Master Thesis

Title

Impact of Identifier Names and Syntactic Similarity on LLM-Based Type-4 Clone Detection

Supervisor
Professor Yoshiki Higo

Author
Ryutaro Inoue

February 2nd, 2026

Software Engineering Laboratory, Department of Computer Science
Graduate School of Information Science and Technology, Osaka University

Fiscal Year 2025 Master Thesis

Impact of Identifier Names and Syntactic Similarity on LLM-Based Type-4 Clone Detection

Ryutaro Inoue

Abstract

Code clones degrade software maintainability, making their efficient detection crucial. However, detection is challenging, especially for Type-4 semantic clones (code fragments that are syntactically different but functionally equivalent). In the era of Generative AI (GenAI), Large Language Model (LLM)-based clone detection has become increasingly important as developers increasingly integrate GenAI into their workflows. Given that many leading GenAI technologies are closed-source systems, it is essential to understand the factors influencing their behavior to improve such detection methods. In this study, we investigate the role that identifiers (names assigned to programming elements such as variables, functions, and classes) play in detecting Type-4 clones. We conducted controlled experiments to evaluate six LLMs (GPT-4.1, Phi-4, Gemini-2.5-pro, Gemma-3-27b-it, CodeLlama-7B-Instruct, and CodeGemma-7b-it) using the FEMPDataset. This dataset contains 2,194 human-written method pairs, including 1,342 functionally equivalent clone pairs and 852 non-clone pairs. We systematically mask different identifier types (specifically method, argument, and variable names) to assess their impact on detection performance. Our findings reveal that identifier names significantly influence LLM-based clone detection decisions. Identifier names are not essential and can even have a negative impact on detection accuracy. While variable names maintain or slightly improve clone detection performance, method names often adversely affect it. Furthermore, the lexical and semantic similarity of method names significantly influences LLM clone detection decisions. In terms of syntactic similarity, method pairs with lower similarity tend to yield reduced clone detection accuracy. Moreover, low syntactic similarity increases the models' reliance on identifier names, whereas high syntactic similarity reduces this dependency. Our insights contribute to a deeper understanding of LLM-based clone detection and provide practical guidance for designing prompting strategies that account for identifier types

and syntactic similarity, ultimately enhancing the detection of Type-4 clones in software systems.

Keywords

Code Clones
Identifiers
Large Language Models
Type-4 Clones
Syntactic Similarity
FEMPDataset

Contents

1 Introduction

Code clones often require consistent changes across all instances; inconsistent changes are known to cause defects [26]. Furthermore, code clones can lead to unpatched vulnerabilities and can accumulate as technical debt, ultimately increasing long-term maintenance costs. Consequently, code clones pose significant challenges for modifying source code and can significantly impair system maintainability. For these reasons, it is crucial for developers to efficiently detect code clones and refactor them when necessary. Although numerous code clone detection tools have been developed over the years [32], overall detection accuracy remains limited, particularly for clones with low syntactic similarity. Static-analysis-based tools, such as those using lexical analysis and software metrics, have demonstrated high accuracy for detecting lexically similar code clones [20, 33, 37] and machine learning-based tools have shown higher accuracy than static-analysis approaches for detecting code clones with low syntactic similarity [46, 41, 36]. Code clones are typically classified into four types according to their degree of similarity [34]: Type-1 (clones that are identical except for formatting differences, such as whitespace and comments), Type-2 (clones that are identical except for differences in identifiers, literals, and data types), Type-3 (clones that are identical except for differences in identifiers, literals, data types, and for statements added, modified, or removed), and Type-4 (clones that exhibit little or no syntactic similarity but remain functionally equivalent).

Detecting Type-4 clones is challenging because significant variations in algorithms, data structures, and API usage make it difficult to determine program equivalence using structural representations such as ASTs and PDGs. Moreover, differing identifier names often render token-based static analysis unreliable for detection. Due to these factors, Type-4 clone detection remains a particularly challenging problem. To address this challenge, many tools have combined multiple information sources, such as ASTs, control flow structures, and machine learning models.

LLM-based clone detection has attracted significant attention [48, 15]. Trained on massive text corpora, LLMs have been successfully applied to various software engineering tasks, including code generation and clone detection [9, 29, 30, 12]. LLMs have demonstrated high accuracy in detecting code clones with low syntactic similarity, which has been difficult for previous detection tools to identify [47]. However, most LLMs are primarily developed for natural language processing and optimized to produce plausible responses that align with user expectations. Consequently, their internal decision-making processes

(a) Before Masking

```
int increment(int value) {
  int result = value + 1;
  return result;
}
```

(b) After Masking

```
int METHOD(int METHODPARAMETER0) {
  int VARIABLE0 = METHODPARAMETER0 + 1;
  return VARIABLE0;
}
```

Figure 1: Concept of masking identifiers. "MA mask" masks method and arguments. "MV mask" masks method and variables. "AV mask" masks method arguments and variables.

are not transparent, making it unclear exactly which features LLMs prioritize when identifying code clones [48].

To address the challenges of Type-4 clone detection, we quantitatively analyze how identifier names and syntactic similarity affect LLM-based code clone detection. Given that input token sequences fundamentally influence LLM behavior, this study investigates the impact of factors such as identifier names (e.g., method and variable names) and syntactic similarity on detection performance [43]. Our goal is to derive insights that help mitigate misclassifications arising from these factors. Specifically, as LLMs are primarily trained on natural language, they may rely excessively on identifier names, potentially misclassifying non-clone pairs as clones based on lexical or semantic similarities between these names. While existing studies have evaluated the impact of identifier names in smaller models such as GraphCodeBERT, there is still a lack of detailed analysis concerning which specific identifier types affect detection in larger LLMs and how syntactic similarity interacts with identifier information.

As shown in Fig. 1, we systematically mask identifiers to understand each identifier's impact on LLM-based code clone detection accuracy, specifically considering identifier names and syntactic similarity. We evaluate six LLMs: GPT-4.1, Phi-4, Gemini-2.5-pro, Gemma-3-27b-it, CodeLlama-7B-Instruct, and CodeGemma-7b-it. We categorize identifiers into three types—method, argument, and variable names—and mask each type to investigate its impact on clone detection accuracy. Our investigation is structured around

two main research questions and five sub-questions, yielding the following insights:

RQ1: How do identifiers impact LLM-based clone detection?

RQ1.a: *How does masking identifier names affect the accuracy of LLM-based code clone detection?*

RQ1.b: *Among method names, parameter names, and variable names, which type of identifier has the greatest impact on LLM decisions?*

RQ1.c: *How do the lexical and semantic similarities of method names affect LLM decisions in code clone detection?*

Results: Identifier names are not essential for Type-4 clone detection and can even negatively affect performance. Among identifier types, method names particularly hinder performance and lead to misclassifications, whereas variable names often maintain or improve detection accuracy. Furthermore, both lexical and semantic similarity of method names significantly influence LLM decisions.

RQ2: How does syntactic similarity impact LLM-based clone detection?

RQ2.a: *How does syntactic similarity affect LLM decisions in code clone detection?*

RQ2.b: *How do syntactic similarity and identifier names jointly affect LLM-based code clone detection?*

Results: Method pairs with lower syntactic similarity tend to yield reduced clone detection accuracy. Furthermore, low syntactic similarity increases the models' reliance on identifier names, whereas high syntactic similarity reduces this dependency.

Our key contributions are as follows:

- A fine-grained empirical analysis of how identifiers and syntactic similarity influence Type-4 clone detection.
- Understanding of the information that LLMs rely on when making clone detection decisions.

The remainder of this paper is structured as follows: Section 2 describes our dataset preparation; Section 3 presents the experimental setup; Section 4 explains the evaluation metrics; Section 5 presents the results and answers the research questions; Section 6 discusses our findings and threats to validity; and Section 7 concludes the paper.

2 Dataset Preparation

Our experiments utilize the FEMPDataset, a benchmark specifically designed for Type-4 clone detection. FEMPDataset consists of Java method pairs, where each entry comprises two methods and a label indicating their functional equivalence. In this context, clone pairs represent Type-4 clones that implement the same functionality through divergent algorithmic structures, whereas non-clone pairs consist of methods that are not functionally equivalent. The dataset contains 1,342 clone pairs and 852 non-clone pairs, for a total of 2,194 method pairs. The functional equivalence of clone pairs was validated through the cross-execution of test cases and manual expert review, ensuring the reliability of the ground-truth labels. Additionally, since the source code in FEMPDataset contains no comments, natural language cues are primarily restricted to identifier names.

In our experiments, we apply minimal formatting normalization to the source code to mitigate the influence of superficial formatting differences. Specifically, we standardize indentation, convert tab characters to spaces, and normalize line endings as well as redundant blank lines. These preprocessing steps do not modify token sequences, statement order, or identifier names; therefore, they do not affect program behavior or syntactic structure. The systematic masking of these identifiers is detailed in Section 3.1.

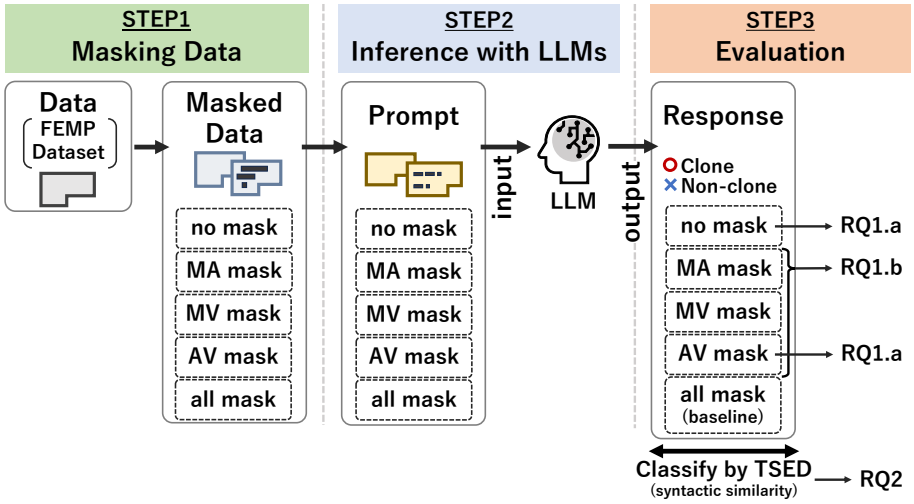


Figure 2: Experiment steps

3 Experiment Setup

Fig. 2 shows the design of our controlled empirical evaluation, which compares the impact of identifier masking on LLM-based code clone detection. Using a fixed set of method pairs from FEMPDataset, we systematically apply various masking patterns and evaluate each LLM under identical prompt and inference conditions. This design enables direct comparison across masking patterns and models while holding non-identifier factors constant. The overall procedure consists of three stages:

- *STEP1 (Masking Data)*: apply identifier masking patterns to each method pair in FEMPDataset and obtain masked code pairs for each pattern;
- *STEP2 (Inference with LLMs)*: construct prompts from the original and masked code pairs and obtain clone/non-clone predictions from each LLM;
- *STEP3 (Evaluation)*: compare LLM outputs with ground-truth labels and compute performance metrics for each masking pattern.

The goal of this experiment is to answer RQ1 and RQ2 by standardizing inputs and inference conditions, allowing for a quantitative comparison of performance differences induced by identifier masking. Specifically, we isolate the contribution of identifier information by limiting input variation to the presence, absence, and specific patterns of identifier masking. By keeping the prompt template and LLMs' decoding settings identical across all

conditions, we can directly attribute performance differences to the masking patterns and models.

3.1 Step 1: Identifier Masking Strategies

We generate variants of the FEMPDataset in which identifier names are replaced with placeholder tokens. FEMPDataset is a dataset that collects functionally equivalent method pairs with different structural implementations. FEMPDataset contains 1,342 clone pairs and 852 non-clone pairs. We apply masking to three categories of identifiers: Method names: replaced with "METHOD.", Method argument names: replaced with "METHOD-PARAMETERX.", Variable names: replaced with "VARIABLEX" (where X represents a sequential integer starting from 0). Masking applies only to identifiers defined within each method. Identifiers from external libraries are not masked to preserve the semantic context of API calls, ensuring the code remains representative of real-world usage during evaluation.

We generate a dataset by masking the specified identifiers. The masking patterns defined in Table 1 are as follows: "no mask" represents the original data; "MA mask" targets method and argument names; "MV mask" targets method and local variable names; "AV mask" targets argument and local variable names; and "all mask" masks all identifier types.

3.2 Step 2: Prompt Design and LLM Inference

As shown in Fig. 3, we use a single fixed prompt template across all experiments to ensure consistency and avoid bias from prompt variations. Each prompt exchange with the LLM consists of a system message and a user message. In the system role, we instruct the model

Table 1: Identifier Name Masking Patterns

	Method	Method Argument	Variable
no mask	×	×	×
MA mask	◯	◯	×
MV mask	◯	×	◯
AV mask	×	◯	◯
all mask	◯	◯	◯

◯: Masking applied, ×: No masking applied

(a) **System message**

```
You must respond with only 'Yes' or 'No'.
```

(b) **User message**

```
I will now give you the two snippets, and you are to answer the questions
    based on the content of the two snippets.

Snippet 1:
int METHOD(int METHODPARAMETER0) {
  int VARIABLE0 = METHODPARAMETER0 + 1;
  return VARIABLE0;
}

Snippet 2:
int METHOD(int METHODPARAMETER0) {
  return METHODPARAMETER0 + 1;
}

Please analyze the two code snippets and determine if they are code clones.
    Respond with 'yes' if the code snippets are clones or 'no' if not.
```

Figure 3: Example of prompt structure for LLM-based clone detection with masked identifiers

Note: The system message constrains the output format, while the user message presents the masked method pair and the question.

to respond only "Yes" or "No." In the user role, we provide two method snippets and ask the model to determine if they are code clones. All LLMs are evaluated under identical inference conditions. We set the temperature to 0.0 to ensure strict reproducibility by generating deterministic outputs, thereby eliminating variability associated with stochastic sampling. For each method pair and masking pattern, we perform a single inference pass, treating the resulting "Yes" or "No" answer directly as the final classification outcome. We utilize the OpenAI API for GPT-4.1 and the Google Gemini API for Gemini-2.5-pro, while all other models are sourced from Hugging Face and executed locally using four NVIDIA RTX A6000 GPUs.

3.3 Step 3: Analyzing LLM Decisions

To analyze LLM decisions and address the RQs, we quantitatively measure the accuracy of classification outcomes, the characteristics of identifier names within the two input methods, and their syntactic similarity. We define the metrics used for these analyses and explain how they are computed. To evaluate classification accuracy, we categorize the predictions for each method pair into four standard groups before computing performance metrics. Each method pair is classified into one of four categories based on the ground truth and the LLM's prediction:

TP (True Positive): The number of actual clone pairs classified as clones.

FP (False Positive): The number of non-clone pairs classified as clones.

FN (False Negative): The number of actual clone pairs classified as non-clones.

TN (True Negative): The number of non-clone pairs classified as non-clones.

Performance Metrics: Based on these classifications, we compute three performance metrics:

Recall: The proportion of actual clone pairs that are correctly identified as code clones.

$$Recall = \frac{TP}{TP + FN}$$

Precision: The proportion of predicted clone pairs that are actual clone pairs.

$$Precision = \frac{TP}{TP + FP}$$

F1-score: The harmonic mean of recall and precision.

$$F1\text{-}score = \frac{2 \times Recall \times Precision}{Recall + Precision}$$

4 Evaluation Metrics

These metrics are employed across our research questions: in RQ1.a to compare performance between masked and preserved identifiers; in RQ1.b to analyze the impact of individual identifier types (method, argument, and variable names); and in RQ2 to evaluate performance variations across different levels of syntactic similarity.

4.1 Code Representation Similarity (RQ1)

To evaluate method name similarity, we compute both lexical and semantic similarity scores between method names.

Lexical Similarity (Jaccard Index): The Jaccard index measures lexical similarity by comparing word sets extracted from method names; specifically, each name is tokenized into a set of words based on CamelCase or snake_case conventions. The Jaccard index between two sets A and B is defined as:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

The resulting value ranges from 0 (no overlap) to 1 (identical sets), where values closer to 1 indicate greater lexical similarity. For example, comparing `commonPrefix` and `getCommonPrefix` yields the common terms {`common`, `Prefix`}, resulting in a high Jaccard index ($2/3 \approx 0.67$).

Semantic Similarity (Cosine Similarity): Cosine similarity measures semantic similarity by comparing vector representations of method names. Each method name is encoded into a semantic vector using the `codet5p-110m-embedding`, as it is pretrained on large-scale code corpora to capture semantic information from identifiers. The cosine similarity between two vectors $\mathbf{A}$ and $\mathbf{B}$ is defined as:

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}$$

The similarity ranges from -1 (opposite direction) to 1 (same direction), where values closer to 1 indicate greater semantic similarity. For example, comparing `areEqualDates` and `isSameDay` yields a Jaccard index of 0.00 but a cosine similarity of 0.73, indicating high semantic similarity despite having no lexical overlap. Unlike the Jaccard index, cosine similarity remains high as long as the embeddings represent similar meanings.

These metrics are used in RQ1.c to analyze how method name similarity affects LLM decisions, specifically examining whether high similarity causes non-clone pairs to be mis-

classified as clones, and whether low similarity causes clone pairs to be misidentified as non-clones.

Identifier Uniqueness: To evaluate method name uniqueness, we use word frequency as a metric to distinguish common words (high frequency) from distinctive words (low frequency). For each method name, we tokenize it into words and compute each word's frequency across the entire dataset. The average frequency of these words serves as a measure of rarity, with lower averages indicating more distinctive names. This metric is used in RQ1.c to investigate the sensitivity of LLM decisions to identifier distinctiveness, specifically analyzing whether method names containing low-frequency words lead to misclassifications.

4.2 Syntactic Similarity (RQ2)

To evaluate the syntactic similarity of method pairs, we analyze structural similarity based on Abstract Syntax Trees (ASTs). We use Tree-based Similarity Edit Distance (TSED) [38] as a metric. TSED leverages the APTED algorithm to calculate the tree edit distance between AST structures. The TSED score is calculated as follows:

$$\text{TSED} = \max\left\{1 - \frac{\delta}{\text{MaxNodes}(G_1, G_2)}, 0\right\}$$

where δ represents the tree edit distance between two ASTs computed using the APTED algorithm and $\text{MaxNodes}(G_1, G_2)$ represents the maximum number of nodes among the two ASTs. TSED normalizes this score to express similarity on a scale from 0 to 1, where values closer to 1 indicate higher similarity. This metric is used in RQ2.a to evaluate how syntactic similarity levels affect LLM detection accuracy, and in RQ2.b to analyze the interaction between syntactic similarity and identifier names.

Preliminary Analysis: Dataset Distribution by TSED Prior to our main analysis, we present the distribution of method pairs across various TSED ranges within the FEMPDataset. As shown in Table 2, the method pairs are divided into four TSED intervals: [0.00, 0.25), [0.25, 0.50), [0.50, 0.75), and [0.75, 1.00]. In RQ2.a, we calculate Recall, Precision, and F1-score for each of the four intervals, while in RQ2.b, we examine two aggregated ranges. This allows us to evaluate the relationship between syntactic similarity and detection performance. The results indicate that clone pairs tend to have higher syntactic similarity, whereas non-clone pairs tend to have lower syntactic similarity. This

categorization enables the analysis of identifier impact across different levels of structural resemblance in RQ2.

Table 2: Counts of method pairs by TSED range in FEMPDataset

	TSED range			
	[0.00, 0.25)	[0.25, 0.50)	[0.50, 0.75)	[0.75, 1.00]
Clone	48	268	389	637
Non-clone	100	308	239	205
Overall	148	576	628	842

5 Empirical Results

We now present our results.

Table 3: Performance Evaluation Results: All Mask vs. No Mask

(a) F1-score

Model	all mask	no mask
GPT-4.1	0.88	0.86(−0.02)
Phi-4	0.84	0.83(−0.01)
Gemini-2.5-pro	0.84	0.84(+0.00)
Gemma-3	0.75	0.81(+0.06)
CodeLlama	0.74	0.70(−0.04)
CodeGemma	0.24	0.15(−0.09)

(b) Recall

Model	all mask	no mask
GPT-4.1	0.88	0.88(+0.00)
Phi-4	0.84	0.82(−0.02)
Gemini-2.5-pro	0.96	0.96(+0.00)
Gemma-3	0.66	0.79(+0.13)
CodeLlama	0.77	0.66(−0.11)
CodeGemma	0.14	0.08(−0.06)

(c) Precision

Model	all mask	no mask
GPT-4.1	0.87	0.84(−0.03)
Phi-4	0.85	0.85(+0.00)
Gemini-2.5-pro	0.76	0.74(−0.02)
Gemma-3	0.88	0.82(−0.06)
CodeLlama	0.71	0.75(+0.04)
CodeGemma	0.92	0.94(+0.02)

Note: The column **all mask** is the baseline. For no mask, each value shows the score, and the value in parentheses indicates the difference from the all-mask score for the same model.

5.1 RQ1: How do identifiers impact LLM-based clone detection?

We now present the approach and results for RQ1 broken into three sub-research questions.

RQ1.a: How does masking identifier names affect the accuracy of LLM-based code clone detection? To evaluate the impact of identifier names on clone detection performance, we compare Recall, Precision, and F1-score under two distinct conditions: *no mask* (preserving all identifiers) and *all mask* (masking all identifiers). Specifically, we calculate the performance difference ($\Delta\text{score} = \text{score}_{\text{no mask}} - \text{score}_{\text{all mask}}$) to quantify the extent to which identifier names influence performance. Using *all mask* as the baseline, an improvement under *no mask* suggests a positive influence from identifier names, whereas a performance decrease indicates a negative influence.

Results As shown in Table 3, identifier names are not essential for Type-4 clone detection and can even negatively affect performance.

For Recall, only Gemma-3 exhibited a noticeable improvement ($\Delta\text{Recall} = +0.13$) when identifier names were preserved. In contrast, all other models showed either no change or a decrease: GPT-4.1 (+0.00), Gemini-2.5-pro (+0.00), Phi-4 (−0.02), CodeLlama (−0.11), and CodeGemma (−0.06). These results indicate that, with the exception of Gemma-3, preserving identifier names did not enhance the models' ability to correctly identify true clone pairs and, in some cases, even reduced it.

For Precision, performance trends varied across models when identifier names were preserved, with no consistent pattern emerging. Three models showed decreases: GPT-4.1 ($\Delta\text{Precision} = -0.03$), Gemini-2.5-pro (−0.02), and Gemma-3 (−0.06); in contrast, three models showed either no change or increases: Phi-4 (+0.00), CodeGemma (+0.02), and CodeLlama (+0.04). These results indicate that the influence of identifier names on the ability to reduce false positives is inconsistent across models.

For F1-score, only Gemma-3 ($\Delta\text{F1-score} = +0.06$) showed improvement when identifier names were preserved, and Gemini-2.5-pro (+0.00) showed no change. All other models showed decreases: GPT-4.1 ($\Delta\text{F1-score} = -0.02$), Phi-4 (−0.01), CodeLlama (−0.04), and CodeGemma (−0.09). These results indicate that preserving identifier names fails to enhance overall detection performance for most models and often leads to a slight degradation.

RQ1.b: Among method names, parameter names, and variable names, which type of identifier has the greatest impact on LLM decisions? To evaluate the individual impact of each identifier type, we employed *all mask* (in which all identifiers are removed) as the baseline. We then compared this baseline with three masking patterns designed to isolate specific identifier categories: *MA mask* (variable names only), *MV mask* (method argument names only), and *AV mask* (method names only). For Recall, Precision, and F1-score, we computed performance differences (difference = score(*AV / MV / MA mask*) − score(*all mask*)) in order to assess the impact of each identifier type on clone detection performance.

Results As shown in Table 4, variable names generally preserve or slightly enhance clone detection performance, whereas method names often adversely affect it.

For the MA mask (variable names only), Recall showed increases in three models: Phi-4 (ΔRecall = +0.04), Gemma-3 (+0.07), and CodeLlama (+0.04), with most other models showing no change or slight decreases. Precision showed only small changes across all models: GPT-4.1 (ΔPrecision = −0.02), Gemini-2.5-pro (+0.00), and CodeLlama (+0.02). For F1-score, most models showed no change or increases: GPT-4.1 (ΔF1-score = +0.00), Gemini-2.5-pro (+0.00), Phi-4 (+0.02), Gemma-3 (+0.04), and CodeLlama (+0.01), with only CodeGemma (−0.01) showing a slight decrease. These results indicate that variable names improve overall classification performance by increasing recall while maintaining precision, suggesting they help mitigate false negatives without introducing additional false positives.

For the MV mask (method argument names only), Recall showed mixed results with no consistent trend: increases in Gemma-3 (ΔRecall = +0.10) and GPT-4.1 (+0.02), but decreases in CodeLlama (−0.08) and CodeGemma (−0.03). Precision showed small changes across all models: GPT-4.1 (ΔPrecision = −0.02), Gemma-3 (−0.03), and CodeLlama (+0.02). For F1-score, results varied across models with increases in Gemma-3 (ΔF1-score = +0.05) and Gemini-2.5-pro (+0.01), but decreases in GPT-4.1 (−0.01), CodeLlama (−0.03), and CodeGemma (−0.05). These results indicate that method argument names show inconsistent effects across models, with no unified trend in recall, precision, or overall detection performance.

For the AV mask (method names only), Recall showed decreases in Phi-4 (ΔRecall = −0.04), CodeLlama (−0.10), and CodeGemma (−0.02), no change in GPT-4.1 (+0.00) and Gemini-2.5-pro (+0.00), and an increase in Gemma-3 (+0.07). Precision showed decreases

in multiple models: GPT-4.1 (ΔPrecision $= -0.04$), Phi-4 (-0.02), and Gemma-3 (-0.06). For F1-score, four of six models showed decreases: GPT-4.1 (ΔF1-score $= -0.02$), Phi-

Table 4: Performance Evaluation Results: All Mask vs. MA Mask, MV Mask, AV Mask

(a) F1-score

Model	all mask	MA mask	MV mask	AV mask
GPT-4.1	0.88	0.88(+0.00)	0.87(−0.01)	0.86(−0.02)
Phi-4	0.84	0.86(+0.02)	0.85(+0.01)	0.81(−0.03)
Gemini-2.5-pro	0.84	0.84(+0.00)	0.85(+0.01)	0.84(+0.00)
Gemma-3	0.75	0.79(+0.04)	0.80(+0.05)	0.77(+0.02)
CodeLlama	0.74	0.75(+0.01)	0.71(−0.03)	0.70(−0.04)
CodeGemma	0.24	0.23(−0.01)	0.19(−0.05)	0.21(−0.03)

(b) Recall

Model	all mask	MA mask	MV mask	AV mask
GPT-4.1	0.88	0.88(+0.00)	0.90(+0.02)	0.88(+0.00)
Phi-4	0.84	0.88(+0.04)	0.85(+0.01)	0.80(−0.04)
Gemini-2.5-pro	0.96	0.96(+0.00)	0.96(+0.00)	0.96(+0.00)
Gemma-3	0.66	0.73(+0.07)	0.76(+0.10)	0.73(+0.07)
CodeLlama	0.77	0.81(+0.04)	0.69(−0.08)	0.67(−0.10)
CodeGemma	0.14	0.13(−0.01)	0.11(−0.03)	0.12(−0.02)

(c) Precision

Model	all mask	MA mask	MV mask	AV mask
GPT-4.1	0.87	0.87(+0.00)	0.85(−0.02)	0.83(−0.04)
Phi-4	0.85	0.84(−0.01)	0.84(−0.01)	0.83(−0.02)
Gemini-2.5-pro	0.76	0.75(−0.01)	0.76(+0.00)	0.75(−0.01)
Gemma-3	0.88	0.86(−0.02)	0.85(−0.03)	0.82(−0.06)
CodeLlama	0.71	0.71(+0.00)	0.73(+0.02)	0.73(+0.02)
CodeGemma	0.92	0.94(+0.02)	0.90(−0.02)	0.94(+0.02)

Note: The column **all mask** is the baseline. For MA/MV/AV mask, each value shows the score, and the value in parentheses indicates the difference from the all-mask score for the same model.

4 (−0.03), CodeLlama (−0.04), and CodeGemma (−0.03), with Gemini-2.5-pro showing no change and only Gemma-3 showing improvement (+0.02). These results indicate that method names consistently degrade detection performance across both recall and precision, representing the most negative impact among all identifier types.

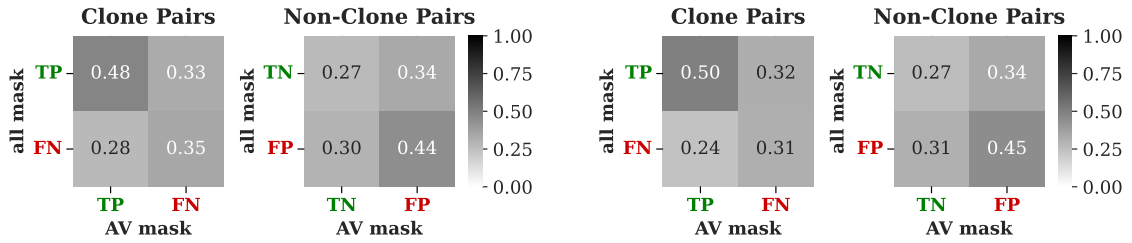
RQ1.c: How do the lexical and semantic similarities of method names affect LLM decisions in code clone detection? To investigate how method names influence clone detection, we analyze GPT-4.1 and Phi-4 by comparing their predictions under *all mask* (baseline with all identifiers removed) and *AV mask* (with only method names preserved).

First, we classify prediction changes from all mask to AV mask into four patterns: maintained correct (predictions remain correct), changed to incorrect (predictions become wrong), maintained incorrect (predictions remain wrong), and changed to correct (predictions are corrected by method names). As shown in Table 5, we record the number of method pairs for each pattern. In particular, the number of pairs that were incorrectly classified as clones is greater than the number of pairs that were correctly classified as non-clones, indicating that the LLM is misclassifying non-clone pairs as clone pairs, leading to performance degradation.

Second, for each of the eight patterns (four change types for clone pairs and four for non-clone pairs), we compute three similarity metrics—Jaccard index (lexical similarity), cosine

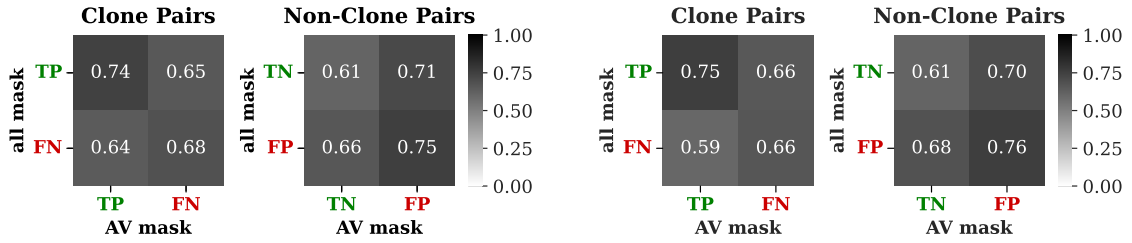
Table 5: Number of method pairs for each judgment change from all mask to AV mask (GPT-4.1 and Phi-4).

Pair Type	**Change Type**	**Judgment Change**	**Number of pairs**	
		(all mask→AV mask)	GPT-4.1	Phi-4
Clone	Maintained correct	TP→TP	1,142	1,047
	Changed to incorrect	TP→FN	41	84
	Maintained incorrect	FN→FN	117	185
	Changed to correct	FN→TP	42	26
Non-clone	Maintained correct	TN→TN	583	583
	Changed to incorrect	TN→FP	94	62
	Maintained incorrect	FP→FP	141	165
	Changed to correct	FP→TN	34	42



(a) GPT-4.1 (b) Phi-4

Figure 4: Heatmaps showing the Averaged Jaccard Index for judgment changes from all mask to AV mask.



(a) GPT-4.1 (b) Phi-4

Figure 5: Heatmaps showing the Averaged Cosine Similarity for judgment changes from all mask to AV mask.

similarity (semantic similarity), and word frequency—to characterize method names. We then analyze how these metrics relate to prediction changes.

Results We analyze how method name characteristics influence prediction changes when method names are added to the baseline (all mask → AV mask) from two perspectives: method name similarity and word frequency.

We present our analysis from two complementary perspectives: method-name similarity (as measured by cosine similarity and Jaccard index) and method-name uniqueness (as captured by the word-frequency metric).

For method name similarity, as shown in Figs. 4 and 5, low similarity in clone pairs causes correct predictions to become incorrect, while high similarity in non-clone pairs causes misclassification as clones.

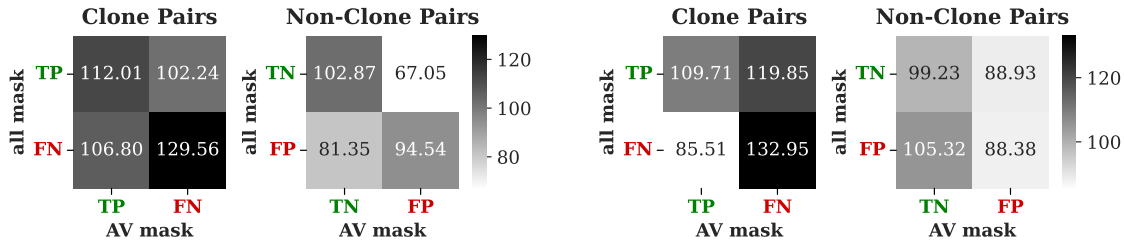
First, for clone pairs, we examine whether dissimilar method names lead to incorrect predictions by comparing pairs whose predictions changed from correct to incorrect (TP→FN) with pairs that maintained correct predictions (TP→TP). In terms of Jaccard index, the

average lexical similarity for pairs that became misclassified (TP→FN: GPT-4.1 = 0.33, Phi-4 = 0.32) is substantially lower than that for pairs that remained correctly classified (TP→TP: GPT-4.1 = 0.48, Phi-4 = 0.50). Cosine similarity shows the same pattern, with pairs that became misclassified showing lower semantic similarity than those that remained correctly classified. This indicates that when clone pairs have dissimilar method names, correct predictions tend to be overturned, leading to incorrect predictions. In contrast, we examine whether similar method names help correct misclassifications by comparing pairs whose predictions changed from incorrect to correct (FN→TP) with pairs that remained misclassified (FN→FN). However, no clear trend was observed in terms of either Jaccard index or cosine similarity between these two groups.

Second, for non-clone pairs, we examine whether similar method names lead to incorrect predictions by comparing pairs whose predictions changed from correct to incorrect (TN→FP) with pairs that maintained correct predictions (TN→TN). In terms of Jaccard index, the average lexical similarity for pairs that became misclassified (TN→FP: GPT-4.1 = 0.34, Phi-4 = 0.34) is higher than that for pairs that remained correctly classified (TN→TN: GPT-4.1 = 0.27, Phi-4 = 0.27). Cosine similarity shows the same pattern, with pairs that became misclassified showing higher semantic similarity than those that remained correctly classified. This indicates that when non-clone pairs have similar method names, correct predictions tend to be overturned, leading to incorrect predictions. In contrast, we examine whether dissimilar method names help correct misclassifications by comparing pairs whose predictions changed from incorrect to correct (FP→TN) with pairs that remained misclassified (FP→FP). In terms of Jaccard index, pairs that became correctly classified (FP→TN: GPT-4.1 = 0.30, Phi-4 = 0.31) show lower lexical similarity than pairs that remained misclassified (FP→FP: GPT-4.1 = 0.44, Phi-4 = 0.45). This indicates that dissimilar method names help correct misclassifications for non-clone pairs.

For method name uniqueness, as shown in Fig. 6, while some trends are observed regarding distinctive (low-frequency) words, no consistent pattern emerges indicating that uniqueness strongly influences clone detection.

First, for clone pairs, we examine whether distinctive (low-frequency) method names lead to incorrect predictions by comparing pairs whose predictions changed from correct to incorrect (TP→FN) with pairs that maintained correct predictions (TP→TP). However, no consistent pattern was observed in the average word frequency between these two groups. In contrast, we examine whether distinctive (low-frequency) method names help correct misclassifications by comparing pairs whose predictions changed from incorrect to



(a) GPT-4.1 (b) Phi-4

Figure 6: Heatmaps showing the Averaged Word Frequency for judgment changes from all mask to AV mask.

correct (FN→TP) with pairs that remained misclassified (FN→FN). The average word frequency for pairs that became correctly classified (FN→TP: GPT-4.1 = 106.80, Phi-4 = 100.79) is lower (more distinctive) than that for pairs that remained misclassified (FN→FN: GPT-4.1 = 129.56, Phi-4 = 135.71). This suggests that distinctive method names may help correct misclassifications for clone pairs.

Second, for non-clone pairs, we examine whether distinctive (low-frequency) method names lead to incorrect predictions by comparing pairs whose predictions changed from correct to incorrect (TN→FP) with pairs that maintained correct predictions (TN→TN). The average word frequency for pairs that became misclassified (TN→FP: GPT-4.1 = 67.05, Phi-4 = 88.93) is lower (more distinctive) than that for pairs that remained correctly classified (TN→TN: GPT-4.1 = 102.87, Phi-4 = 99.23). This indicates that when non-clone pairs have distinctive method names, correct predictions tend to be overturned, leading to incorrect predictions. In contrast, we examine whether distinctive (low-frequency) method names help correct misclassifications by comparing pairs whose predictions changed from incorrect to correct (FP→TN) with pairs that remained misclassified (FP→FP). However, no consistent pattern was observed in the average word frequency between these two groups.

Based on these results, we can draw the following conclusion for RQ1.

Answer to RQ1

We break our answer into three sub-questions

- (RQ1.a) Identifier names are not essential for Type-4 clone detection and may even have a negative impact.
- (RQ1.b) Variable names maintain or slightly improve clone detection performance. Method names may adversely affect clone detection performance.
- (RQ1.c) The lexical and semantic similarity of method names has a significant influence on clone detection using LLMs. Low method name similarity leads to incorrect predictions in non-clone pairs and LLMs' performance degradation in clone detection.

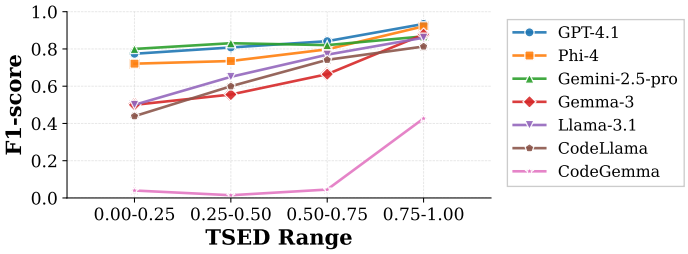
5.2 RQ2: How does syntactic similarity impact LLM-based clone detection?

We now present the approach and results for RQ2 broken into two sub-research questions.

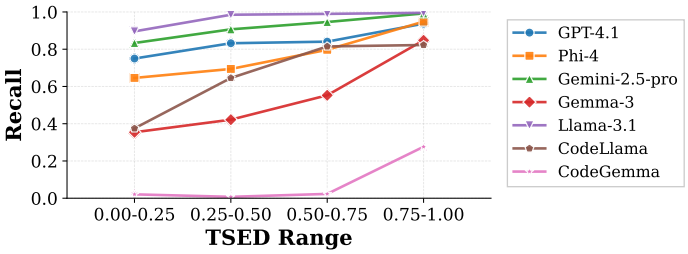
RQ2.a: How does syntactic similarity affect LLM decisions in code clone detection? We evaluate clone detection performance based solely on syntactic structure using the all mask pattern, in which all identifiers are masked. To observe performance differences across varying degrees of syntactic similarity, we divided the 0–1 range of TSED into four equal-width intervals (0.00–0.25, 0.25–0.50, 0.50–0.75, and 0.75–1.00). The number of clone and non-clone pairs in each interval is presented in Table 2. For each TSED interval, we compared the LLMs' predictions with the ground-truth labels and computed precision, recall, and F1-score.

Results As shown in Fig. 7, syntactic similarity significantly affects clone detection performance, with lower TSED values (lower syntactic similarity) consistently leading to reduced detection accuracy across all models.

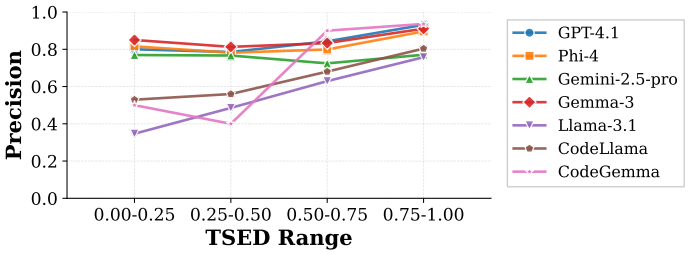
A similar trend was observed for recall and precision: method pairs with lower TSED values generally exhibited lower recall and precision. Specifically, for low-TSED pairs, the drop in recall was more pronounced than the drop in precision. For example, in GPT-4.1, the precision difference between the TSED ranges 0.00–0.25 and 0.75–1.00 is approximately 0.13, while the recall difference is about 0.19. These findings indicate that



(a) F1-score



(b) Recall



(c) Precision

Figure 7: Recall, Precision, and F1-score by model using all mask dataset, analyzed by TSED ranges.

clone pairs with low syntactic similarity tend to be more difficult for LLMs to detect. In particular, many low-similarity clone pairs are missed, indicating that correctly identifying such pairs is difficult for current LLMs.

In particular, Gemma-3, CodeLlama, and CodeGemma struggled to detect low-similarity method pairs compared with high-similarity ones. The F1-scores showed a difference of about 0.4 between the 0.00–0.25 and 0.75–1.00 TSED ranges, with the low-TSED range performing substantially worse. In contrast, Gemini-2.5-pro maintained consistently high detection performance even in the lowest TSED range, indicating greater robustness to variations in syntactic similarity.

RQ2.b: How do syntactic similarity and identifier names jointly affect LLM-based code clone detection? We analyze the combined effect of syntactic similarity (TSED) and identifier masking patterns on LLMs' clone detection decisions. Based on their TSED values, method pairs are classified into two groups: Low-TSED, representing pairs exhibiting large syntactic differences, and High-TSED, representing pairs exhibiting small syntactic differences. For each group, we compute the performance difference between the *all mask* baseline and other masking patterns (differenceΔ = score(masking pattern) − score(*all mask*)) for Recall, Precision, and F1-score to evaluate how identifier types affect detection performance at different levels of syntactic similarity. We compare the performance differences between Low-TSED and High-TSED across all 24 model-masking pattern combinations (6 models × 4 masking patterns).

Results As shown in Fig. 8, identifier names have a substantially greater impact on clone detection performance when syntactic similarity is low compared to when it is high.

For F1-score, 13 out of 24 combinations (54%) show stronger identifier influences in Low-TSED than in High-TSED, while 6 combinations (25%) show the opposite, and 5 combinations (21%) show equal influences. In Low-TSED, MA mask (variable names) tends to improve performance (e.g., Phi-4: ΔF1-score = +0.04, Gemma-3: +0.07), while AV mask (method names) tends to decrease it (e.g., Phi-4: ΔF1-score = −0.05, CodeLlama: −0.05), aligning with RQ1.b.

For Recall, this pattern is even stronger: 15 out of 24 combinations (62%) show stronger influences in Low-TSED, while only 6 combinations (25%) show stronger influences in High-TSED. For Precision, a similar pattern is observed: 14 out of 24 combinations (58%) show stronger influences in Low-TSED, while only 6 combinations (25%) show stronger influences in High-TSED.

These results indicate that when syntactic similarity is low, identifier names have substantial impact on clone detection performance and the choice of identifier type becomes critical, but when syntactic similarity is high, identifier information provides limited additional value regardless of type.

Based on these results, we can draw the following conclusion for RQ2.

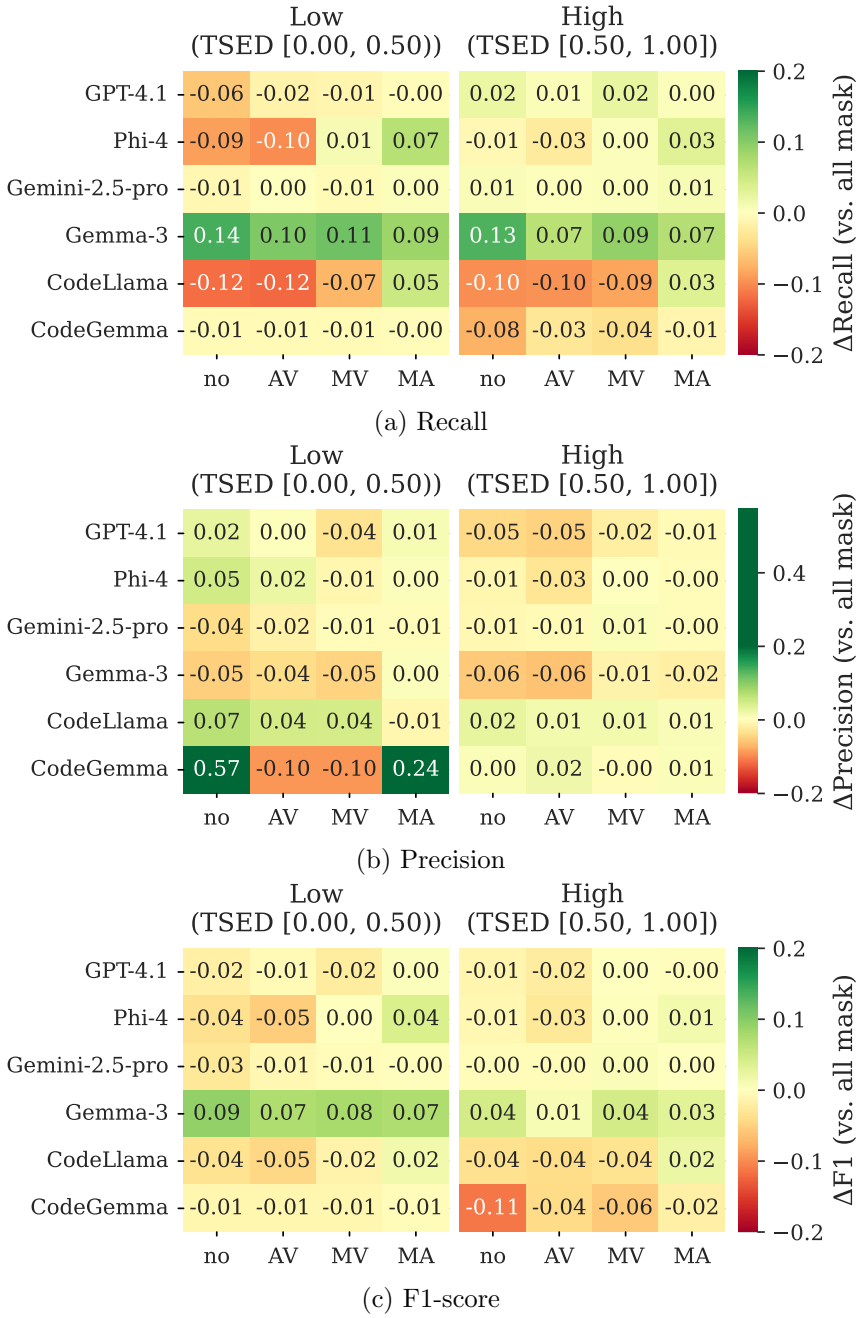


Figure 8: Heatmaps showing the relationship between syntactic similarity (TSED) and identifier masking patterns for F1-score, Precision, and Recall.

Note: The values are the performance differencesΔ of each masking pattern from the all-mask baseline. The masking patterns are abbreviated as no (no mask), AV (AV mask), MV (MV mask), and MA (MA mask).

Answer to RQ2

We summarize the answers to RQs by two sub-research questions:

- (RQ2.a) - Method pairs with lower syntactic similarity (TSED) tend to show reduced clone detection accuracy.
- (RQ2.b) - Low syntactic similarity increases the impact of identifier names on clone detection, while high syntactic similarity reduces it.

6 Discussion

Building on the findings of RQ1 and RQ2, we discuss LLMs' dependence on identifiers and the influence of syntactic similarity on their clone detection performance.

6.1 Dependency of LLMs on Identifiers in Clone Detection

Based on the findings of RQ1 (RQ1.a–RQ1.c), we confirmed that identifiers (method names, argument names, and variable names) affect the clone detection decisions of LLMs. For method names in particular, several models showed indications of performance decline when method names were retained, with modest decreases in F1-score observed under the AV mask pattern. This suggests that, for the models evaluated in this study, method names did not consistently provide beneficial information for clone detection.

In contrast, for variable names, we did not observe any substantial performance degradation when they were retained instead of masked. Several models, including Phi-4, Gemma-3, and CodeLlama, even showed improvements in recall and F1-score, suggesting that variable names may provide useful auxiliary signals for clone classification.

These findings indicate that LLMs rely on identifiers as part of their process of making decisions. However, the manner in which identifiers contribute appears to differ substantially depending on their type. Because method names often contain words describing functionality, non-clone pairs whose method names are lexically or semantically similar may be misclassified as clones. By contrast, variable names tend to provide localized cues about the operational details within a method and are therefore less likely than method names to induce misclassifications.

It is important to note that these interpretations are based solely on observed model behavior, as we do not directly analyze how identifiers influence the internal representations of LLMs. Nevertheless, within the scope of this study, retaining method names rarely improved clone detection, while retaining variable names generally maintained or improved performance. Therefore, from a practical perspective, masking method names may serve as an effective strategy for improving detection accuracy.

6.2 Influence of Syntactic Similarity in Clone Detection

Based on the results of RQ2.a, we confirmed that low syntactic similarity method pairs were consistently difficult to detect across all models. This trend aligns with the well-known challenges of Type-4 clone detection. It suggests that differences in structural

information can also reduce detection accuracy in LLM-based clone detection.

Furthermore, the results of RQ2.b indicate that the influence of identifiers becomes more pronounced when syntactic similarity is low. These observations suggest that when structural cues are weak, LLMs tend to rely more heavily on identifier information. In contrast, when TSED is high, the performance differences between masking and non-masking conditions are small. This implies that, when structural information is strong, LLMs may be able to make decisions without depending excessively on identifiers. Although the magnitude of these effects varies across models, the overall pattern is consistent. For pairs with low syntactic similarity, how identifier information is handled has a substantial impact on detection accuracy.

6.3 Differences in LLM Behavior

A comparison of the six models evaluated in this study reveals that, although there are commonalities in how identifiers influence performance, each model also exhibits distinct behavior. First, in many models (GPT-4.1, Phi-4, CodeLlama, and CodeGemma), retaining method names under the *AV mask* led to a decrease in F1-score, indicating that method names often contribute to misclassifications. However, this trend did not apply uniformly across all models. In the case of Gemma-3, retaining method names improved both Recall and F1-score, making it one of the few models for which method names provided a beneficial signal. This suggests that the impact of identifiers on clone detection depends on the characteristics of a model ' s learned internal representations. In contrast, no model showed a substantial decrease in performance when variable names were retained. For several models, such as Phi-4 and Gemma-3, Recall even improved. These results indicate that variable names may serve as localized cues about a method ' s internal behavior. They can be particularly helpful as supplementary information for method pairs with low syntactic similarity.

Gemini-2.5-pro exhibited the smallest performance variation across masking conditions, showing stable performance without relying heavily on either structural or identifier information. By contrast, CodeLlama and CodeGemma showed large drops in Recall in Low-TSED ranges. This suggests that these models struggle when the structural information in the code is weak. Overall, although there are model-dependent differences in how identifiers affect detection, variable names rarely introduce negative effects and often provide modest benefits. In contrast, method names tend to degrade performance in many models, although exceptions such as Gemma-3-27B-it exist. These findings indicate that

the handling of identifiers should be tailored to the characteristics of each model.

7 Threats to Validity

Limitations This study has several limitations concerning the interpretation of its results and the scope of the analysis. First, the influence of identifiers and structural information on LLM judgments cannot be completely separated. Although we attempted to disentangle these effects through masking schemes and AST-based similarity metrics, LLMs process identifiers and structural features simultaneously as token sequences. Therefore, there are inherent limitations in evaluating them as independent factors. Second, the internal representations and reasoning processes of LLMs are not directly observable. As a result, we cannot causally explain the specific internal mechanisms that give rise to the observed performance differences. Finally, the vocabulary distribution of identifiers depends on the dataset. Thus, the trends observed in this study, such as frequent words providing weaker cues and rare words increasing misclassification risk, may be partially influenced by lexical biases specific to the dataset. These effects may not reflect universal properties of LLMs.

We now discuss threats to the validity of this study.

Internal Validity In this study, we employed a masking approach that replaces identifier names with predefined tokens: method names with "METHOD", method argument names with "METHODPARAMETERX", and variable names with "VARIABLEX". However, our masking strategy is not the only possible one. Prior studies have used alternative techniques, such as replacing identifiers with random strings or shuffling identifier names. Therefore, the results of this study depend on the specific token-replacement strategy used, and different masking methods may lead to different outcomes. Furthermore, our analysis focuses on method name similarity. However, the overall structure and syntactic similarity of the methods could also influence classification outcomes. As shown in RQ2.b, pairs with low syntactic similarity rely strongly on identifier names, making it difficult to completely disentangle the effect of identifiers from other factors.

FEMPDataset is a high-quality benchmark in which functional equivalence is verified through mutual execution of test cases and subsequently reviewed by three experts. However, the dataset mainly consists of relatively simple Java methods for which such tests can be automatically generated. As a result, it may not fully capture the diversity found in real-world code. Future work should evaluate the models on more complex, multi-layered systems to confirm whether the identified trends hold true. Consequently, our conclusions

are inherently tied to the characteristics of this dataset and may not fully reflect how LLMs behave on complex and diverse real-world codebases.

External Validity This study focuses on a specific dataset of Java code pairs and targets Type-4 clones with functional equivalence. Therefore, our findings may not be directly generalizable to other programming languages (C++, Python, JavaScript, etc.), different domains, or datasets of different scales. Our evaluation is limited to six models available as of June 2025: GPT-4.1, Phi-4, Gemini-2.5-pro, Gemma-3-27b-it, CodeLlama-7B-Instruct, and CodeGemma-7b-it. The degree to which LLMs rely on identifier information varies depending on their architectures and training data, and is likely to change further as the technology continues to advance. In particular, earlier domain-specific models such as CodeLlama-7B-Instruct and CodeGemma-7b-it exhibited lower detection performance compared with the more recent general-purpose models. Accordingly, the conclusions drawn in this study are constrained by the capabilities of the specific models available at the time of evaluation. Moreover, the prompts used for inference followed a single fixed format: a message in the system role specifying the answer format and a message in the user role containing the method pair and the classification request. LLMs' performance is known to be highly sensitive to prompt formulation. Alternative prompt structures or instruction styles may yield different outcomes. Moreover, the use of techniques such as chain-of-thought prompting [44] or few-shot prompting [7] may also yield different results.

8 Related Work

In this section, we describe related work of this study.

8.1 Challenges for Code Clone Detectors

A code clone is defined as a code fragment that is identical or similar to another fragment [32]. A pair of code fragments forming a code clone is called a clone pair. It is well known that large-scale software systems contain a substantial number of code clones. Roy et al.[32] investigated numerous industrial projects and reported that approximately 20–30% of the code in these systems consists of clones. Similarly, Juergens et al.[19] found that commercial systems also contain a significant amount of cloned code. There are various reasons why code clones are created. Clones are often introduced through copy-and-paste practices, where developers duplicate existing code and apply local modifications [22, 32, 24]. This is often considered a pragmatic choice because reusing and modifying existing code requires less implementation effort. Clones may also arise when similar functionality is reimplemented independently within a large system [45, 32, 24]. In such cases, developers fail to locate reusable code, for example because of insufficient code search capabilities. As a result, they reimplement the same functionality, which leads to unintentional cloning.

Whether clones are harmful remains a subject of discussion. It has been reported that clones may be introduced intentionally to improve performance or maintainability in certain contexts [22, 32]. However, numerous studies have shown that clones can adversely affect system quality and maintainability. Because clones often require consistent modification, inconsistent updates can lead to defects caused by missed or partial changes [26]. Clones have also been shown to increase the likelihood of leaving vulnerabilities untreated [23]. Several empirical studies on commercial and open-source systems report real cases in which clones contributed to system failures [19]. Even when defects do not appear immediately, clones accumulate as long-term technical debt [8]. As a result, maintenance costs increase because developers must modify multiple locations when changes are required.

Many studies on code clone detection have primarily relied on lexical or syntactic similarity to extract clones. Static analysis approaches based on tokenization and lexical normalization are fast and perform well at detecting Type-1 and Type-2 clones. Representative tools include CCFinder [20], NiCad [33], NIL [28], and SourcererCC [37]. However, because these approaches rely heavily on token-sequence matching, their accuracy de-

grades substantially for Type-4 clones, where control structures or the order of statements differ. To capture code structure more directly, approaches based on abstract syntax trees (ASTs) have been proposed. Representative tools include Deckard [18]. They use subtree structures to capture structural similarities beyond lexical differences. However, detecting code that implements the same functionality using entirely different syntactic structures or control constructs remains challenging.

Subsequently, machine learning–based methods tailored to Type-4 clone detection, such as Oreo [36], ASTNN [46], and GMN [41] have been proposed. These methods treat ASTs and code metrics as features and have demonstrated high accuracy in detecting Type-4 clones. Because Type-4 clones are syntactically different yet functionally equivalent, static-analysis tools using lexical or syntactic similarity often struggle to detect them. Therefore, the detection of Type-4 clones requires the capability to capture semantic equivalence beyond surface-level lexical and syntactic similarities.

8.2 Code Clone Detection with LLMs

Research on models for semantic understanding of source code has advanced rapidly since the advent of the Transformer architecture [40]. Transformer-based models can capture long-range dependencies through self-attention. They can also represent programs as embeddings. Leveraging these properties, numerous large-scale pretrained models specialized for code, or trained jointly on code and natural language, have been proposed. Representative code-specific models include CodeBERT [11], CodeT5 [42], and PLBART [2]. All of these models are built on Transformer architectures and acquire capabilities for code understanding and generation through pretraining on large corpora of source code and natural language. Subsequently, general-purpose large language models (LLMs), with substantially expanded pretraining scales and data diversity, have emerged and been applied to code understanding tasks. LLMs such as GPT-4 [30] and Gemini [12] have been trained on text corpora containing substantial amounts of code and are capable of handling a variety of tasks, including clone detection. In addition, many LLMs that can run locally have been proposed, including Gemma-3 [13], Phi-4 [1], CodeLlama [35], and CodeGemma [10].

Research using LLMs has been rapidly expanding, with active work in programming-related areas such as clone detection [21, 6, 27] and code generation [9, 29]. These models have been shown to achieve high accuracy in detecting Type-4 clones, which is an area where conventional methods have struggled [47]. However, it is not well understood what

types of information LLMs rely on when detecting clones. LLMs are influenced by various factors, including identifier names, syntactic structures, token-level patterns, and patterns present in their training data. However, the extent to which each factor contributes to model predictions remains unclear. In particular, it remains unclear to what extent identifier names and syntactic structures influence model performance in clone detection. This lack of transparency in the model's decision process means that even small differences in the input code may lead to unexpected variations in predictions.

In our previous work [17], we improved the accuracy of LLM-based code clone detection by fine-tuning models on FEMPDataset. We fine-tuned the following models: `gpt-3.5-turbo`, `Llama2-Chat-7B`, and `CodeLlama-7B-Instruct` For fine-tuning, we used the OpenAI API and applied the LoRA [16] and ZeRO [31] techniques. Fine-tuning improved accuracy for all models, with `CodeLlama-7B-Instruct` (a model specialized for code processing) showing the largest gain.

Consistent with the studies introduced in Section **??**, this work shows that LLMs are effective for Type-4 clone detection and that their accuracy improves with additional training. It further demonstrates that the magnitude of improvement varies across models, depending on their underlying characteristics. However, the reasons underlying LLMs ' ability to detect clones, as well as the specific factors on which their judgments rely, remain unclear.

One notable study on LLM-based code clone detection is that of Almatrafi et al. [6]. They applied few-shot instruction tuning to GPT-4 and GPT-3.5-turbo models, achieving higher clone-detection accuracy. They trained on 100 examples from BigCloneBench [39] and evaluated on 2,000 examples from the same benchmark.

Research on identifiers and syntactic similarity This section discusses studies that investigate how LLMs are influenced by identifiers and syntactic similarity. Wang et al. [43] evaluated the impact of anonymizing variable and method names in code search and code clone detection tasks using the pretrained model GraphCodeBERT [43]. In their experiments, they created anonymized datasets by replacing or shuffling identifier names with random strings. The model was then fine-tuned on these datasets, and its performance was evaluated on both tasks. As a result, the performance in the code search task dropped significantly after anonymization. For the clone detection task, the F1 score decreased from 94.87 to 84.76 at most. These findings suggest that while identifier names are important elements for LLMs, their impact on code clone detection is relatively limited.

The study by Wang et al. demonstrates that identifier names play a certain beneficial role in LLM-based clone detection. In this study, we extend this work and evaluate how identifier names and syntactic structure influence clone detection accuracy in larger LLMs. We also assess how syntactic structure, beyond identifier names, influences the clone detection process. By examining these two perspectives, we provide a comprehensive analysis of the factors that LLMs rely on when making clone detection decisions. Zhang et al. measured the accuracy of clone detection across Types 1–4 using GPT-3.5 and GPT-4. Their results indicated a correlation between model accuracy in clone detection and the degree of code similarity. For highly similar Type-1 clones, both models achieved a recall of 1.00. However, for the most complex Type-4 clones with less than 50% similarity, both models performed poorly. GPT-4 achieved a recall of 0.23 and GPT-3.5 achieved only 0.07.

8.3 Datasets for Clone Detection

There are several datasets that are used as benchmarks for clone research.

BigCloneBench BigCloneBench is a code clone detection benchmark created by Svajlenko et al. BigCloneBench extracts clone pairs by grouping methods that implement one of 45 tasks (e.g., file copying, bubble sort). This benchmark covers clone Type-1 to Type-4 and is often used to evaluate code clone detection tools. However, its suitability for training machine learning models is questionable due to potential biases and inconsistent labels [25].

SemanticCloneBench SemanticCloneBench [3] is a Type-4 clone dataset extracted from Stack Overflow, a programming Q&A site. Two judges manually verified each clone pair, and the dataset was assembled from their annotations.

GPTCloneBench GPTCloneBench [4] is a new Type-4 clone dataset created using gpt-3 based on the SemanticCloneBench dataset. Six judges reviewed the generated pairs, and the final dataset reflects their consensus. GPTCloneBench substantially extends SemanticCloneBench and contains approximately 77,000 samples, making it a promising candidate for use as training data in machine-learning-based approaches. However, because this dataset is generated using GPT, prior work has pointed out that such data may differ from datasets collected from real-world programs and may be easier for LLMs to de-

tect [5]. Moreover, although the dataset is augmented by generating additional programs with LLMs trained on real code, there is no guarantee that the generated programs reflect the characteristics of real-world code. This raises concerns about the suitability of this dataset for reliably evaluating LLM performance.

FEMPDataset FEMPDataset [14] is a collection of method pairs that are structurally different but functionally equivalent. This corresponds to Type-4 clones. These method pairs exhibit little syntactic similarity but perform the same function. The dataset was constructed in two stages: first by identifying candidate method pairs through cross-executing test cases, and then by confirming truly equivalent pairs through visual inspection. Unlike the three datasets mentioned above, FEMPDataset ensures functional equivalence by cross-executing test cases and expert review by three specialists. We use this dataset in our study because each method pair included in FEMPDataset is guaranteed to be functionally equivalent.

9 Conclusion

In this study, we systematically analyzed the impact of identifier names on the accuracy of code clone detection using LLMs. By evaluating the performance of six LLMs (GPT-4.1, Phi-4, Gemini-2.5-pro, Gemma-3-27b-it, CodeLlama-7B-Instruct, and CodeGemma-7b-it) on a dataset with masked identifiers from FEMPDataset, we obtained the following findings. For many models, clone detection performance was maintained or even improved when identifier names were masked. Furthermore, in high-performance models (GPT-4.1, Phi-4), the lowest performance occurred with the masking pattern that left only method names unmasked, demonstrating the strong influence of method names on clone detection. Our detailed analysis of GPT-4.1 and Phi-4 showed that both models rely on the lexical and semantic similarity of method names. The misclassification of non-clone pairs with high method name similarity remains a major issue. In contrast, the influence of word frequency on method name misclassifications was inconsistent and limited.

Regarding syntactic similarity, our analysis revealed that method pairs with lower syntactic similarity are more difficult to detect. In particular, models with fewer parameters (Gemma-3, CodeLlama, CodeGemma) struggle to detect pairs with low syntactic similarity, with F1-scores decreasing by approximately 0.4 compared to pairs with high syntactic similarity. In contrast, Gemini-2.5-pro maintains high detection accuracy even for data in the lowest syntactic similarity range, demonstrating robustness to syntactic similarity. Our analysis of the relationship between syntactic similarity and identifier names showed that identifier names have a greater impact on pairs with low syntactic similarity. In particular, retaining variable names improved performance in multiple models (Phi-4, Gemma, CodeLlama). This suggests that for pairs with low syntactic similarity, variable names play a complementary role in providing structural information. On the other hand, for pairs with high syntactic similarity, the performance difference due to the presence or absence of identifiers is small, and most models are robust to identifier names. Future research topics include developing selective masking processes using word frequency and extending our work to other programming languages.

Acknowledgement

Over the course of my research and the writing of this paper, I would like to express my sincere gratitude to all those who have supported and guided me.

First and foremost, I would like to thank my supervisor, Professor Yoshiki Higo, for his invaluable suggestions and continuous guidance throughout my research. In addition to his research advice, he provided extensive support in preparing presentation slides and developing effective presentation skills, and generously offered me many opportunities to present my work at external conferences.

I am also sincerely grateful to Professor Raula Gaikovina Kula, who provided extensive advice on the overall structure and organization of my paper, which greatly improved the clarity and quality of this work.

I would like to express my sincere gratitude to Assistant Professor Olivier Nourry for his valuable advice and thoughtful discussions throughout my research. I am also grateful for the opportunity to share informal discussions with him, including enjoying drinks together in Hokkaido.

I would like to thank Associate Professor Makoto Matsushita for his valuable comments and insightful suggestions on my research.

I am also grateful to Professor Katsuro Inoue for his valuable comments and advice. His feedback, particularly during domestic conferences, greatly helped me to improve my research.

I would like to thank Mrs. Mizuho Karube and Mrs. Takara Miyazaki for their kind support and assistance. I am especially grateful for their warm support in everyday matters, including administrative assistance and casual conversations when I visited the laboratory.

I am thankful to all the members of the Higo Laboratory for creating such an excellent research environment.

Finally, I would like to express my heartfelt gratitude to my family for their constant encouragement and support. They supported my education financially for six years and continuously encouraged me behind the scenes. Through regular communication and financial assistance, they have always supported me, and I deeply appreciate everything they have done for me.

References

[1] Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J. Hewett, Mojan Javaheripi, Piero Kauffmann, James R. Lee, Yin Tat Lee, Yuanzhi Li, Weishung Liu, Caio C. T. Mendes, Anh Nguyen, Eric Price, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Xin Wang, Rachel Ward, Yue Wu, Dingli Yu, Cyril Zhang, and Yi Zhang. Phi-4 Technical Report, 2024.

[2] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified Pre-training for Program Understanding and Generation. In Kristina Toutanova, Anna Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tur, Iz Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty, and Yichao Zhou, editors, *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 2655–2668, Online, June 2021. Association for Computational Linguistics.

[3] 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)*, pp. 57–63, 2020.

[4] 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 *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2023.

[5] Ajmain I. Alam, Palash R. Roy, Farouq Al-omari, Chanchal K. Roy, Banani Roy, and Kevin A. Schneider. Are Classical Clone Detectors Good Enough for the AI Era? . In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 295–307, Los Alamitos, CA, USA, September 2025. IEEE Computer Society.

[6] Afnan A. Almatrafi, Fathy A. Eassa, and Sanaa A. Sharaf. Code Clone Detection Techniques Based on Large Language Models. *IEEE Access*, Vol. 13, pp. 46136–46146, 2025.

[7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[8] Debarshi Chatterji, Jeffrey C. Carver, Nicholas A. Kraft, and Jan Harder. Effects of cloned code on software maintainability: A replicated developer study. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pp. 112–121, 2013.

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code, 2021.

[10] CodeGemma Team. Codegemma: Open code models based on gemma, 2024.

[11] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547, 2020.

[12] Gemini Team. Gemini: A Family of Highly Capable Multimodal Models, 2023.

[13] Gemma Team. Gemma 3 Technical Report, 2025.

[14] Yoshiki Higo. Dataset of Functionally Equivalent Java Methods and Its Application to Evaluating Clone Detection Tools. *IEICE Trans. Inf. & Syst.*, Vol. E107.D, No. 6, pp. 751–760, 2024.

[15] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.*, Vol. 33, No. 8, December 2024.

[16] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-Rank Adaptation of Large Language Models. In *International Conference on Learning Representations (ICLR)*, 2022.

[17] Ryutaro Inoue and Yoshiki Higo. Improving Accuracy of LLM-based Code Clone Detection Using Functionally Equivalent Methods. In *Proceedings of the 22nd IEEE/ACIS International Conference on Software Engineering Research, Management and Applications (SERA)*, pp. 24–27, 2024.

[18] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pp. 96–105, 2007.

[19] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pp. 485–495, USA, 2009. IEEE Computer Society.

[20] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Software Engineering*, Vol. 28, No. 7, pp. 654–670, 2002.

[21] Mohamad Khajezade, Jie Wu, Fatemeh Fard, Gema Rodriguez, and Mohamed Shehata. Investigating the Efficacy of Large Language Models for Code Clone Detection. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension (ICPC2024)*, pp. 161–165, 2024.

[22] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin. An Ethnographic Study of Copy and Paste Programming Practices in OOPL. In *Proceedings of the*

2004 International Symposium on Empirical Software Engineering, ISESE '04, pp. 83—92, USA, 2004. IEEE Computer Society.

[23] Seulbae Kim and Heejo Lee. Software systems at risk: An empirical study of cloned vulnerabilities in practice. *Comput. Secur.*, Vol. 77, No. C, pp. 720—-736, August 2018.

[24] Rainer Koschke. Survey of Research on Software Clones. In Rainer Koschke, Ettore Merlo, and Andrew Walenstein, editors, *Duplication, Redundancy, and Similarity in Software*, Vol. 6301 of *Dagstuhl Seminar Proceedings (DagSemProc)*, pp. 1–24, Dagstuhl, Germany, 2007. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[25] J. Krinke and C. Ragkhitwetsagul. BigCloneBench Considered Harmful for Machine Learning. In *Proceedings of the 16th IEEE International Workshop on Software Clones (IWSC)*, pp. 1–7, 2022.

[26] M. Mondal, C. Roy, and K. Schneider. A Fine-Grained Analysis on the Inconsistent Changes in Code Clones. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 220–231, 2020.

[27] Micheline Bénédicte Moumoula, Abdoul Kader Kaboré, Jacques Klein, and Tegawendé F. Bissyandé. The Struggles of LLMs in Cross-Lingual Code Clone Detection. *Proc. ACM Softw. Eng.*, Vol. 2, No. FSE, June 2025.

[28] Tasuku Nakagawa, Yoshiki Higo, and Shinji Kusumoto. NIL: large-scale detection of large-variance clones. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, ESEC/FSE 2021, pp. 830—-841. Association for Computing Machinery, 2021.

[29] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Haiquan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *International Conference on Learning Representations*, 2022.

[30] OpenAI. GPT-4 Technical Report, 2023.

[31] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. ZeRO: Memory optimizations Toward Training Trillion Parameter Models. In *SC20: International Conference for*

High Performance Computing, Networking, Storage and Analysis, pp. 1–16, 2020.

[32] C. Roy and J. Cordy. A Survey on Software Clone Detection Research. *School of Computing TR 2007-541*, pp. 3–7, 01 2007.

[33] C. Roy and J. Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC)*, pp. 172–181, 2008.

[34] C. Roy, J. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, Vol. 74, No. 7, pp. 470–495, 2009.

[35] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code Llama: Open Foundation Models for Code, 2024.

[36] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V. Lopes. Oreo: detection of clones in the twilight zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, p. 354–365, New York, NY, USA, 2018. Association for Computing Machinery.

[37] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. Sourcerercc: Scaling code clone detection to big-code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 1157–1168, 2016.

[38] Yewei Song, Cedric Lothritz, Xunzhu Tang, Tegawendé Bissyandé, and Jacques Klein. Revisiting code similarity evaluation with abstract syntax tree edit distance. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, p. 38–46. Association for Computational Linguistics, 2024.

[39] J. Svajlenko, J. Islam, I. Keivanloo, C. Roy, and M. Mia. Towards a Big Data Curated Benchmark of Inter-project Code Clones. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 476–480, 09 2014.

[40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NeurIPS)*, p. 6000–6010, 2017.

[41] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree. In *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 261–271, 2020.

[42] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 8696–8708, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.

[43] Zhilong Wang, Lan Zhang, Chen Cao, Nanqing Luo, Xinzhi Luo, and Peng Liu. How Does Naming Affect Language Models on Code Analysis Tasks? *Journal of Software Engineering and Applications*, Vol. 17, pp. 803–816, 01 2024.

[44] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc.

[45] Reishi Yokomori and Katsuro Inoue. An Empirical Analysis of Code Clone Authorship in Apache Projects . In *2023 IEEE 17th International Workshop on Software Clones (IWSC)*, pp. 1–7, Los Alamitos, CA, USA, October 2023. IEEE Computer Society.

[46] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, pp. 783–794, 2019.

[47] Zixian Zhang and Takfarinas Saber. Assessing the code clone detection capability of large language models. In *2024 4th International Conference on Code Quality (ICCQ)*, pp. 75–83, 2024.

[48] Wayne Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, and Ji-Rong Wen. A Survey of Large Language Models, 2023.