Katsuro Inoue. 2023. Investigating the Generalizability of Deep Learning-based Clone Detectors. In *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*. 181–185. doi:10.1109/ICPC58990.2023.00032
- [5] Eunjong Choi, Norihiro Fuke, Yuji Fujiwara, Norihiro Yoshida, and Katsuro Inoue. 2023. Investigating the generalizability of deep learning-based clone detectors. In *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*. IEEE, 181–185.
- [6] Shihan Dou, Junjie Shan, Haoxiang Jia, Wenhao Deng, Zhiheng Xi, Wei He, Yueming Wu, Tao Gui, Yang Liu, and Xuanjing Huang. 2023. Towards Understanding the Capability of Large Language Models on Code Clone Detection: A Survey. arXiv:2308.01191 [cs.SE] <https://arxiv.org/abs/2308.01191>
- [7] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:2002.08155 [cs.CL] <https://arxiv.org/abs/2002.08155>
- [8] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A multilingual token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670.
- [9] Mohamad Khajezade, Jie Jw Wu, Fatemeh Hendijani Fard, Gema Rodríguez-Pérez, and Mohamed Sami Shehata. 2024. Investigating the Efficacy of Large Language Models for Code Clone Detection. In *2024 IEEE/ACM 32nd International Conference on Program Comprehension (ICPC)*. 161–165.
- [10] Rainer Koschke, Raimar Falke, and Pierre Frenzel. 2006. Clone detection using abstract syntax suffix trees. In *2006 13th Working Conference on Reverse Engineering*. IEEE, 253–262.
- [11] V Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. *Proceedings of the Soviet physics doklady* (1966).
- [12] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. 2017. CCLearner: A Deep Learning-Based Clone Detection Approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 249–260. doi:10.1109/ICSME.2017.46
- [13] Chenyao Liu, Zeqi Lin, Jian-Guang Lou, Lijie Wen, and Dongmei Zhang. 2021. Can Neural Clone Detection Generalize to Unseen Functionalities? In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 617–629. doi:10.1109/ASE51524.2021.9678907
- [14] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
- [15] Seyyed Mehdi Nasehi, Gholam Reza Sotudeh, and Maziar Gomrokchi. 2007. Source code enhancement using reduction of duplicated code. In *Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering*. 192–197.
- [16] Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. 2023. LLM is Like a Box of Chocolates: the Non-determinism of ChatGPT in Code Generation. *arXiv preprint arXiv:2308.02828* (2023).
- [17] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. arXiv:2105.12655 [cs.SE] <https://arxiv.org/abs/2105.12655>
- [18] Zeeshan Rasheed, Malik Abdul Sami, Muhammad Waseem, Kai-Kristian Kemell, Xiaofeng Wang, Anh Nguyen, Kari Systä, and Pekka Abrahamsson. 2024. AI-powered Code Review with LLMs: Early Results. *arXiv preprint arXiv:2404.18496* (2024).
- [19] Chanchal Kumar Roy and James R Cordy. 2007. A survey on software clone detection research. *Queen's School of computing TR* 541, 115 (2007), 64–68.
- [20] Chanchal K. Roy and James R. Cordy. 2008. NiCad: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *Proceedings of the 16th International Conference on Program Comprehension*. 172–181. doi:10.1109/ICPC.2008.41
- [21] Chanchal K Roy, James R Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming* 74, 7 (2009), 470–495.
- [22] Hitesh Sajani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-Code. In *Proceedings of the 38th International Conference on Software Engineering*. 1157–1168. doi:10.1145/2884781.2884877
- [23] Chutweeraya Sriwilailak, Yoshiki Higo, Pongpop Lapvikai, Chaoyong Ragkhitwet-sagul, and Morakot Choetkiertikul. [n. d.]. Autorepairability of ChatGPT and Gemini: A Comparative Study. ([n. d.]).
- [24] Jeffrey Svajlenko and Chanchal K Roy. 2015. Evaluating clone detection tools with bigclonebench. In *2015 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 131–140.
- [25] Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Haotian Hui, Weichuan Liu, Zhiyuan Liu, et al. 2024. Debugbench: Evaluating debugging capability of large language models. *arXiv preprint arXiv:2401.04621* (2024).
- [26] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* (2024).
- [27] Min Wang, Pengcheng Wang, and Yun Xu. 2017. CCSharp: An efficient three-phase code clone detector using modified PDGs. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 100–109.
- [28] Yueming Wu, Deqing Zou, Shihan Dou, Siru Yang, Wei Yang, Feng Cheng, Hong Liang, and Hai Jin. 2020. SCDetector: Software functional clone detection based on semantic tokens analysis. In *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*. 821–833.
- [29] Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational automated program repair. *arXiv preprint arXiv:2301.13246* (2023).
- [30] Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. 2019. Neural detection of semantic code clones via tree-based convolution. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 70–80.
- [31] Li Yujian and Liu Bo. 2007. A normalized Levenshtein distance metric. *IEEE transactions on pattern analysis and machine intelligence* 29, 6 (2007), 1091–1095.
- [32] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 783–794. doi:10.1109/ICSE.2019.00086
- [33] Gang Zhao and Jeff Huang. 2018. DeepSim: deep learning code functional similarity. In *2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 141–151.
- [34] Wenqing Zhu, Norihiro Yoshida, Toshihiro Kamiya, Eunjong Choi, and Hiroaki Takada. 2022. MSCCD: grammar pluggable clone detection based on ANTLR parser generation. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. 460–470.
- [35] Wenqing Zhu, Norihiro Yoshida, Toshihiro Kamiya, Eunjong Choi, and Hiroaki Takada. 2025. Development and benchmarking of multilingual code clone detector. *Journal of Systems and Software* 219 (2025), 112215. doi:10.1016/j.jss.2024.112215