big data and cognitive computing

MDPI

Article

Exploring the Boundaries Between LLM Code Clone Detection and Code Similarity Assessment on Human and AI-Generated Code

Zixian Zhang [1,*,†] and Takfarinas Saber [2,†]

[1] CRT-AI, School of Computer Science, University of Galway, H91 TK33 Galway, Ireland
[2] Lero, School of Computer Science, University of Galway, H91 TK33 Galway, Ireland; takfarinas.saber@universityofgalway.ie
* Correspondence: z.zhang15@universityofgalway.ie
† These authors contributed equally to this work.

Abstract: As Large Language Models (LLMs) continue to advance, their capabilities in code clone detection have garnered significant attention. While much research has assessed LLM performance on human-generated code, the proliferation of LLM-generated code raises critical questions about their ability to detect clones across both human- and LLM-created codebases, as this capability remains largely unexplored. This paper addresses this gap by evaluating two versions of LLaMA3 on these distinct types of datasets. Additionally, we perform a deeper analysis beyond simple prompting, examining the nuanced relationship between code cloning and code similarity that LLMs infer. We further explore how fine-tuning impacts LLM performance in clone detection, offering new insights into the interplay between code clones and similarity in human versus AI-generated code. Our findings reveal that LLaMA models excel in detecting syntactic clones but face challenges with semantic clones. Notably, the models perform better on LLM-generated datasets for semantic clones, suggesting a potential bias. The fine-tuning technique enhances the ability of LLMs to comprehend code semantics, improving their performance in both code clone detection and code similarity assessment. Our results offer valuable insights into the effectiveness and characteristics of LLMs in clone detection and code similarity assessment, providing a foundation for future applications and guiding further research in this area.

Keywords: code clone detection; code similarity; large language model; fine-tuning; LLM-generated code

check for updates

Academic Editors: Ximing Li, Zuchao Li and Min Peng

Received: 11 December 2024
Revised: 23 January 2025
Accepted: 7 February 2025
Published: 13 February 2025

Citation: Zhang, Z.; Saber, T. Exploring the Boundaries Between LLM Code Clone Detection and Code Similarity Assessment on Human and AI-Generated Code. *Big Data Cogn. Comput.* **2025**, *9*, 41. https://doi.org/10.3390/bdcc9020041

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

In software engineering, code clones refer to code segments that are identical or similar to other segments within or across different software systems [1]. Code clones can result in increased maintenance efforts, inconsistencies during bug fixes, and difficulties in managing and evolving software systems [2,3].

For a comprehensive evaluation, researchers commonly classify code clones into four types: **Type-1** (identical similarity), **Type-2** (lexical similarity), **Type-3** (syntactical similarity), and **Type-4** (semantic similarity). Some well-known tools, including CCFinder [4], SourcererCC [5], and NiCad [6], are effective in detecting the first two types and partially effective in detecting the third type. However, effectively detecting Type-3 and Type-4 clones remains challenging due to the complexity of semantic analysis and syntactical dissimilarity.

In recent years, numerous studies have focused on semantic similarity using deep learning-based models. These models include from-scratch architectures such as Recurrent Neural Network (RNN) based models [7–9], Convolutional Neural Network (CNN) based models [10,11], and Graph Neural Network (GNN) based models [12–14]. Traditional models like RNNs, CNNs and GNNs primarily rely on handcrafted features or carefully designed input representations to capture structural or syntactic information in the code, facing challenges in capturing nuanced semantic relationships. Recent advancements in Large Language Models (LLMs) have revolutionized the field by offering pre-trained architectures capable of understanding code semantics at a deeper level. Unlike traditional models, LLMs are trained on a vast corpus of code and natural language, enabling them to generalize across multiple programming languages and tasks without extensive task-specific feature engineering. However, most existing research primarily focuses on analyzing the performance of LLMs in code clone detection with human-generated codes [15–18].

Despite these advancements, there is a significant gap in understanding LLMs' effectiveness in detecting code clones across the emerging and increasingly prevalent LLM-generated code, such as those produced by tools like ChatGPT's Code Interpreter, GitHub Copilot, and Cursor. Existing research does not explicitly address how LLMs perform when dealing with clones within and across human-generated and LLM-generated codebases, leaving the impact of these differences unexplored. To address this gap, we assess the difference in LLMs (in our case, two LLaMA3 models) code clone detection capabilities between human-generated and LLM-generated code corpora.

Furthermore, we expand our analysis beyond the simple performance analysis: (i) we delve deeper into the relationship between LLMs' code clone detection and code similarity inference, analyzing the nuanced ways in which LLMs interpret these concepts, and (ii) we explore the impact that LLM fine-tuning has on these two concepts, aiming to understand how these adjustments influence model effectiveness.

Our research paper aims particularly to answer the following Research Questions (RQs):

- **RQ1: How effective are LLMs at detecting code clones when using zero-shot prompts?** Our research aims to explore the performance of LLMs in detecting code clones using a zero-shot prompt approach. Specifically, we formulate a prompt that directly requests the LLMs to make a judgment regarding code clone detection. We plan to analyze the results based on the size of LLM parameters and the types of clones.
- **RQ2: What differences exist in LLMs' detection capabilities between human-generated and LLM-generated code clones?** Our research explores LLMs' effectiveness in detecting human-generated versus LLM-generated code clones. We use two datasets: one is human-made (i.e., BigCloneBench), whereas the other one is LLM-generated (GPTCloneBench). By comparing the detection capabilities of LLMs on these distinct datasets, we aim to uncover how well LLMs react to the variations in different sources of clones.
- **RQ3: Is there a relationship between LLM's code clone detection and code similarity inference?** In this research question, we aim to investigate the mechanisms by which LLMs identify similarities between pairs of code snippets. Our focus is on understanding the criteria LLMs use to define code similarity, and examining how these criteria compare to syntactic similarity. By delving into this question, we hope to uncover the underlying processes LLMs use to detect code clones, enhancing our understanding of their operational frameworks and potential biases in code analysis.
- **RQ4: What is the impact of LLM fine-tuning on its code clone detection and code similarity inference?**

This research question focuses on assessing the impact of LLM fine-tuning on code clone detection. We employ parameter-efficient fine-tuning techniques on the code clone detection performance and its relationship with the inferred code similarity.

This paper is organized as follows: Section 2 introduces the background necessary to understanding this work (code clone detection and LLM fine-tuning) and the work related to this research. Section 3 provides our experimental set-up information. Section 4 reports our experimental results. Section 5 discusses the threats to the validity of our work and future work. Section 6 concludes this study.

2. Background and Related Work

In this section, we describe the background necessary to the understanding of our study in two parts (i.e., Code Clone Detection and LLM Fine-Tuning) as well as its related work.

2.1. Code Clone Detection

The objective of code clone detection is to identify code fragments that exhibit both syntactic and semantic (or functional) similarities. The commonly accepted classification strategy categorizes code clones into four distinct types [19]:

Type-1: Identical code fragments, where variations are limited to comments, layout, and whitespace. **Type-2:** Structurally or syntactically identical code fragments, where the primary distinctions arise from variations in identifier names and literal values, including those differences specified in Type-1. **Type-3:** Syntactically similar fragments that exhibit differences at the statement level. In addition to the variations found in Type-1 and Type-2 clones, these code fragments may have statements that are added, removed, or modified. **Type-4:** Refers to code fragments that are syntactically dissimilar yet implement the same functionality.

Svajlenko et al. [20] proposed a more detailed categorization scheme for Type-3 and Type-4 clones based on syntactical similarity. Specifically, they defined Very-Strongly Type-3 (VST3) clones as those exhibiting a similarity ranging from 90% (inclusive) to 100%, Strongly Type-3 (ST3) as those between 70% and 90%, Moderately Type-3 (MT3) as those between 50% and 70%, and Weakly Type-3 or Type-4 (WT3/4) as those between 0% and 50%. To the best of our knowledge, this scheme is employed by BigCloneBench [21] and the GPTCloneBench [22].

With the advancement of Machine Learning (ML) techniques, various ML-based tools have been proposed for detecting code clones, including those based on RNN [7–9], CNN [10,11], and GNN [12–14]. However, despite the rapid development of LLMs, there have been limited explorations into using LLMs for code clone detection. To the best of our knowledge, several research efforts have evaluated the capabilities of LLMs through prompt-engineering techniques, including simple-prompt [18], one-shot prompting [23], and chain-of-thought approaches [15]. Nevertheless, there is a noticeable gap in research regarding how fine-tuning techniques influence the capabilities of LLMs in code clone detection.

2.2. Large Language Models and Fine-Tuning

LLMs are language models trained using transformer-based frameworks on a large corpus, such as QWen [24], LLaMA [25], and Phi-3 [26]. With high parameters and vast training datasets, these models are adept at generating, understanding, and manipulating human-like text across a wide range of applications.

Fine-tuning is the process of adjusting model weights through learning on a downstream task. This method can significantly decrease training time and resource costs while effectively enhancing performance in specific tasks [27]. However, due to the extensive

parameters of LLMs, fine-tuning LLMs typically requires substantial GPU memory and is time-consuming. To mitigate these consumption issues, our research implements two specific fine-tuning techniques.

- **Quantized Low-Rank Adaptation (QLoRA):** QLoRA [28] is a technique designed to enhance the efficiency of LLMs by reducing their computational and memory demands without significantly compromising performance. It achieves this by integrating quantization and low-rank adaptation strategies [29]. Specifically, QLoRA employs 4-bit quantization to compress a pre-trained language model and utilizes Low-Rank Adaptation (LoRA) to transform models' weights into trainable low-rank matrices.
- **Zero Redundancy Optimizer (ZeRO):** ZeRO [30] is a memory optimization framework that facilitates the scaling of deep learning models [30]. It partitions model states—optimizer states, gradients, and parameters—across multiple devices, thereby utilizing their combined memory capacity. This strategy not only reduces the memory load per device but also enhances the efficiency of the training process across multiple GPUs, allowing for the training of considerably larger models than was previously feasible.

2.3. Related Work

With the rapid advancements in deep learning, numerous studies have utilized deep learning models for code clone detection tasks. A variety of network architectures have been explored in this domain, including CNNs [31], RNNs [32], and GNNs [33,34], among others. In addition, pre-trained embedding models have gained significant traction in this field, with notable implementations such as CodeBERT [35], GraphCodeBERT [36], and UniXcoder [37].

More recently, LLMs have emerged as transformative tools, significantly enhancing their applications across various stages of the software development lifecycle, such as code generation, code summarization, bug detection, and repair. Code clone detection is recognized as a key component of the software development process, essential for promoting efficient code reuse and maintenance. However, despite its importance, the application of LLMs to this task has not yet been extensively explored or rigorously analyzed.

Some of the most notable research efforts in this area include the work of Dou et al. [15], which provides a comprehensive evaluation of LLMs for code clone detection by employing methodologies such as simple prompts, chain-of-thought reasoning, code embeddings, and cross-language tests. Similarly, Moumoula et al. [16] investigate how LLMs compare with pre-trained embedding models in detecting cross-lingual code clones. While both studies use large-scale benchmarks, they rely on binary 'yes' or 'no' responses from the models, which may introduce inconsistencies. Further contributing to this growing body of research, Inoue et al. [17] examine the effects of fine-tuning LLMs, such as GPT-3.5 Turbo and Llama2-Chat-7B, for code clone detection on small-scale datasets. Khajezade et al. [18] also assess ChatGPT's performance in this domain, offering additional insights into the strengths and limitations of LLMs for code clone detection.

Our research differentiates itself by being the first to analyze code similarity and the clone detection process from the perspective of LLMs using a large-scale benchmark. This study also explores the effects of fine-tuning on LLM performance in detecting code clones between human-generated and LLM-generated code, offering new insights into the efficacy of LLMs in these tasks.

3. Experimental Set-Up

In this section, we detail the set-up of our experimental analysis in six steps: dataset, prompts, LLMs, fine-tuning techniques, valuation metrics, and other configurations. We

make the code related to this work available online (Available at https://github.com/ZixianReid/LLM_Code_Clone_Validation (accessed on 10 February 2025)).

3.1. Dataset Collection

Our evaluations were conducted using the BigCloneBench [21] and the GPTCloneBench dataset [38].

The BigCloneBench dataset is a comprehensive and widely utilized benchmark for clone detection, containing a collection of 8 million validated clones with code authors by humans. It was selected due to its extensive size, rigorous validation process, and established reputation as a standard benchmark in the field. This ensures comparability and reproducibility of results across studies. For this experiment, we employed a selected version of BigCloneBench [39], excluding code fragments that lacked clone pair assessment (i.e., no true or false tagging), thereby providing a more effective ML-oriented dataset. We dedicate the test set in BigCloneBench for our evaluations and retain the train set in BigCloneBench for fine-tuning purposes. The code segments in BigCloneBench are function-level, and assigned with varying syntactical levels and similarity scores. According to the BigCloneBench evaluation framework [20], the clone pairs are categorized into Type-1, Type-2, VST3, ST3, MT3, WT3/T4. The demographics of the test set used in this research are detailed in Table 1.

Table 1. Clone Type Demographics in BigCloneBench.

Clone Type	Non-Clone	T1	T2	ST3	MT3	WT3/T4
Number	358,596	146	68	143	593	55,880

The GPTCloneBench dataset is an extensive benchmark of GPT-generated code clones specifically designed for evaluating machine learning tools in detecting semantic and cross-language code clones. It compiles datasets using code fragments from SemanticCloneBench [22] as input for the GPT-3 model. This includes true semantic clones labeled as MT3 and WT3/T4, false semantic pairs (Type-1/Type-2), and cross-language clones in Java, C, C#, and Python. It was chosen to address the emerging need to assess clone detection in LLM-generated code, reflecting real-world trends where AI-generated code is increasingly prevalent. For this study, we focused on the Java segment of the dataset for evaluation purposes. The demographics of the dataset employed in this research are detailed in Table 2.

By combining these two datasets, our evaluations benefit from a comprehensive analysis of both human-authored clones and modern AI-generated clones, ensuring a well-rounded assessment of LLM performance in code clone detection.

Table 2. Clone Type Demographics in GPTCloneBench.

Clone Type	T1-2	MT3	WT3/T4
Number	5443	4116	7499

3.2. Prompt Engineering

3.2.1. Prompt for Classifying Code Clone

The prompt utilized for classifying code clones is presented below:

> ###Instruction:
> Please analyze the following two code snippets and determine if they are code clones. Respond with only ###RESULT###YES for clones or ###RESULT###NO if not. Provide no other output.
> Code snippet 1: $code_1
> Code snippet 2: $code_2
> ###Response:

We observed that when directly prompting an LLM to output 'yes' or 'no', the results could sometimes deviate unexpectedly, such as producing responses like 'The answer is no, and the reason is ...'. Unlike other related studies that typically generate simple 'yes' and 'no' flags for recognition [15,40], we sought to achieve clearer outputs from the LLM. Therefore, we employed more distinctive flags: `###RESULT###YES` for affirmative and `###RESULT###NO` for negative responses. This modification ensured more stable outputs from the LLM during our validation process.

3.2.2. Prompt for Recognizing Code Similarity

We ask the model to output the similarity between two code snippets on a scale of 0 to 10. By analyzing the LLM-inferred similarity scores for code pairs, we would like to investigate the relationship that LLMs associate between code similarity and code clone concepts. The prompt utilized for recognizing code similarity is presented below:

> ###Instruction:
> Compare the following two code snippets and rate their similarity on a scale from 0 to 10, where 0 indicates no similarity and 10 indicates maximum similarity. Respond with only the similarity score as your output.
> Code snippet 1: $code_1
> Code snippet 2: $code_2
> ###Response:

3.3. Large Language Models

In our experiments, we employ the LLaMA3 model. The LLaMA series, as described by Touvron et al. [25], marks a significant advancement in the field of NLP, utilizing transformer architecture and diverse datasets throughout the processes of pre-training and fine-tuning. The pre-training phase involves predicting the next word in sentences to internalize language patterns, followed by fine-tuning specific tasks. This training process is notable for its scalability and efficiency, leveraging distributed computing and advanced optimization techniques to meet substantial computational demands. The LLaMA family comprises two branches: LLaMA, a general-purpose model trained across various NLP tasks, and LLaMA-Instruct, specifically fine-tuned to adhere to detailed user instructions and generate precise responses. We utilized LLaMA3-Instruct, the third iteration in the LLaMA series. For zero-shot prompt-based inference, we deployed both the 8 billion parameter model, LLaMA3-8B-Instruct, and the 70 billion parameter model, LLaMA3-70B-Instruct. Due to constraints on GPU resources Table 3.

Table 3. Comparison of SOTA code clone detection methods and LLaMA3 in BigCloneBench.

Method	Recall							Precision	F1-Score
	T1	T2	VST3	ST3	MT3	T4	Overall		
Deckard [41]	0.60	0.58	-	0.420	0.120	0.010	0.020	0.930	0.030
SourcererCC [5]	1.000	0.98	-	0.610	0.05	0.000	0.01	0.88	0.010
Token-transformer [42]	1.000	1.000	-	1.000	0.985	0.948	0.952	0.961	0.956
ASTNN [7]	1.000	1.000	-	0.998	0.970	0.932	0.934	0.947	0.941
FA-AST + GMN [43]	1.000	1.000	-	0.996	0.979	0.922	0.924	0.934	0.929
CodeBERT [44]	1.000	1.000	-	0.998	0.970	0.932	0.934	0.947	0.941
LLaMA3-8b-instruct	1.000	0.927	0.9444	0.769	0.718	0.170	0.180	0.631	0.281
LLaMA3-70b-instruct	1.000	0.900	0.9444	0.727	0.401	0.03	0.044	0.885	0.034

3.4. Fine-Tuning Techniques

To mitigate GPU memory consumption, we integrated the ZeRO [30] and QLoRA [28] techniques. Specifically, ZeRO was utilized to facilitate parameter and optimizer offloading, enabling the storage of model parameters, gradients, and optimizers in the CPU or on disk. For QLoRA, we selected the NF4 (normalized float 4) configuration with double quantization, employing bfloat16 as the computation data type. The hyper-parameter r for LoRA was set at 64. Fine-tuning was conducted on a single NVIDIA RTX 6000 graphics card.

3.5. Evaluation Metrics

3.5.1. Machine Learning Metrics

We use Recall, Precision, and F1 Score which are widely used for classification tasks as performance metrics. The definition and formulas of these metrics are shown below:

Precision measures the accuracy of positive predictions:

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

where TP is the number of true positives and FP is the number of false positives.

Recall assesses the ability of the classifier to identify all relevant instances:

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

where TN is the number of false negatives.

F1 Score is the harmonic mean of Precision and Recall:

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

3.5.2. Syntactic Similarity Score

In this study, we use syntactic similarity scores of code clones as a reference for comparing the similarity scores inferred by LLMs. For the BigCloneBench dataset, we utilize the syntactic line similarity scores provided by BigCloneEval [20]. These scores are calculated as the proportion of shared lines between code fragments, employing the Unix-diff tool [45], following Type-1 and Type-2 normalization to account for variations in comments, whitespace, and identifier names. For the GPTCloneBench dataset, which lacks

precomputed syntactic similarity scores, we calculate them using the same methodology as proposed in BigCloneEval.

3.6. Hardware and Parameters

The experiments were conducted on a server outfitted with a single NVIDIA RTX 6000 graphics card. Additionally, for the inference tasks involving the LLaMA3-70B-Instruct model, we utilized the Huggingface API service [46]. Related parameters used in this study are shown in Table 4.

Table 4. Related parameters used in this research.

Parameter	Value
Temperature	1.0
Training epoch	2×10^{-4}
LORA_rank	32
LOAR_Alpha	16
Learning Rate	2×10^{-4}
Optimizer	paged_adamw_32bit
Zero optimization stage	3

4. Experimental Results

In this section, we describe the results of our experimental analysis and attempt to answer each of the defined research questions.

4.1. Effectiveness of LLMs with Zero-Shot Prompts (RQ1)

In this research question, we would like to analyze the variance in performance across different parameter sizes of LLaMA3 and how such performances compare against non-LLM methods.

Table 5 presents a comparison of Recall performance for code clone detection across various methods, including LLaMA3 models (i.e., LLaMA3-8b-instruct and LLaMA3-70b-instruct), state-of-the-art deep learning methods (i.e., Token-transformer [42], FA-AST + GMN [43]), and BERT-based methods (i.e., CodeBERT [44]). Results are shown for both human-generated and AI-generated code variants using two benchmarks: BigCloneBench and GPTCloneBench.

From Table 5, we observe that LLaMA3 models exhibit good accuracy in detecting T1 and T2 code clones in BigCloneBench. However, there is a notable degradation in performance for MT3 and T4 in comparison to other trained state-of-the-art (SOTA) models. This decline suggests that the pre-trained LLaMA3 models may lack the requisite capabilities to effectively infer semantically similar code clone pairs. A comparative analysis between the two LLaMA3 variants indicates that LLaMA3-8b consistently outperforms LLaMA3-70b across nearly all categories within both datasets. This observation may point to differences in how model architecture or training datasets impact clone detection efficacy.

In the comparative analysis between LLaMA3 models and traditional non-LLM approaches, it is evident that LLaMA3 demonstrates superior efficacy in identifying T1 and T2 clones, achieving perfect recall rates comparable to other SOTA models, These SOTA models include both traditional non-learning methods and advanced machine learning-based tools. However, the performance of LLaMA3 models significantly diminishes when detecting more semantically complex clone types such as MT3 and T4 clones. This decline is in stark contrast to the performance exhibited by non-LLM models and other advanced deep

learning frameworks, such as Token-transformer, ASTNN, FA-AST + GMN, and CodeBERT. These models exhibit robust capabilities across all clone types, consistently maintaining high levels of recall and precision.

Key Findings

- LLaMA3 models excel in detecting T1 and T2 code clones but falter with MT3 and T4 types without training compared with other SOTA models.
- LLaMA3-8b outperforms LLaMA3-70b in most categories.

Table 5. Comparison of LLaMA3 Model Performance in terms of Recall for Code Clone Detection over Human-Generated vs. AI-Generated Code Variants.

Method	BigCloneBench					GPTCloneBench			
	T1	T2	MT3	T4	Overall	T1-2	MT3	T4	Overall
Token-transformer [42]	1.000	1.000	0.985	0.948	0.952	0.877	0.173	0.132	0.337
FA-AST + GMN [43]	1.000	1.000	0.979	0.922	0.924	0.988	0.984	0.750	0.940
CodeBERT [44]	1.000	1.000	0.970	0.932	0.934	1.000	1.000	0.996	0.998
LLaMA3-8b-instruct	1.000	0.927	0.718	0.170	0.180	0.858	0.867	0.849	0.856
LLaMA3-70b-instruct	1.000	0.900	0.401	0.03	0.044	0.850	0.772	0.520	0.686

4.2. Variation in Detection Capability between Human and GPT-Generated Clones (RQ2)

Our analysis concentrates on the performance of LLaMA3 models in identifying code clones within datasets comprised of human-authored code (i.e., BigCloneBench) versus those generated by GPT algorithms (i.e., GPTCloneBench).

For this comparative assessment, we include state-of-the-art models such as Token-transformer [42], FA-AST + GMN [43], and CodeBERT [44] as benchmarks. It is critical to acknowledge that the GPTCloneBench lacks non-clone pairs, which necessitates that Token-transformer, ASTNN, FA-AST + GMN, and CodeBERT be trained exclusively on the BigCloneBench and subsequently tested across both datasets. Comprehensive comparisons are detailed in Table 5, illustrating the differential performance of these models across the two datasets. Due to the absence of non-clone pairs in the GPTCloneBench dataset, the recall was the sole evaluation metric employed for assessing performance on this dataset.

The performance metrics, illustrated in Table 5, reveal a stark contrast in the recall rates between these two datasets. For human-generated code in BigCloneBench, LLaMA3 models demonstrate good recall across simpler clone types (T1 and T2), with a significant drop in performance for more complex clones (MT3 and T4). Conversely, when evaluating the performance on GPTCloneBench, which comprises LLM-generated code clones, both models report a much higher performance in MT3 and T4 code clones, although still lower than the SOTA models like CodeBERT, which nearly perfects recall at 0.998 overall. This suggests difficulties in inferring semantically similar or syntactically dissimilar code when originating from humans and a potential model bias toward recognizing semantic code clones more effectively in certain datasets for the LLaMA3 model. These observations suggest a potential challenge for the LLaMA3 models to accurately identify semantically similar or syntactically divergent code originating from human programmers while indicating a possible model predisposition towards more effectively recognizing semantic code clones within specific datasets.

Key Findings

- LLaMA3 models excel in detecting simple clones in human-authored code but perform better on complex clones in LLM-generated code.
- LLaMA3 models may struggle with human code complexity and show potential bias towards LLM-generated code.

4.3. Effect of Fine-Tuning on LLMs' Code Clone Detection Capabilities (RQ3)

In this research question, we examine the impact of fine-tuning techniques on the performance of the LLaMA3 model in code clone detection tasks for both the human-made dataset and the LLM-generated dataset. The model was trained using the BigCloneBench dataset and evaluated on both the BigCloneBench and GPTCloneBench.

We use LLaMA3-8b-instruct as it achieves better performance results over its 70b counterpart and also due to the limiting resource constraints associated with fine-tuning such a large model.

Table 6 presents a comparison of the performance of LLaMA3-8b-instruct and its fine-tuned variant for code clone detection. The results are evaluated in terms of Recall across two benchmarks: BigCloneBench and GPTCloneBench. We observe that the fine-tuned model achieves perfect recall for Types 1 and 2 clones in the BigCloneBench dataset, marking an enhancement from a recall of 0.927 in Type 2 for the baseline model. However, for semantic code clones (VST, ST3, MT3, T4), while the fine-tuned model generally exhibits improved recall scores over the base model, a notable exception is observed in Very Similar Type 3 where recall significantly decreases to 0.5000 from 0.9444. Conversely, in the LLM-generated dataset which is GPTCloneBench, the improvements afforded by fine-tuning are less pronounced. The fine-tuned model shows improved recall in Types 1 and 2 but exhibits decreased performance in MT3 and T4 compared to the baseline model. These findings suggest that the fine-tuning process might introduce a bias towards the specific types of data and patterns present in the human-made dataset. This can enhance the performance when the model encounters similar data (both the training set and testing sets of BigCloneBench are from the same project) but can degrade performance on datasets with different characteristics, like those generated by AI models, which may not follow the same patterns or contain the same biases (GPTCloneBench).

Table 6. Comparison of LLaMA3-8b-instruct and fine-tuned LLaMA3-8b-instruct.

Method	BigCloneBench									GPTCloneBench			
	Recall							Precision	F1	Recall			
	T1	T2	VST3	ST3	MT3	T4	Overall			T1-2	MT3	T4	Overall
LLaMA3-8b-instruct	1.000	0.927	0.9444	0.769	0.718	0.170	0.180	0.631	0.281	0.858	0.867	0.849	0.856
LLaMA3-8b-instruct-fine-tuned	1.000	1.000	0.5000	0.855	0.787	0.345	0.355	0.726	0.476	0.918	0.454	0.377	0.565

Figure 1 provides visual representations of these analyses, with histograms illustrating the performance of LLaMA3 8b and fine-tuned LLaMA3 8b across the BigCloneBench and GPTCloneBench datasets. Each subplot displays the proportion of true positives (in green) and false positives (in red) as a function of the syntactic similarity scores of code clones, which range from 0 to 1. The upper subplots illustrate the performance of LLMs on the BigCloneBench dataset. As the syntactic similarity scores decrease, we observe a corresponding decline in model performance. Fine-tuning the LLaMA3-8B model significantly improves its performance for code clones with low similarity scores, especially those below 0.5, which are indicative of highly semantic clones. However, this fine-tuning

technique appears to have a detrimental effect on the GPTCloneBench dataset, particularly for clones with lower similarity scores. This suggests that while fine-tuning enhances the model's ability to assimilate semantic information from human-generated code clones in a zero-shot prompt setting, it may introduce biases that reduce its effectiveness for LLM-generated code.

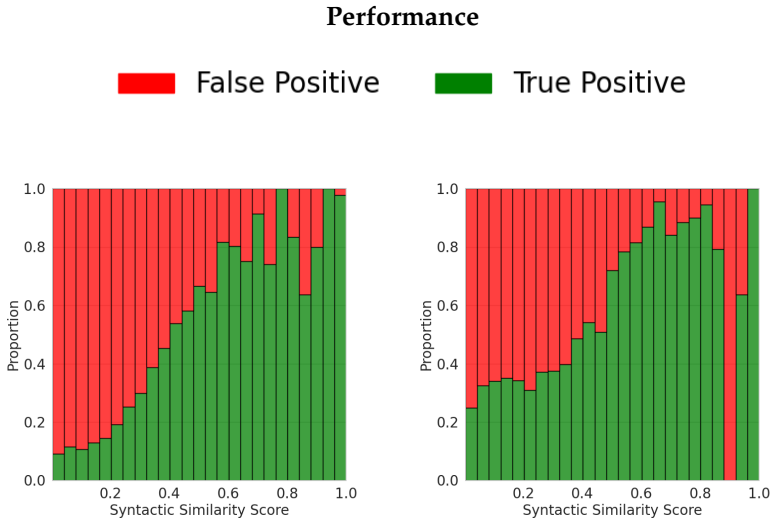


Figure 1. Performance comparison of fine-tuned and non-fine-tuned LLaMA3 8B models by code clone similarity scores.

The observed performance disparities between human-generated and LLM-generated code clones highlight the model's reduced efficacy on the GPTCloneBench dataset, underscoring the need for further investigation into these differences.

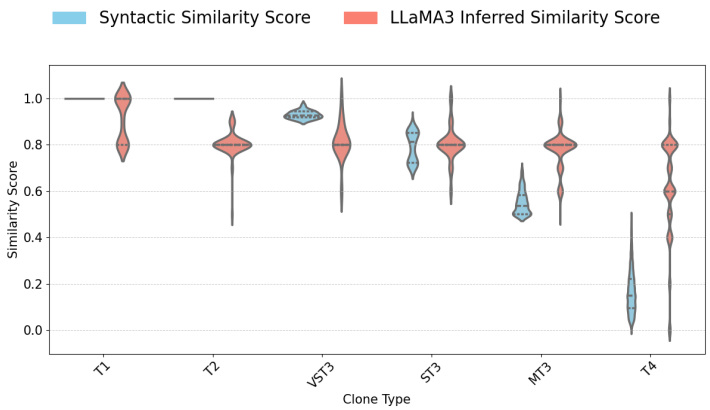
Key Findings

- The fine-tuned LLaMA3 model shows improved performance on the BigCloneBench dataset, achieving perfect recall for T1 and T2 clones, alongside notable advancements in similarity scores across various categories of code clones.
- The fine-tuning process based on a human-made dataset may bias the LLM model, leading to a reduction in its effectiveness in detecting LLM-generated code.

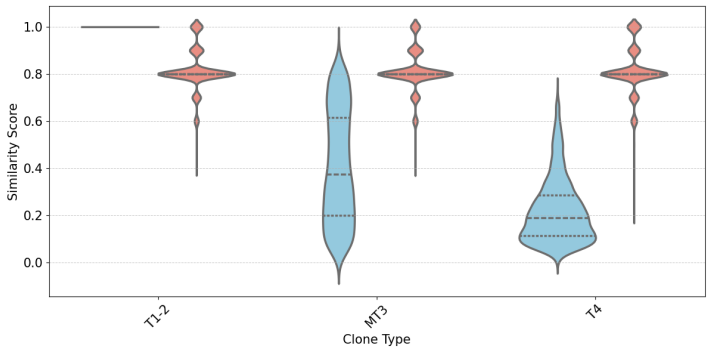
4.4. Relationship between LLM Code Clone Detection and Code Similarity Inference (RQ4)

In this research question, we aim to evaluate how LLMs comprehend code similarity, particularly by comparing their understanding to syntactic similarity and examining the differences between human-made and LLM-generated code clones.

Figure 2 presents a comparative analysis of syntactic similarity and the similarity as inferred by the LLaMA3 model (LLaMA3-8b-instruct) for various types of code clones across two benchmark datasets: BigCloneBench (Human-made) and GPTCloneBench (LLM-generated). In the BigCloneBench dataset, the disparity between LLaMA3's inferred similarity scores and syntactic similarity scores increases as the clone type progresses from syntactic to semantic clones. For T1 and T2 clones, the LLaMA3 similarity scores are slightly below 1.00, indicating that modifications to comments and variable names have some impact on the model's assessment of similarity. This suggests that while the code structure remains nearly identical, even minor changes can affect the model's perception. For VST3 and ST3 clones, the similarity scores show a relative consistency between syntactic measures and LLaMA3's inferred similarities, indicating that the model captures similarity in a way that aligns closely with structural features. However, for MT3 and T4 clones, there is a pronounced increase in LLaMA3's similarity scores compared to syntactic similarity, suggesting the model's ability to recognize deeper semantic similarities that are not captured by syntactic analysis alone. Additionally, the distribution of similarity scores for T4 is notably broader than for other clone types, indicating higher variability in how LLaMA3 assesses semantic similarity within this category. In contrast, the GPTCloneBench dataset exhibits a different pattern: the distributions of LLaMA3 similarity scores for T1-2, MT3, and T4 clones are relatively similar, with mean values around 0.8. This consistency suggests that, for LLM-generated code clones, the LLaMA3 model tends to perceive different clone types as similarly related at a semantic or functional level, despite structural differences.



(**a**) Human-Generated Code: BigCloneBench



(**b**) LLM-Generated Code: GPTCloneBench

Figure 2. Performance comparison of fine-tuned and non-fine-tuned LLaMA3 8B models by code clone similarity scores.

Figure 3 shows a more detailed LLaMA3-inferred similarity score over syntactic similarity scores for non-fine-tuned and fine-tuned LLaMA3 8b models. LLaMA3-inferred similarity scores range from 0 to 10, representing low to high levels of similarity, while the syntactic similarity scores range from 0.0 to 1.0, reflecting the degree of syntactic similarity between code clones. In the left-upper panel, which illustrates the performance of the non-fine-tuned LLaMA3 on the BigCloneBench dataset, a clear trend is observed: as syntactic similarity decreases, LLaMA3 also assigns lower similarity scores. In contrast, the right-upper panel, depicting the fine-tuned LLaMA3 on the same dataset, shows that fine-tuning leads to significantly higher inferred similarity scores, indicating that the fine-tuning process enhances LLaMA3's ability to extract semantic information from code pairs. For the LLM-generated code in the GPTCloneBench dataset, represented in the lower panels, LLaMA3 consistently assigns high similarity scores across the entire range of syntactic similarity values. This suggests that the model is capable of detecting functional and semantic similarities, even when syntactic similarities are minimal. This effect is even more enhanced in the fine-tuned LLaMA3, highlighting its improved ability to recognize deeper relationships in LLM-generated code after fine-tuning. These results emphasize the model's strength in identifying semantic similarities, particularly in LLM-generated code.

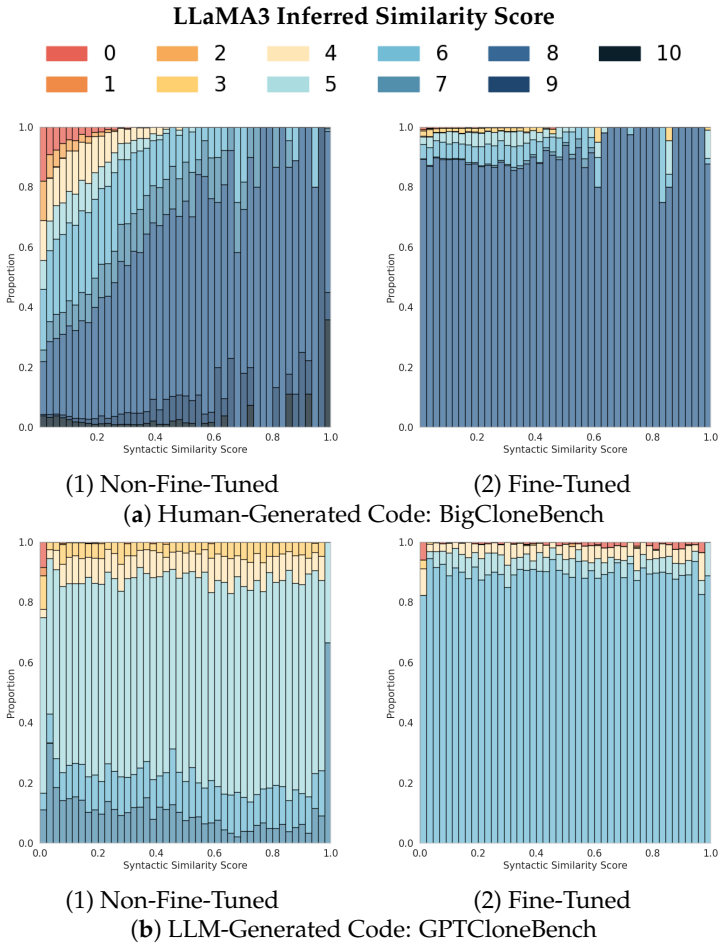


Figure 3. Comparison of syntactic similarity and LLaMA3 8B inferred similarity scores.

Figure 4 illustrates the distribution of LLaMA3-inferred code similarity scores across various clone types in the BigCloneBench dataset. From left to right, up and down, they are distributions of non-clone, type-1, type-2, strongly type-3, moderately type-3, and type-4 code pairs for non-fine-tuned and fine-tuned LLaMA3-8b model. The blue and red colors indicate whether LLaMA3 identifies these as code or not. The non-fine-tuned LLaMA3

model consistently demonstrates high accuracy in identifying true clones across clone types from T1 to MT3, with similarity scores predominantly clustered between 7 and 10. The fine-tuned LLaMA3 model follows a similar trend but with a higher concentration of similarity scores closer to 10, indicating that fine-tuning improves the model's confidence inferring similarity for T1 to MT3 code clones. In contrast, both the Non-Clone and Type-4 clone categories in the non-fine-tuned LLaMA3 model exhibit a broader distribution of similarity scores, ranging from low to high. This pattern suggests that the model struggles to accurately distinguish non-clones from Type-4 clones at a semantic level, as it assigns a wide range of similarity scores to both categories. Furthermore, the majority of code pairs in these categories are incorrectly identified as non-clones, highlighting the model's difficulty in detecting deeper semantic similarities. However, the fine-tuned LLaMA3 model shows a marked improvement in distinguishing between non-clones and Type-4 clones. In the fine-tuned model, a substantial number of non-clone code pairs are correctly assigned a similarity score of 0, while most Type-4 code clones are assigned a similarity score of 10. This clear distinction demonstrates that the fine-tuning process significantly enhances LLaMA3's ability to differentiate between non-clone code pairs and semantically similar code clones, improving the model's performance in code similarity detection.

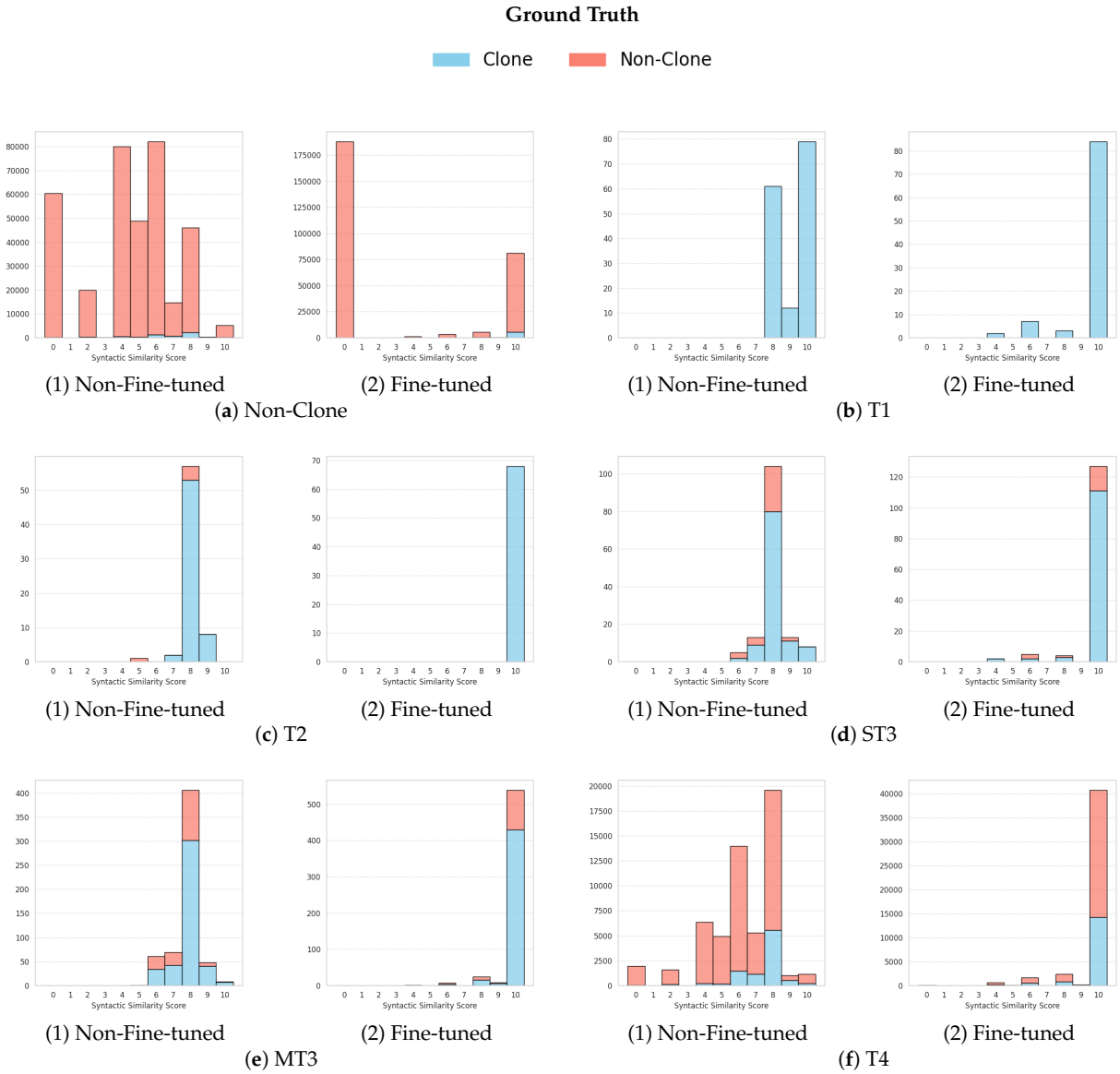


Figure 4. Distribution of code similarity scores inferred by LLaMA3 8B Models for various clone types in BigCloneBench.

Key Findings

- The LLaMA3 model consistently aligns similarity with syntactic measures in BigCloneBench, with a general decrease from T1 to T4 clones.
- In GPTCloneBench, similarity scores for all clone types are narrowly clustered, suggesting high semantic or functional similarity inferred by LLaMA across LLM-generated clones.
- Challenges arise with complex clones(T4), where LLaMA shows a similar distribution of scores with non-clone pairs and struggles to accurately identify semantic-level clones.
- The fine-tuning technique can significantly enhance the model's ability to infer semantic relationships between code pairs, particularly from the perspective of code similarity.

5. Discussion

In this section, we briefly discuss the threats to the validity of our work and present some potential future work.

5.1. Threats to Validity and Limitations

① **Limited Model Variants:** The study employed only two variants of the LLaMA3 models, 8B and 70B, which may not fully represent the broader range of available LLMs. This limitation could impact the generalizability of the findings, as other model sizes or architectures might exhibit different behaviors or performance characteristics.

② **Dataset Limitations:** The dataset selection also poses notable limitations. Existing datasets in code clone detection are limited in scope, and while BigCloneBench is the most comprehensive dataset available for this task, it may not fully represent all aspects of clone detection, especially in terms of diversity across languages and clone types. Similarly, GPTCloneBench is the only LLM-generated code clone dataset currently available, making it indispensable for this study. However, its relatively small size and our focus on the Java programming language constrain the generalizability of the findings. In future studies, the inclusion of more diverse datasets could facilitate a more comprehensive analysis. For instance, extending the study to additional programming languages and cross-language code clone detection would provide greater insights into the performance of LLMs in diverse and practical scenarios.

The dataset selection also poses notable limitations. While BigCloneBench is comprehensive and widely recognized, it has an imbalanced distribution of clone pairs across different clone types, which may bias model training and evaluation. Similarly, GPTCloneBench, though relevant for evaluating LLM-generated code clones, is relatively small in scale and primarily focused on the Java programming language. This limited scope restricts the analysis of potential biases and generalizations across diverse programming languages and larger AI-generated codebases.

③ **Lack of In-Context Learning Techniques:** Due to resource constraints and time limitations, the study did not incorporate advanced in-context learning methods, such as few-shot learning or chain-of-thought prompting.

④ **Limited Fine-Tuning Process:** The fine-tuning process was conducted over only five epochs, which, while offering some insights into model performance improvements, may not have been sufficient for the model to fully adapt to the dataset or optimize its learning process. Moreover, different fine-tuning strategies, such as curriculum learning, and multi-stage fine-tuning, could potentially improve the performance on complex clones by addressing overfitting or underfitting.

5.2. Future Work

For future work, several steps are planned to expand and enhance the study. First, we will extend the scale of experiments by incorporating a broader range of model variants, programming languages, and in-context learning techniques, such as few-shot prompting and chain-of-thought reasoning. This will provide a more comprehensive evaluation of the LLMs' performance across diverse contexts. Next, we will focus on improving LLM methods by integrating various code representations and embedding models. This will include exploring alternative code representations, such as abstract syntax trees (ASTs), control flow graphs (CFGs), and token-based approaches, to assess their influence on models' ability to detect code clones. Additionally, we will evaluate the impact of different embedding models, particularly those pre-trained on code-specific tasks, to further refine the model's performance in identifying both syntactic and semantic code similarities.

6. Conclusions

This study provides a comprehensive analysis of the capabilities and limitations of Large Language Models (LLMs) in the context of code clone detection, focusing on the differences between human-generated and LLM-generated code. Our findings reveal that while LLMs excel in identifying syntactic clones, such as Type-1 and Type-2 clones, they face significant challenges when detecting semantic clones, particularly Type-3 and Type-4. Fine-tuning techniques were found to substantially enhance the model's ability to infer semantic relationships, improving performance in both code clone detection and similarity assessment. However, the fine-tuning process also introduced biases, as LLMs demonstrated a stronger capability to detect LLM-generated clones compared to human-generated ones. This highlights the need for further research into mitigating such biases. Additionally, our analysis of the relationship between code clone detection and similarity inference revealed that LLMs effectively align with syntactic measures for simpler clones but tend to infer higher similarity for semantic clones, indicating their ability to capture functional similarities. These results emphasize both the potential and limitations of LLMs in understanding complex code semantics. Future work should aim to address these challenges by exploring more diverse datasets, enhancing model architectures, and investigating strategies to reduce biases, thereby advancing the role of LLMs in software engineering. Practical applications of these findings include enhancing code refactoring, vulnerability detection, and cross-language code translation, making LLMs a valuable tool in modern software development workflows.

Author Contributions: Conceptualization, Z.Z. and T.S.; Methodology, Z.Z. and T.S.; Software, Z.Z. and T.S.; Validation, Z.Z. and T.S.; Writing—original draft, Z.Z.; Writing—review & editing, T.S.; Visualization, Z.Z. and T.S.; Supervision, T.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by Science Foundation Ireland under grant numbers 18/CRT/6223 (SFI Centre for Research Training in Artificial Intelligence), and 13/RC/2094/P_2 (Lero SFI Centre for Software). For the purpose of Open Access, the author has applied a CC BY public copyright license to any Author Accepted Manuscript version arising from this submission.

Data Availability Statement: The data supporting the reported results are available at https://github.com/ZixianReid/LLM_Code_Clone_Validation (accessed on 10 February 2025).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Zakeri-Nasrabadi, M.; Parsa, S.; Ramezani, M.; Roy, C.; Ekhtiarzadeh, M. A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges. *J. Syst. Softw.* **2023**, *204*, 111796. [CrossRef]
2. Ehsan, O.; Khomh, F.; Zou, Y.; Qiu, D. Ranking code clones to support maintenance activities. *Empir. Softw. Eng.* **2023**, *28*, 70. [CrossRef]
3. Islam, J.F.; Mondal, M.; Roy, C.K. Bug replication in code clones: An empirical study. In Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Osaka, Japan, 14–18 March 2016; IEEE: Piscataway, NJ, USA, 2016; Volume 1, pp. 68–78.
4. Kamiya, T.; Kusumoto, S.; Inoue, K. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.* **2002**, *28*, 654–670. [CrossRef]
5. Sajnani, H.; Saini, V.; Svajlenko, J.; Roy, C.K.; Lopes, C.V. Sourcerercc: Scaling code clone detection to big-code. In Proceedings of the 38th International Conference on Software Engineering, Austin, TX, USA, 14–22 May 2016; pp. 1157–1168.
6. Roy, C.K.; Cordy, J.R. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In Proceedings of the 2008 16th IEEE International Conference On Program Comprehension, Amsterdam, The Netherlands, 10–13 June 2008; IEEE: Piscataway, NJ, USA, 2008; pp. 172–181.
7. Zhang, J.; Wang, X.; Zhang, H.; Sun, H.; Wang, K.; Liu, X. A novel neural source code representation based on abstract syntax tree. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 783–794.
8. Hua, W.; Sui, Y.; Wan, Y.; Liu, G.; Xu, G. FCCA: Hybrid code representation for functional clone detection using attention networks. *IEEE Trans. Reliab.* **2020**, *70*, 304–318. [CrossRef]
9. Zeng, J.; Ben, K.; Li, X.; Zhang, X. Fast code clone detection based on weighted recursive autoencoders. *IEEE Access* **2019**, *7*, 125062–125078. [CrossRef]
10. Li, Y.; Yu, C.; Cui, Y. TPCaps: A framework for code clone detection and localization based on improved CapsNet. *Appl. Intell.* **2023**, *53*, 16594–16605. [CrossRef]
11. Zhang, A.; Liu, K.; Fang, L.; Liu, Q.; Yun, X.; Ji, S. Learn to align: A code alignment network for code clone detection. In Proceedings of the 2021 28th Asia-Pacific Software Engineering Conference (APSEC), Taipei, Taiwan, 6–9 December 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 1–11.
12. Yang, H.; Li, Z.; Guo, X. A Novel Source Code Clone Detection Method Based on Dual-GCN and IVHFS. *Electronics* **2023**, *12*, 1315. [CrossRef]
13. Lu, Z.; Li, R.; Hu, H.; Zhou, W.a. A code clone detection algorithm based on graph convolution network with AST tree edge. In Proceedings of the 2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C), Hainan, China, 6–10 December 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 1027–1032.
14. Yuan, Y.; Kong, W.; Hou, G.; Hu, Y.; Watanabe, M.; Fukuda, A. From local to global semantic clone detection. In Proceedings of the 2019 6th International Conference on Dependable Systems and Their Applications (DSA), Harbin, China, 3–6 January 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 13–24.
15. Dou, S.; Shan, J.; Jia, H.; Deng, W.; Xi, Z.; He, W.; Wu, Y.; Gui, T.; Liu, Y.; Huang, X. Towards understanding the capability of large language models on code clone detection: A survey. *arXiv* **2023**, arXiv:2308.01191.
16. Moumoula, M.B.; Kabore, A.K.; Klein, J.; Bissyande, T. Large Language Models for cross-language code clone detection. *arXiv* **2024**, arXiv:2408.04430.
17. Inoue, R.; Higo, Y. Improving Accuracy of LLM-based Code Clone Detection U sing Functionally Equivalent Methods. In Proceedings of the 2024 IEEE/ACIS 22nd International Conference on Software Engineering Research, Management and Applications (SERA), Honolulu, HI, USA, 30 May–1 June 2024; IEEE: Piscataway, NJ, USA, 2024; pp. 24–27.
18. Khajezade, M.; Wu, J.J.; Fard, F.H.; Rodríguez-Pérez, G.; Shehata, M.S. Investigating the Efficacy of Large Language Models for Code Clone Detection. In Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension, Lisbon, Portugal, 15–16 April 2024; pp. 161–165.
19. Bellon, S.; Koschke, R.; Antoniol, G.; Krinke, J.; Merlo, E. Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng.* **2007**, *33*, 577–591. [CrossRef]
20. Svajlenko, J.; Roy, C.K. Bigcloneeval: A clone detection tool evaluation framework with bigclonebench. In Proceedings of the 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), Raleigh, NC, USA, 2–7 October 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 596–600.
21. Svajlenko, J.; Islam, J.F.; Keivanloo, I.; Roy, C.K.; Mia, M.M. Towards a big data curated benchmark of inter-project code clones. In Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, 28 September–3 October 2014; IEEE: Piscataway, NJ, USA, 2014; pp. 476–480.

22. Al-Omari, F.; Roy, C.K.; Chen, T. Semanticclonebench: A semantic code clone benchmark using crowd-source knowledge. In Proceedings of the 2020 IEEE 14th International Workshop on Software Clones (IWSC), London, ON, Canada, 18 February 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 57–63.
23. Zhang, Z.; Saber, T. Assessing the Code Clone Detection Capability of Large Language Models. In Proceedings of the 2024 4th International Conference on Code Quality (ICCQ), Kazan, Russia, 22 June 2024; IEEE: Piscataway, NJ, USA, 2024; pp. 75–83.
24. Bai, J.; Bai, S.; Chu, Y.; Cui, Z.; Dang, K.; Deng, X.; Fan, Y.; Ge, W.; Han, Y.; Huang, F.; et al. Qwen technical report. *arXiv* **2023**, arXiv:2309.16609.
25. Touvron, H.; Lavril, T.; Izacard, G.; Martinet, X.; Lachaux, M.A.; Lacroix, T.; Rozière, B.; Goyal, N.; Hambro, E.; Azhar, F.; et al. Llama: Open and efficient foundation language models. *arXiv* **2023**, arXiv:2302.13971.
26. Abdin, M.; Jacobs, S.A.; Awan, A.A.; Aneja, J.; Awadallah, A.; Awadalla, H.; Bach, N.; Bahree, A.; Bakhtiari, A.; Behl, H.; et al. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv* **2024**, arXiv:2404.14219.
27. Liu, P.; Yuan, W.; Fu, J.; Jiang, Z.; Hayashi, H.; Neubig, G. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Comput. Surv.* **2023**, *55*, 1–35. [CrossRef]
28. Dettmers, T.; Pagnoni, A.; Holtzman, A.; Zettlemoyer, L. Qlora: Efficient finetuning of quantized llms. *Adv. Neural Inf. Process. Syst.* **2024**, *36*.
29. Hu, E.J.; Shen, Y.; Wallis, P.; Allen-Zhu, Z.; Li, Y.; Wang, S.; Wang, L.; Chen, W. Lora: Low-rank adaptation of large language models. *arXiv* **2021**, arXiv:2106.09685.
30. Rajbhandari, S.; Rasley, J.; Ruwase, O.; He, Y. Zero: Memory optimizations toward training trillion parameter models. In Proceedings of the SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, Atlanta, GA, USA, 9–10 November 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 1–16.
31. Sheneamer, A.; Roy, S.; Kalita, J. An effective semantic code clone detection framework using pairwise feature fusion. *IEEE Access* **2021**, *9*, 84828–84844. [CrossRef]
32. Sun, X.; Liu, C.; Dong, W.; Liu, T. Improvements to code2vec: Generating path vectors using RNN. *Comput. Secur.* **2023**, *132*, 103322. [CrossRef]
33. Dai, L. A study on the application of graph neural network in code clone detection: Improving the performance of code clone detection through graph neural networks and attention mechanisms. In Proceedings of the CNCIT, Qinghai, China, 16–18 June 2023; pp. 172–176.
34. Zhang, L.; Luo, S.; Pan, L.; Wu, Z.; Gong, K. FSD-CLCD: Functional semantic distillation graph learning for cross-language code clone detection. *Eng. Appl. Artif. Intell.* **2024**, *133*, 108199. [CrossRef]
35. Sharma, R.; Chen, F.; Fard, F.; Lo, D. An exploratory study on code attention in BERT. In Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, Pittsburgh, PA, USA, 16–17 May 2022; pp. 437–448.
36. Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; LIU, S.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; et al. GraphCode{BERT}: Pre-training Code Representations with Data Flow. In Proceedings of the International Conference on Learning Representations, Vienna, Austria, 4 May 2021.
37. Guo, D.; Lu, S.; Duan, N.; Wang, Y.; Zhou, M.; Yin, J. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*; Muresan, S., Nakov, P., Villavicencio, A., Eds.; Association for Computational Linguistics: Dublin, Ireland, 2022; pp. 7212–7225. [CrossRef]
38. Alam, A.I.; Roy, P.R.; Al-Omari, F.; Roy, C.K.; Roy, B.; Schneider, K.A. GPTCloneBench: A comprehensive benchmark of semantic clones and cross-language clones using GPT-3 model and SemanticCloneBench. In Proceedings of the 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME), Bogota, Colombia, 1–6 October 2023; IEEE: Piscataway, NJ, USA, 2023; pp. 1–13.
39. Wei, H.; Li, M. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In Proceedings of the IJCAI, Melbourne, Australia, 19–25 August 2017; pp. 3034–3040.
40. Wang, P.; Zhu, L.; Wang, Q.; Jaiteh, O.; Guo, C. An Empirical Understanding of Code Clone Detection by ChatGPT. In Proceedings of the 2023 6th International Conference on Data Science and Information Technology (DSIT), Shanghai, China, 28–30 July 2023; IEEE: Piscataway, NJ, USA, 2023; pp. 78–83.
41. Jiang, L.; Misherghi, G.; Su, Z.; Glondu, S. Deckard: Scalable and accurate tree-based detection of code clones. In Proceedings of the 29th International Conference on Software Engineering (ICSE'07), Minneapolis, MN, USA, 20–26 May 2007; IEEE: Piscataway, NJ, USA, 2007; pp. 96–105.
42. Zhang, A.; Fang, L.; Ge, C.; Li, P.; Liu, Z. Efficient transformer with code token learner for code clone detection. *J. Syst. Softw.* **2023**, *197*, 111557. [CrossRef]
43. Wang, W.; Li, G.; Ma, B.; Xia, X.; Jin, Z. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), London, ON, Canada, 18–21 February 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 261–271.

44. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. Codebert: A pre-trained model for programming and natural languages. *arXiv* **2020**, arXiv:2002.08155.
45. Eggert, P.; Haertel, M.; Hayes, D.; Stallman, R.; Tower, L. Diffutils-gnu project-free software foundation. 2015.
46. Hugging Face. Inference Pro: Fast, Reliable, and Cost-Effective Machine Learning Inference. 2024. Available online: https://huggingface.co/blog/inference-pro (accessed on 19 July 2024).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.