Prompt-Driven Large Language Models for Automated Code Clone Detection: An Empirical Study

Shihan Dou, Chaofan Li, Junjie Shan, Haoxiang Jia, Yutao Hu, Yueming Wu, Xuanjing Huang

***Abstract*—Code cloning, which involves duplicating code fragments, is a common practice in software development. While some code reuse is beneficial for productivity, excessive cloning can negatively impact maintainability and spread bugs. Thus, automatic clone detection is crucial for software engineering. The pre-trained corpus of large language models (LLMs) contains a vast amount of code-related tasks, making them capable of tackling a wide range of challenges related to code in software engineering. However, LLMs' performance in code clone detection is still not fully understood, requiring further research to assess their accuracy and effectiveness in this area. In our paper, we present the first comprehensive evaluation of the application of LLMs in automated code clone detection. The evaluation dimensions cover simple prompts, informative prompts, the cost-benefit balance of LLMs, LLM-based code embedding, applicability in practical scenarios, and performance across programming languages, while also considering detection variations across different clone types. The core research findings indicate that LLMs have strong natural language understanding and code semantic analysis capabilities, which allow them to excel at detecting complex semantic clones (Type-3 and Type-4). Informative prompts improve the detection performance of most models, but their effectiveness depends on the model. LLM-based code embedding schemes perform better than dedicated pre-trained code embedding models. Open-source and commercial models exhibit differences in cross-language detection performance, with Python achieving the best overall results. Overall, this study demonstrates that LLMs, supported by their strong natural language and code semantic understanding capabilities, have substantial application value in code clone detection. The findings provide a solid foundation for refining prompt strategies, enhancing complex clone detection performance, and developing practical LLM-based code clone detection solutions.**

***Index Terms*—Code Clone Detection, Large Language Model, Study**

I. Introduction

CODE cloning, the replication of code fragments, is a common phenomenon in software development. While some code reuse aids productivity, excessive cloning negatively impacts maintainability and propagates bugs [1], [2]. Thus, automatic clone detection is an important research area.

Shihan Dou and Xuanjing Huang are with FudanNLP Lab, School of Computer Science and Technology, Fudan University, Shanghai, 200082, China (e-mails: shdou21@m.fudan.edu.cn, xjhuang@fudan.edu.cn).

Chaofan Li, Junjie Shan, Haoxiang Jia, Yueming Wu, and Yutao Hu are with Huazhong University of Science and Technology, Wuhan , 430000, China (e-mails: junjieshan@hust.edu.cn, haoxiangjia@hust.edu.cn yuemingwu@hust.edu.cn, chaofanli1@hust.edu.cn, yutaohu@hust.edu.cn).

Shihan Dou and Chaofan Li contributted equally, Yutao Hu is the corresponding author.

To better comprehend clone detection, researchers have undertaken a methodical classification of code clones into distinct categories. A widely accepted taxonomy segregates code clones into four types: Type-1 (*Identical Similarity*), Type-2 (*Lexical Similarity*), Type-3 (*Syntactical Similarity*), and Type-4 (*Semantic Similarity*) [3], [4]. The first three types can generally be encapsulated under the umbrella of syntactic similarities, while the fourth type epitomizes semantic similarities. Given that Type-4 clones may comprise clones that display a wide range of syntactic dissimilarities, they present the most formidable challenge for most clone detection methodologies.

There exists extensive literature focusing on code syntactic similarities [5], [6], [7]. However, in recent years, attention has gradually shifted toward the study of code semantic similarities. This shift has been facilitated by advancements in the field of deep neural networks. As a result, a plethora of deep learning-based methodologies have been proposed, all designed to discern semantic similarities through a process of data-driven learning [8]. These methodologies largely adopt a two-pronged approach: firstly, neural networks are leveraged to generate a vector representation for each code fragment, which is then followed by calculating the similarities between the representations of two code fragments to detect clones [9].

As a matter of fact, the development of *pre-trained language models* (PLMs) has revolutionized the area of deep learning. These models, such as *BERT* [10] and *GPT-1* [11], are pre-trained with specially designed pre-training tasks on large-scale unlabeled text corpora to learn generalized knowledge. After that, many works such as *CodeBERT* [12] and *CodeT5+* [13] introduce pre-training to further boost code-related tasks in software engineering. Although these works have a great performance, they still need to be fine-tuned to adapt to different downstream tasks [14], [15].

Recently, researchers have observed that scaling *pre-trained language models* (PLMs), such as increasing model size or training data, often leads to substantially improved capacity on downstream tasks [16]. Although most scaling efforts focus on enlarging model size while preserving similar architectures and pre-training objectives, large-scale PLMs (*e.g., GPT-3* [17], *MPT* [18], and *LLaMA* [19]) exhibit behaviors that differ markedly from those of smaller models (*e.g.,* 330M-parameter *BERT* and 1.5B-parameter *GPT-2* [20]). In particular, these models demonstrate surprising emergent abilities and are able to solve a wide range of complex tasks using only natural language instructions without task-specific fine-tuning [21].

Moreover, as the pre-training corpora of *large language models* (LLMs) contain substantial amounts of source code and code-related data, LLMs have shown strong potential in addressing various software engineering tasks. For instance, Feng et al. [22] proposed an automatic technique to replay bugs from natural language bug reports through prompt engineering, while Deng et al. [23] developed a testing framework that leverages generative and infilling LLMs to synthesize and mutate programs for deep learning library testing. However, despite these advances, the effectiveness and limitations of LLMs in the specific task of code clone detection remain largely unexplored.

In this paper, we investigate the feasibility of leveraging LLMs for code clone detection. Our study is motivated by the strong capability of LLMs to comprehend complex semantic inputs and produce meaningful representations. We argue that such capabilities can be exploited to identify and classify code clones, offering a new perspective on this long-standing problem. Specifically, we conduct a comprehensive empirical study to evaluate the clone detection performance of a diverse set of LLMs, including *StarChat2* [24], *Qwen3-Coder* [25], *DeepSeek-Coder* [26], *DeepSeek-V3* [27], *DeepSeek-R1* [28], *GPT-3.5* [29], *GPT-4.1* [30], *Gemini-2.5-Flash* [31], and *Claude-Sonnet-4.5* [32].

Our study focuses on the following research questions:

- *RQ1*: Can LLMs outperform traditional detection approaches in code clone detection?
- *RQ2*: Can *informative prompts* improve the code clone detection performance of LLMs?
  - *RQ2.1*: Which prompt components are most effective in improving LLM-based code clone detection performance?
  - *RQ2.2*: Do *informative prompts* outperform *simple prompts* in LLM-based code clone detection?
- *RQ3*: How can an effective balance between the costs and benefits of using LLMs be achieved for code clone detection?
- *RQ4*: Can embeddings generated by LLMs outperform those produced by pre-trained code embedding models in code clone detection?
- *RQ5*: How does the performance of LLMs in code clone detection vary across different programming languages?

Regarding **RQ1**, our results show that when using *simple prompts*, LLMs outperform traditional detection approaches in detecting Type-3 and Type-4 clone pairs, while exhibiting slightly inferior performance on Type-1 and Type-2 clones. These findings indicate that LLMs and traditional detection approaches demonstrate different strengths across clone types, rather than one consistently dominating the other.

Regarding **RQ2**, our observations indicate that *informative prompts* can substantially improve the code clone detection performance of most models, especially for clone types that initially exhibit weaker results. Among all evaluated models, *Gemini-2.5-Flash* shows the most notable performance gains. However, we also find that *informative prompts* may lead to performance degradation for certain models that already achieve strong baseline performance, highlighting the need for prompt design to be adapted to model characteristics.

Regarding **RQ3**, we observe that *informative prompts* generally increase the resource consumption of LLMs in code clone detection tasks, although the extent of this increase varies considerably across models due to differences in their inference and output behaviors. Furthermore, our results indicate that there is no inherent positive correlation between model cost and detection effectiveness. This finding provides a practical basis for designing cost effectiveness tradeoff strategies when deploying LLMs for clone detection.

Regarding **RQ4**, we find that embeddings generated by *text-embedding-ada-002* outperform those produced by specialized pre-trained code embedding models in identifying code clones and achieve superior overall performance. This advantage can be attributed to its ability to produce a broader distribution of similarity scores, which enables more effective separation of true positives from false positives. These results offer useful guidance for selecting embedding models in code clone detection tasks.

Regarding **RQ5**, we observe that open-source LLMs achieve strong code clone detection performance mainly for Python, likely due to its simpler syntax and higher representation in training data. In contrast, commercial LLMs perform consistently well across multiple programming languages, benefiting from larger model scales and stronger semantic understanding. These results highlight the impact of model characteristics on cross-language clone detection performance.

In summary, our paper makes the following contributions:

- We conduct a comprehensive empirical study to evaluate the capabilities of existing LLMs for code clone detection from six distinct perspectives, including simple prompts, informative prompts, cost-benefit tradeoffs, LLM-based code embeddings, and cross-language performance.
- We open source all datasets and implementation code used in this study and provide systematic insights into the strengths and limitations of LLMs for code clone detection. Our findings offer practical guidance for future research on improving LLM-based clone detection as well as other software engineering tasks.

Paper Organization. The remainder of the paper is organized as follows. Section 2 explains the background. Section 3 introduces our experimental setup. Section 4 reports the experimental results. Section 5 discusses future work. Section 6 concludes the present paper.

II. Background and Related Work

A. Code Clone Definition

Code clones are commonly categorized into four types based on their syntactic and semantic similarity:

- **Type-1 clones (T1, Textual Similarity):** Code fragments that are identical except for differences in whitespace, layout, and comments.
- **Type-2 clones (T2, Lexical Similarity):** Code fragments that are structurally identical but allow variations in identifiers such as variable names, function names, and class names.
- **Type-3 clones (T3, Syntactic Similarity):** Code fragments that are syntactically similar but differ at the statement level, where statements may be added, modified, or removed.

Following the BigCloneBench dataset, T3 clones are further divided into *Very Strongly Type-3* (VST3, $\geq$ 90% similarity), *Strongly Type-3* (ST3, 70%–90% similarity), *and Moderately Type-3* (MT3, 50%–70% similarity), based on line-level and token-level similarity after T1 and T2 normalization. Due to data distribution limitations and following common practice in prior studies [33], [34], [35], we merge ST3 and VST3 clones into a single category, referred to as ST3, in our experiments.

- **Type-4 clones (T4, Semantic Similarity):** Code fragments that are syntactically dissimilar but implement the same functionality.

B. Code Clone Detection

A variety of methods have been proposed for code clone detection, which can be broadly categorized into text-based, token-based, tree-based, and graph-based approaches.

Token-based methods [[5], [36], [5], [37], [38], [7] extract token sequences through lexical analysis and detect clones by analyzing repeated subsequences. For example, *SourcererCC* [5] identifies clones based on token overlap similarity with a predefined threshold, offering simplicity and scalability. *NIL* [7] uses N-gram sequences and an inverted index to locate candidate clones, which are then verified using the longest common subsequence. These methods are effective for textually similar clones but are generally limited in detecting Type-4 semantic clones.

Tree-based methods [39], [40], [5], [37], [38], [41], parse code into tree structures and detect clones through tree matching. *Deckard* [39] clusters AST vectors to enable multi-language clone detection. Liang and Ai [40] introduce tree-path analysis, learning vector representations via a compare-aggregate model to determine clone similarity. Compared to token-based methods, tree-based approaches better capture structural and some semantic information.

Graph-based methods [42], [43], [44], [9], [43] construct program graphs for clone detection. Traditional subgraph matching approaches are computationally expensive. *CC-Graph* [43] reduces the size of program dependence graphs (PDG) and applies a two-stage filtering process to improve efficiency. *SCDetector* [9] treats program graphs as social networks to extract semantic tokens, enabling Type-4 clone detection. Hu et al. [44] leverage FC-PDG and relational graph convolutional networks (R-GCN) to capture contextual information. Overall, graph-based methods can comprehensively capture both structural and semantic information, offering clear advantages for detecting semantic and complex clones.

Recent work, such as [45], explores code clone detection using LLMs but has several limitations compared to our study. First, [45] uses a non-mainstream dataset, whereas we utilize BigCloneBench [46], a widely recognized benchmark in code clone detection. Additionally, while [45] focuses solely on a single decoder-only model, our study investigates both encoder-only and decoder-only LLMs, including the latest state-of-the-art model, *text-embedding-ada-002*. Our work also expands on prompt diversity and explores the effects of various CoT and scoring strategies, in contrast to the single prompt design in [45]. Moreover, we include comparisons with traditional code clone detection methods, providing a comprehensive performance evaluation. Finally, as our study predates [45], our work serves as the first and broader investigation into the potential of LLMs for code clone detection.

C. Large Language Models

Recent advancements in LLMs have significantly advanced *natural language processing* (NLP). In general, an LLM is a Transformer-based model with hundreds of billions of parameters, such as *Starchat2* [24], *DeepSeek-V3* [27], *Deepseek-R1* [28], *GPT-3.5* [29], and *GPT-4.1* [30]. Trained on massive text corpora, these models exhibit strong generalization ability across a wide range of natural and programming language tasks. Traditionally, language sequence tasks have relied on task-specific fine-tuning to achieve satisfactory performance [14]. Fine-tuning updates model parameters using labeled data from a downstream task [15], often requiring substantial task-specific datasets. In contrast, in-context learning enables LLMs to perform downstream tasks by following instructions and examples provided at inference time, without modifying model parameters [47], [21], [48]. In this study, we design diverse instruction prompts to guide LLMs in understanding code clone detection from multiple perspectives, enabling a systematic evaluation of their effectiveness on this task.

In this study, we design diverse instruction prompts to guide LLMs in understanding clone detection from multiple perspectives, enabling a systematic evaluation of their effectiveness on this task. Moreover, this paradigm allows us to investigate LLM behavior under different prompt configurations without introducing additional training data or model-specific tuning. Such flexibility is particularly important for empirical studies that aim to assess the general applicability and robustness of LLMs across different code analysis scenarios.

D. Prompting as Task Specification for Code Clone Detection

In LLM-based systems, prompting has emerged as a central mechanism for specifying task intent and constraining model behavior without explicit model retraining [17], [49]. Rather than serving as mere input formatting, prompts effectively function as a task interface that guides how models interpret inputs, perform reasoning, and generate outputs [50]. Recent studies in software engineering have also shown that prompt design can substantially influence LLM performance on code understanding tasks, such as program analysis, defect detection, and code summarization [51], [52].

Among various prompting strategies, reasoning-guided prompts, particularly *chain-of-thought* (CoT) prompting, have been shown to improve performance on tasks that require multi-step reasoning by encouraging models to explicitly articulate intermediate reasoning processes [49]. Code clone detection, especially for Type-3 and Type-4 clones, often involves semantic equivalence analysis beyond surface-level syntactic similarity, making it a natural candidate for reasoning-guided prompting. In this work, we treat CoT not as an independent technique, but as a form of reasoning guidance embedded within the task prompt, and empirically examine its effectiveness and limitations in LLM-based code clone detection in Section IV.

III. Experimental Setup

A. Research Questions

Our empirical study delved into five research questions to improve the understanding of code clone detection using LLMs.

- **RQ1: Can LLMs outperform traditional detection approaches in code clone detection?** This research question investigates whether LLMs can achieve superior performance compared to traditional detection approaches in code clone detection. To this end, we conduct an evaluation on the Java dataset covering five clone types (*i.e.,* Type-1, Type-2, *Strong Type-3* (ST3), *Moderate Type-3* (MT3), Type-4), involving nine LLM-based approaches and nine traditional detection approaches. For LLM-based detection, we adopt the *simple prompt* shown in Table I, which requires models to directly output a binary judgment of "yes" or "no".
- **RQ2: Can *informative prompts* improve the code clone detection performance of LLMs?** This research question examines whether *informative prompts*, defined as refined and information-rich prompts, can enhance the code clone detection performance of LLMs. We design such prompts along three dimensions, namely the *system prompt*, *task prompt*, and *few-shot examples*. The prompt templates are presented in Table I. Experiments are conducted to analyze the effectiveness of individual prompt components and to compare *informative prompts* with *simple prompts*.
  - ◦ **RQ2.1: Which prompt components are most effective in improving LLM-based code clone detection performance?** This sub-question focuses on identifying which prompt components contribute most to improving LLM-based code clone detection. We conduct ablation experiments by constructing different combinations of prompt components and comparing model performance under each setting, thereby clarifying the contribution of each component and identifying the most influential ones.
  - ◦ **RQ2.2: Do *informative prompts* outperform *simple prompts* in LLM-based code clone detection?** This sub-question evaluates whether *informative prompts* can consistently outperform *simple prompts* in code clone detection. Based on the optimal component combination identified in RQ2.1, we conduct a comprehensive evaluation across all models to analyze performance improvements and examine their applicability boundaries.
- **RQ3: How can an effective balance between the costs and benefits of using LLMs be achieved for code clone detection?** This research question explores the cost-benefit tradeoffs of employing LLMs for code clone detection. By conducting targeted experiments on four closed-source commercial models and two commercially deployed versions of open-source models, we analyze differences in resource consumption under two prompting strategies and examine the relationship between cost and detection performance.
- **RQ4: Can embeddings generated by LLMs outperform those produced by pre-trained code embedding models in code clone detection?** This research question compares the effectiveness of embeddings generated by LLMs with that of traditional pre-trained code embedding models for code clone detection. We select one LLM-related embedding model and four representative pre-trained code embedding models, and evaluate their clone detection capability by computing cosine similarity between code pair representations. This comparison relies on the embedding API provided by *OpenAI* [53]. As this research question focuses on existing embedding models, no additional prompt design is involved.
- **RQ5: How does the performance of LLMs in code clone detection vary across different programming languages?** This research question investigates whether the performance of LLMs in code clone detection varies across different programming languages. Using the same prompt as in RQ1 and without specifying the target programming language, we evaluate the ability of LLMs to handle code clone detection tasks across multiple languages.

B. Dataset Construction

To enable a robust and comprehensive evaluation of code clone detection tasks, we construct two datasets in this study: a Java dataset and a cross-language dataset.

It is worth noting that, to prevent potential data leakage, all samples from Google Code Jam and *CodeNet* are screened following the methodology proposed in [54]. Specifically, we analyze differences in the model's generation behaviors for the same input under different temperature settings to determine whether the model exhibits memorization effects on certain samples. Based on this analysis, samples with potential leakage risks are removed, and only leakage-free data are retained for subsequent experiments.

1) Java Dataset

The Java dataset consists of Type-1 and Type-2 clone samples from BigCloneBench (BCB) [46], along with ST3, MT3, and Type-4 clone samples from the GCJ dataset [55]. This construction is adopted to address known annotation inaccuracies in the semantic clone portion of BCB [56], [57]. Specifically, GCJ is used as a complementary source of semantic clones, as it has been widely adopted in prior studies [58] and is derived from real-world programming competition submissions featuring diverse implementation styles.

The resulting Java dataset covers five clone types: Type-1, Type-2, ST3, MT3, and Type-4. For each clone type, we include 500 clone pairs along with 2,500 non-clone pairs. Type-1 and Type-2 clones are sampled from the more reliably annotated portions of BCB, while ST3, MT3, and Type-4 clones are drawn from the GCJ dataset. It should be noted that GCJ, as a semantic clone dataset, does not provide fine-grained clone type labels for its samples. Therefore, we further annotate GCJ clone pairs by computing token-level similarity between code pairs according to the definitions of different clone types, as described in Section 2.A.

2) Cross-language Dataset

To evaluate the performance of LLMs on cross-language code clone detection, we construct a cross-language dataset

TABLE I: Prompt Design for Code Clone Detection Research Questions

Instruction Type	**Instance**
RQ1: Can LLMs outperform traditional detection approaches in code clone detection?	
Simple Prompt	Please analyze the following two code snippets and determine if they are code clones. Respond with ‘yes’ if the code snippets are clones or ‘no’ if not.
RQ2.1: Which prompt components are most effective in improving LLM-based code clone detection performance?	
System Prompt	A code clone refers to two or more identical or similar source code snippets existing in a code repository.You are a capable software development assistant specializing in code clone detection, aiming to help other developers understand the characteristics of code clones and identify clone relationships existing in code.
Task Prompt	Now, given the following two function snippets, please return the judgment result of code clone detection in JSON format. Think step by step and provide analysis around the following aspects: 1) Text similarity of code; 2) Semantic similarity of codep; 3) Syntactic similarity of code; 4) Functional similarity of code.
Few-shot Examples	Here are several sets of reference examples for code clone detection, and you may understand the reasoning behind determining clone relationships by referring to these cases. **Example 1: Result = yes** **Function 1:** `int calculate_sum(int x, int y) { int total = x + y; return total; }` **Function 2:** `int compute_sum(int a, int b) { int sum_val = a + b; return sum_val; }` **Result**: yes **Example 2: Result = no** **Function 1:** `List<Integer> filter_even(List<Integer> nums) { List<Integer> evens = new ArrayList<>(); for (int num : nums) { if (num % 2 == 0) evens.add(num); } return evens; }` **Function 2:** `List<Integer> filter_odd(List<Integer> nums) { List<Integer> odds = new ArrayList<>(); for (int num : nums) { if (num % 2 != 0) odds.add(num); } return odds; }` **Result**: no
RQ2.2: Do *informative prompts* outperform *simple prompts* in LLM-based code clone detection?	
System Prompt	A code clone refers to two or more identical or similar source code snippets existing in a code repository.You are a capable software development assistant specializing in code clone detection, aiming to help other developers understand the characteristics of code clones and identify clone relationships existing in code.
Task Prompt	Now, given the following two function snippets, please return the judgment result of code clone detection in JSON format. Think step by step and provide analysis around the following aspects: 1) Text similarity of code; 2) Semantic similarity of codep; 3) Syntactic similarity of code; 4) Functional similarity of code.
RQ5: How does the performance of LLMs in code clone detection vary across different programming languages?	
Simple Prompt	Same as RQ1.

based on the *CodeNet* dataset [59], covering three programming languages: Java, C/C++, and Python.

CodeNet is a widely adopted dataset for cross-language code clone detection [60]. For each language, we randomly select 500 clone pairs and 1,500 non-clone pairs, thereby constructing a balanced cross-language dataset to systematically assess the generalization capability of models across different programming languages.

C. Language Models

We evaluate a total of 14 language models, including nine autoregressive LLMs and five code embedding models. Among the autoregressive LLMs, two are open-source models specifically fine-tuned for code-related tasks, while the remaining seven are general-purpose LLMs, of which five are open-source and two are closed-source. The complete list of evaluated models is summarized in Table II. For code embedding models, we include four widely used pre-trained models (*CodeBERT* [12], *CodeBERT-MLM* [12], *GraphCodeBERT* [61], and *UniXcoder* [62]), along with one state-of-the-art closed-source embedding model (*text-embedding-ada-002* [63]).

1) Open-source Large Language Models

Five of the evaluated models are open-source LLMs supporting local deployment. Due to computational constraints, we evaluate the commercial API versions of *DeepSeek-V3* and *DeepSeek-R1*. All models are trained on large-scale corpora containing both natural language text and source code, with parameter sizes ranging from several billion to hundreds of billions. These models are included to leverage their large-scale learning capabilities for code clone detection tasks.

StarChat2-15B [64] is an open-source language model built upon the *StarCoder2* architecture [24], with approximately 16 billion parameters. After fine-tuning on synthetic instruction datasets, the model achieves balanced and stable performance across multiple benchmarks, including *MT-Bench*, *IFEval*, and the *HumanEval* code generation benchmark. It supports over 600 programming languages and integrates two primary capabilities: efficient code generation and natural conversational interaction. The model is accessible through the *Transformers* framework. Despite its strengths, *StarChat2-15B* may exhibit output bias in certain cases, and practical deployment requires appropriate adaptation to specific application scenarios.

Qwen3-Coder-30B-Instruct [25] is a causal language model designed for intelligent coding scenarios. It is pre-trained and post-trained on large-scale, high-quality code corpora. The model adopts a *Mixture of Experts* architecture with *Grouped Query Attention*, comprising 30.5 billion total parameters and 3.3 billion activated parameters. It supports an ultra-long context window of up to 262,144 tokens, enabling effective understanding of repository-level code structures. *Qwen3-Coder-30B-Instruct* demonstrates strong performance in multilingual code generation and complex engineering tasks, while remaining compatible with existing toolchains.

DeepSeek-Coder-7B-Instruct [26] is a large language model specialized for programming tasks. Built on the *DeepSeek-LLM 7B* base model, it is pre-trained on approximately 2 trillion tokens and further fine-tuned with 2 billion instruction tokens to better follow diverse coding instructions. As a 7B-parameter causal language model with a 4K-token context window, it supports multiple mainstream programming languages. Consequently, *DeepSeek-Coder-7B-Instruct* effectively understands developer intent and performs common coding tasks such as code generation, completion, and debugging.

DeepSeek-V3 [27] is a high-performance Mixture of Experts LLM pre-trained on 14.8 trillion high-quality tokens and further optimized through supervised fine-tuning and reinforcement learning. It incorporates architectural innovations such as Multi-head Latent Attention and *DeepSeekMoE*, and introduces an auxiliary-loss-free load-balancing strategy together with a multi-token prediction objective. With 671 billion total parameters and 37 billion parameters activated per token, *DeepSeek-V3* supports a native context window of 128K tokens. These design choices enable efficient inference and cost-effective training, allowing the model to achieve performance comparable to state-of-the-art closed-source LLMs across tasks including knowledge question answering, code development, and mathematical reasoning.

DeepSeek-R1 [28] is a first-generation reasoning-oriented LLM developed by the DeepSeek team to address limitations of pure reinforcement learning approaches in complex reasoning tasks. The model adopts a Mixture of Experts architecture with 671 billion total parameters and 37 billion activated parameters per token, and is trained through multiple stages based on the *DeepSeek-V3* foundation. Its key innovation lies in incorporating cold-start data prior to reinforcement learning, which alleviates issues such as repetitive outputs, low readability, and language mixing observed in *DeepSeek-R1-Zero*. As a result, *DeepSeek-R1* exhibits structured and human-like reasoning behavior and demonstrates strong performance in mathematics, programming, and other complex reasoning tasks. The model also supports a 128K-token context window, enabling efficient handling of multi-step reasoning scenarios.

2) Closed-source Large Language Models

We further evaluate four widely used closed-source LLMs, including *GPT-3.5-turbo* [29], *GPT-4.1* [30], *Gemini-2.5-flash* [31], and *Claude-Sonnet-4-5* [32]. These models are accessible through official APIs, with *GPT-3.5-turbo* and *GPT-4.1* provided by OpenAI, and *Gemini-2.5-Flash* and *Claude-Sonnet-4-5* offered by Google and Anthropic, respectively. As state-of-the-art models developed by their respective organizations, they demonstrate strong capabilities across a broad range of tasks, including natural language understanding, text generation, multilingual code processing, and complex reasoning. Accordingly, these closed-source LLMs serve as important benchmarks in our experimental evaluation.

3) Pre-trained Language Models for Code Embedding

Embedding is a machine learning technique that maps high-dimensional and complex data, such as text and images, into compact, low-dimensional vector representations. These representations can be directly used as features or further fine-tuned with task-specific supervision.

We evaluate four pre-trained models specifically designed for code embedding: *CodeBERT-Base* [12], *CodeBERT-MLM* [12], *GraphCodeBERT* [61], and *UniXcoder* [62].

CodeBERT-Base is trained on a mixture of natural language and source code corpora, while *CodeBERT-MLM* adopts a masked language modeling objective to enhance code understanding capabilities [65]. *GraphCodeBERT* extends *CodeBERT-Base* by incorporating syntactic structure information and structure-aware pre-training tasks, improving its effectiveness for code comprehension [61]. *UniXcoder* is trained on multilingual code and natural language data using cross-lingual and cross-modal objectives, enabling semantic alignment between code and natural language as well as across multiple programming languages [62].

In addition, we evaluate *text-embedding-ada-002* [63], a state-of-the-art embedding model capable of generating representations for both natural language and code, which has shown strong effectiveness in similarity-based tasks such as code clone detection.

Although trained with different objectives, both BERT-series models and OpenAI's *text-embedding-ada-002* provide effective solutions for code embedding. BERT-series models rely on masked language modeling to capture token-level relationships within code [12], whereas *text-embedding-ada-002* is trained with a contrastive objective to learn semantic similarities from large-scale, unsupervised data [66], [63]. Despite these differences, both approaches generate embeddings well suited for code clone detection by effectively encoding the structural and functional semantics of code snippets.

D. Traditional Detection Techniques

We also include traditional code clone detection techniques as baseline methods. These approaches are categorized into three groups based on their underlying representations: token-based, *abstract syntax tree* (AST)-based, and *control flow graph* (CFG)-based techniques. For each category, we select three representative tools.

Token-based techniques. *NiCad* [6] computes similarity by comparing the longest common subsequence (LCS) between tokenized code fragments after preprocessing. *Lvmapper* [67] similarly adopts the LCS algorithm to measure token sequence similarity and further applies heuristic optimization strategies. *SourcererCC* [5] detects code clones by constructing a global token position graph and measuring overlap similarity between code fragments.

AST-based techniques. the approach proposed by *Lazar et al.* [68] measures similarity by matching subtrees with the same number of nodes. *Zhao et al.* [69] linearize abstract syntax trees and store hash values of subtrees with identical numbers of child nodes, while assigning different weights to operands on either side of operators to improve detection accuracy. *Yang et al.* [70] propose a structural abstraction approach that replaces original AST nodes with higher-level representations based on predefined node types, followed by similarity comparison using the Smith-Waterman algorithm [71].

CFG-based techniques. *Amme et al.* [72] propose *StoneDetector*, which analyzes control flow graphs to construct dominator trees and generates fingerprints by counting paths within these trees for similarity computation. *Marastoni et al.* [73] introduce *GroupDroid*, which represents methods as feature vectors by extracting weighted centroids from their CFGs to measure similarity. *ATVHunter* [74] is a third-party library detection tool for Android applications that also supports clone detection. It first identifies libraries using coarse-grained CFGs and then employs basic blocks as fine-grained features to enable precise similarity matching.

E. Implementation

For **RQ1**, we evaluate a total of nine LLM-based models. The results show that among open-source models, *Qwen3-Coder-30B-Instruct* achieves the best performance, while *GPT-4.1* and *Claude-Sonnet-4-5* are the top-performing closed-source models. Based on these findings, we select these three models for **RQ2** to investigate the effectiveness of Informative Prompts by decomposing prompts into three dimensions: *System Prompt*, *Task Prompt*, and *Few-shot Examples*.

For **RQ3**, we examine the cost consumption of LLMs in code clone detection through targeted evaluations on four closed-source commercial models and two commercially deployed versions of open-source models.

For **RQ4**, we focus on classical pre-trained embedding models, including *CodeBERT-Base*, *CodeBERT-MLM*, *GraphCodeBERT*, *UniXcoder*, and *text-embedding-ada-002*. These models are well suited for generating embedding vectors that capture code semantics, enabling a systematic comparison between open-source and closed-source embedding approaches for code clone detection.

Finally, **RQ5** analyzes the performance of LLMs in multilingual environments, with the goal of assessing their adaptability to multilingual code clone detection tasks.

When leveraging language models for code-related tasks, practical scenarios typically prioritize response accuracy over output diversity. Accordingly, inference hyperparameters are configured differently from those commonly used in natural language generation tasks [30]. In our experiments, we followed the approach commonly used in prior studies [75], [76], [77], set the temperature [78], [79] to 0.2, Top-p (nucleus sampling [80] to 0.1, and Top-k to 10. This low-temperature and low Top-p configuration ensures stable and deterministic outputs, thereby improving the reliability of code generation and analysis tasks.

For encoder-only pre-trained language models such as *CodeBERT*, each code fragment is independently processed to obtain its embedding representation. Following our prior work [81], we use the final hidden state of the first token in the last layer as the embedding of a code fragment. The similarity between two code fragments is computed using cosine similarity. In practice, the classification threshold is determined on a training or validation set and then applied to the test set for evaluation.

F. Evaluation Metrics

We evaluate detection performance using three widely used metrics: precision, recall, and F1 score. Precision, which is calculated overall across all clone types, is defined as $P = \frac{TP}{TP+FP}$, recall as $R = \frac{TP}{TP+FN}$, and F1 score as $F1 = \frac{2 \cdot P \cdot R}{P+R}$. In this paper, precision is computed as an overall metric for

TABLE II: Comparison of SOTA Traditional Detection Approaches and LLMs-based Code Clone Detection Methods

Method	**Modality**	**Recall (%)**					**Precision (%)**
		T1	**T2**	**ST3**	**MT3**	**T4**	
Traditional Detection Approaches							
NiCad [6]	Token	100.00	100.00	97.00	0.20	0.00	100.00
SourcererCC [5]	Token	100.00	100.00	96.40	2.80	0.00	99.93
Lvmapper [67]	Token	100.00	97.00	96.60	0.60	0.00	100.00
Lazar [68]	AST	100.00	100.00	92.60	0.80	0.00	100.00
Zhao [69]	AST	100.00	100.00	97.80	16.60	0.20	99.68
Yang [70]	AST	100.00	100.00	94.40	4.20	0.00	99.73
StoneDetector [72]	CFG	98.80	92.20	91.80	30.00	8.80	89.23
GroupDroid [73]	CFG	96.60	96.00	92.20	29.20	5.00	94.94
ATVHunter [74]	CFG	96.60	95.80	93.40	10.80	0.40	99.53
LLM-Based Detection Approaches							
Deepseek-Coder-7B-Instruct [26]	Open-source / Code	71.40	9.80	27.00	4.20	4.20	73.15
Qwen3-Coder-30B-Instruct [25]	Open-source / Code	98.80	16.00	73.40	89.20	60.20	76.66
Starchat2-15B [64]	Open-source / Text	98.60	10.20	13.20	0.80	0.00	55.37
Deepseek-V3 [27]	Open-source / Text	100.00	90.40	97.20	79.80	29.80	89.78
Deepseek-R1 [28]	Open-source / Text	100.00	99.80	95.00	89.00	61.80	87.34
GPT-3.5-turbo [29]	Closed-source / Text	89.60	92.20	77.20	84.80	47.40	87.99
GPT-4.1 [30]	Closed-source / Text	100.00	97.20	94.00	95.80	77.60	86.45
Gemini-2.5-Flash [31]	Closed-source / Text	91.80	16.00	72.60	34.00	8.20	89.61
Claude-sonnet-4-5 [32]	Closed-source / Text	100.00	99.20	94.60	82.20	27.40	88.93

all clone types, while recall is calculated separately for each clone type to account for differences in detection accuracy across types. In these definitions, the term *true positive* (TP) represents the number of samples correctly classified as clone pairs, *false positive* (FP) represents the number of samples incorrectly classified as clone pairs, and *false negative* (FN) represents the number of samples incorrectly classified as non-clone pairs. This calculation approach is consistently applied in all tables in this paper.

G. Hardwares

The experiments were conducted on a server equipped with dual AMD EPYC 7742 64-Core Processors, 128 CPUs, 1TB of memory, and eight NVIDIA A800-SXM4-80GB GPUs.

IV. Experimental Result

A. RQ1: Can LLMs outperform traditional detection approaches in code clone detection?

For this research question, we investigate whether LLMs can perform code clone detection using simple prompting strategies. We evaluate nine LLM-based and nine traditional detection approaches on our constructed Java dataset, covering five clone types.

As shown in Table II, among LLM-based approaches, closed-source models exhibit a clear overall advantage in code clone detection. Their stronger capabilities in code comprehension and logical reasoning enable better adaptation to the detection requirements of different clone types. This advantage is particularly evident for low-difficulty clones. For the most basic clone category, Type-1, nearly all models achieve detection rates close to 100%, with the exception of *Deepseek-Coder-7B-Instruct*, which performs relatively poorly (71.40%). For Type-2 clones, which introduce a modest increase in detection difficulty, closed-source models (*GPT-3.5-Turbo*, *GPT-4.1*, and *Claude-Sonnet-4-5*), as well as state-of-the-art open-source models (*DeepSeek-R1* and *DeepSeek-V3*), continue to demonstrate strong performance. In contrast, smaller-scale open-source models (*Qwen3-Coder-30B-Instruct*, *DeepSeek-Coder-7B-Instruct*, and *StarChat2-15B*) experience a noticeable decline in detection accuracy. This trend confirms that clone detection difficulty increases progressively as clone types become more complex.

The detection performance for Type-3 clones exhibits pronounced dataset-dependent variability, with different models demonstrating distinct strengths. On the ST3 dataset, *DeepSeek-V3* (97.20%), *DeepSeek-R1* (95.00%), and *GPT-4.1* (94.00%) achieve the best performance. In contrast, on the MT3 dataset, *GPT-4.1* attains the highest detection rate (95.80%), while *Gemini-2.5-Flash* suffers a substantial performance drop, decreasing from 72.60% on ST3 to 34.00%. Notably, small-scale open-source models perform particularly poorly on MT3, with detection rates consistently below 5%, indicating limited generalization capability in more complex Type-3 clone scenarios. These results further highlight the differences in adaptability across models when handling structurally and semantically complex clones.

Type-4 semantic clones represent a major performance bottleneck for all evaluated models, with detection accuracy dropping substantially. Among all models, *GPT-4.1* demonstrates the strongest semantic understanding capability, achieving a detection rate of 77.60%. A second performance tier includes *DeepSeek-R1* (61.80%) and *Qwen3-Coder-30B-Instruct* (60.20%). In contrast, the remaining models exhibit a pronounced performance gap: *GPT-3.5-Turbo* drops to 47.40%, *Claude-Sonnet-4-5* and *DeepSeek-V3* fall below 30%, while *Gemini-2.5-Flash* and smaller-scale open-source models achieve detection rates below 10%. These results indicate that, despite their overall strengths, most models still struggle to effectively capture semantic-level clone relationships.

Bad Case Analysis. To further investigate the underlying mechanisms of model errors in code clone detection under the Simple Prompt setting, we conduct a systematic categorization and in depth analysis of representative error cases, as

TABLE III: FP and FN Analysis Results of LLM Code Clone Detection Using *Simple Prompt*

Methods	Model-Cap. Errors			Instr.-Level Errors		
	FP	FN	Total	FP	FN	Total
Gemini-2.5-Flash	49	1197	1246	80	190	270
GPT-4.1	364	177	541	0	0	0
GPT-3.5-turbo	267	544	811	0	0	0
Deepseek-V3	163	209	372	63	305	368
Claude-sonnet-4-5	158	335	493	93	148	241
Deepseek-R1	314	249	563	9	23	32
Starchat2-15B	5	56	61	490	1830	2320
Deepseek-Coder-7B-Instruct	20	1107	1127	194	810	1004
Qwen3-Coder-30B-Instruct	302	257	559	212	555	767

summarized in Table III. The observed errors can be clearly divided into two distinct and mutually independent types, which differ substantially in their manifestations, the models most susceptible to them, and their underlying causes.

1) The first type of error originates from insufficient model capability. A defining characteristic of this error type is that the model correctly follows the instruction to produce binary yes or no outputs, thereby satisfying the required output format of the experiment. However, the predicted judgments deviate from the ground truth labels, resulting in incorrect false positive or false negative outcomes.

Experimental results indicate that this error type constitutes the primary error category for closed source models. Specifically, all erroneous outputs produced by the two closed source models, *GPT-4.1* and *GPT-3.5-Turbo*, fall exclusively into this category, with no other error types observed. For *GPT-4.1*, a total of 541 errors are recorded, including 364 false positives and 177 false negatives. *GPT-3.5-Turbo* exhibits a higher error count of 811, consisting of 267 false positives and 544 false negatives. The underlying cause of these errors can be attributed to an insufficient understanding of domain specific methodologies and the core decision logic required for code clone detection. As a result, the model fails to accurately capture critical clone characteristics between code snippets, including syntactic structure, business logic, and execution flow, leading to judgments that are inconsistent with the ground truth labels.

2) The second type of error corresponds to instruction following failures, which occur more frequently and more broadly among open source models. A defining characteristic of this error type is that the model fails to produce the required binary yes or no clone judgments. Instead, it deviates from the task objective and generates invalid outputs, such as derivative code snippets based on the input functions, irrelevant code analysis content, or redundant syntax explanations. Due to the severe deviation from both the required output format and the core task objective, valid clone relationship judgments cannot be extracted from such outputs. Consequently, these cases result in false positive or false negative errors and significantly degrade the overall accuracy of the experimental results.

Experimental results show that multiple open source models exhibit consistently high frequencies of the second error type. Among them, *StarChat2-15B* is the most severely affected model, producing 490 false positives and 1830 false negatives, for a total of 2320 errors. This error count is substantially higher than that of other open source models, indicating that *StarChat2-15B* exhibits the most pronounced manifestation of instruction following failures in our evaluation. In addition, other open source models, including *Deepseek-V3* with 368 errors, *Deepseek-R1* with 32 errors, *Deepseek-Coder-7B-Instruct* with 1004 errors, and *Qwen3-Coder-30B-Instruct* with 767 errors, also exhibit this error type to varying extents. These results further confirm the widespread presence of instruction following errors among open source models.

TABLE IV: T-Test Results for Performance Differences Between LLM-Based and Traditional Code Clone Detection

Performance Metrics	T-value	P-value	Significance ($\alpha = 0.05$)
Recall	4.2276	0.0026	Significance
Precision	-3.3497	0.0036	Significance

During the in depth analysis of the second error type in *Deepseek-Coder-7B-Instruct*, we identify a distinctive and representative error manifestation, namely false negative errors caused by the model directly refusing to respond. A total of 75 such cases are observed. In these instances, the model fails to generate any valid output related to code clone detection or the required binary judgments. Instead, it produces standardized capability disclaimer responses. For example, the model repeatedly outputs statements that lack practical judgment value, such as the following: "As an AI, I do not have the capability to directly analyze code snippets from external sources. However, I can explain that determining whether two code snippets are clones typically requires a detailed comparison of their structure and logic."

The underlying cause of this special error type is likely related to the scale of the input data. When the combined length of the input code snippets and the task description exceeds the model context window limit, the model fails to effectively parse the core task requirements. As a consequence, it cannot accurately comprehend the objective and decision logic of code clone detection. As a result, the model produces generic capability disclaimer responses instead of valid clone judgments, leading to false negative errors. This phenomenon also provides valuable insights for future efforts to improve the robustness of open source models in clone detection tasks.

Comparing Against Traditional Detection Approaches. To further clarify the role of large language models in code clone detection tasks, we conduct a comparative analysis between LLM based approaches and traditional detection approaches. Specifically, quantitative statistical tests are employed to examine whether the observed performance differences between the two categories are statistically significant. To ensure that differences in core performance metrics are not attributable to random variation, an independent samples t-test is applied using Recall and Accuracy as evaluation metrics. The significance level is set to $\alpha = 0.05$, and the corresponding statistical results are reported in Table IV.

According to the results reported in Table IV, the t-value for the Recall metric between the two categories is 4.2276 with a corresponding p-value of 0.0026, while the t-value for

TABLE V: LLM Code Clone Detection Performance with Different Prompt Component Combinations

Combination	Method	Recall (%)					Precision (%)	Average (%)
		T1	T2	ST3	MT3	T4		
Sys-P + Task-P	*Qwen3-Coder-30B-Instruct*	100.00	100.00	96.00	94.00	58.00	83.58	86.29
	GPT-3.5-turbo	100.00	96.00	100.00	84.00	30.00	87.98	
	Claude-sonnet-4-5	100.00	100.00	100.00	96.00	72.00	87.31	
Sys-P + Examples	*Qwen3-Coder-30B-Instruct*	98.00	94.00	92.00	94.00	36.00	79.62	84.71
	GPT-3.5-turbo	100.00	98.00	100.00	84.00	32.00	88.09	
	Claude-sonnet-4-5	100.00	100.00	100.00	92.00	54.00	86.43	
Task-P + Examples	*Qwen3-Coder-30B-Instruct*	98.00	100.00	66.00	96.00	46.00	76.89	84.01
	GPT-3.5-turbo	100.00	96.00	98.00	84.00	36.00	88.84	
	Claude-sonnet-4-5	100.00	100.00	100.00	100.00	78.00	86.28	
Sys-P + Task-P + Examples	*Qwen3-Coder-30B-Instruct*	98.00	100.00	70.00	96.00	46.00	77.65	84.08
	GPT-3.5-turbo	100.00	94.00	100.00	84.00	30.00	88.31	
	Claude-sonnet-4-5	100.00	100.00	100.00	100.00	78.00	86.28	

Accuracy is -3.3497 with a p-value of 0.0036. In both cases, the p-values are below the predefined significance level, indicating statistically significant performance differences between traditional detection approaches and LLM based detection approaches. These results suggest that the observed disparities in recall and classification accuracy stem from inherent methodological differences between the two categories rather than random variation. Furthermore, they provide empirical evidence that the performance advantages of LLM based approaches in code clone detection tasks arise from their intrinsic technical characteristics.

Building on the statistically significant differences, a further comparative analysis reveals complementary strengths between traditional detection approaches and LLM based approaches. For Type-1 and Type-2 clone detection, traditional methods exhibit near perfect performance. Both token based and AST based techniques achieve a Recall of 100%, significantly outperforming several LLM models. This observation indicates that for clone types primarily characterized by textual or structural similarity, deterministic matching strategies provide highly reliable detection. However, traditional methods demonstrate clear limitations when handling more complex clone types. For MT3 clones, even the best performing traditional approach achieves a Recall of only approximately 30%, while token based and AST based methods detect almost no clones. The limitation becomes more pronounced for Type-4 semantic clones, where the Recall of all traditional methods remains below 10%. In contrast, LLM based models such as *GPT-4.1* and *Deepseek-R1* achieve substantially higher Recall rates of 77.60% and 61.80%, respectively. This performance gap can be attributed to the reliance of traditional methods on explicit feature matching, which is insufficient for capturing deep semantic equivalence across diverse implementations. To further substantiate this observation, we released all clone pairs detected only by LLMs but missed by traditional methods. These examples are available in the project's GitHub repository. Overall, the empirical results suggest that traditional clone detection techniques and LLM-based approaches exhibit complementary capabilities, rather than a simple performance dominance relationship.

Building on this complementarity, an effective detection framework can adopt a hierarchical strategy. Traditional methods can be employed as an initial filtering stage to efficiently identify simple clone types, such as Type-1 and Type-2. Subsequently, LLMs can be applied to the remaining ambiguous code pairs to perform deeper semantic analysis, enabling accurate detection of higher level clones including Type-3 and Type-4. Such a hybrid approach balances detection efficiency and accuracy, thereby achieving improved overall performance.

Summary: *LLM based approaches demonstrate superior performance in detecting Type-3 and Type-4 code clones, with Deepseek-R1 and GPT-4.1 achieving the best overall performance. However, LLMs are slightly less effective than traditional detection tools for Type-1 and Type-2 clone detection.*

B. RQ2: Can informative prompts improve the code clone detection performance of LLMs?

Prior research on large language models has shown that carefully designed prompts can significantly improve task performance. Following common prompt enrichment strategies for LLM task optimization, we enhance the prompts along three dimensions. (1) *System Prompt* (Sys-P): Providing a concise introduction to the background of the code clone detection task and explicitly defining the role of the model. (2) *Task Prompt* (Task-P): Clearly specifying code clone types, outlining potential analytical perspectives, and guiding the model to derive conclusions through step by step reasoning using Chain of Thought prompting. (3) *Few-shot Examples* (Examples): Supplying three pairs of sample code snippets along with their corresponding analysis processes and final answers as demonstrations, which help the model better understand the task.

We evaluate the effectiveness of the proposed three dimensional prompt design through a series of experiments, as detailed in the following sections.

1) RQ2.1: Which prompt components are most effective in improving LLM-based code clone detection performance?

To quantify the contribution of individual prompt components to code clone detection performance, we conduct an ablation study based on the previously proposed three component informative prompt. Specifically, we construct multiple prompt

TABLE VI: Comparison of LLM Code Clone Detection Performance Using *Informative Prompts* and *Simple Prompts*

Method	***Informative Prompt***						***Simple Prompt***					
	Recall					**Precision**	**Recall**					**Precision**
	T1	**T2**	**ST3**	**MT3**	**T4**		**T1**	**T2**	**ST3**	**MT3**	**T4**	
Deepseek-Coder-7B-Instruct	95.20	76.80	65.00	32.60	18.40	74.50	71.40	9.80	27.00	4.20	4.20	73.15
Qwen3-Coder-30B-Instruct	100.00	100.00	94.20	99.40	80.60	83.11	98.80	16.00	73.40	89.20	60.20	76.66
Starchat2-15B	98.60	73.80	52.00	47.40	12.20	72.75	98.60	10.20	13.20	0.80	0.00	55.37
Deepseek-V3	89.20	83.40	82.20	82.80	53.00	72.95	100.00	90.40	97.20	79.80	29.80	89.78
Deepseek-R1	99.60	97.40	95.00	91.00	75.00	83.70	100.00	99.80	95.00	89.00	61.80	87.34
GPT-3.5-turbo	100.00	96.20	95.20	89.20	56.60	88.72	89.60	92.20	77.20	84.80	47.40	87.99
GPT-4.1	100.00	99.80	91.20	86.80	65.60	77.87	100.00	97.20	94.00	95.80	77.60	86.45
Gemini-2.5-Flash	100.00	100.00	97.60	96.80	84.00	86.35	91.80	16.00	72.60	34.00	8.20	89.61
Claude-sonnet-4-5	100.00	100.00	98.60	96.40	79.20	86.85	100.00	99.20	94.60	82.20	27.40	88.93

variants by selectively removing one component at a time and evaluate model performance under each configuration.

Experiments are conducted on a randomly sampled subset of 50 instances from each dataset. Three representative models are selected, including *GPT-3.5-Turbo*, *Claude-Sonnet-4-5*, and *Qwen3-Coder-30B-Instruct*. The evaluated prompt configurations include the full prompt and variants with one component removed. The experimental results are reported in Table V.

The ablation results reported in Table V show that, in terms of average performance, the prompt configuration combining the *System Prompt* and *Task Prompt* achieves the best overall results, with a score of 0.8629, slightly outperforming the other configurations. In contrast, prompt variants that incorporate Few shot Examples consistently exhibit noticeable performance degradation. This degradation can be attributed to the high diversity of clone pairs, which makes it difficult for a limited number of examples to be both comprehensive and representative. As a result, the model detection capability for challenging clone types, particularly Type-3 and Type-4, is weakened, leading to a decline in overall performance. Consequently, the prompt configuration that excludes Few shot Examples and retains only the System Prompt and Task Prompt yields the optimal performance.

When examining detection performance across different clone types, the *System Prompt* plus *Task Prompt* configuration demonstrates clear advantages, especially for the most challenging Type-4 clone detection task. For *Qwen3-Coder-30B-Instruct*, the Type-4 recall achieved by the *System Prompt* plus *Task Prompt* configuration reaches 58%, which is substantially higher than that of the *System Prompt* plus *Few shot Examples* at 36% and the *Task Prompt* plus *Few shot Examples* at 46%. A similar pattern is observed for *Claude-Sonnet-4-5*. In this case, the *System Prompt* plus *Task Prompt* configuration achieves a Type-4 recall of 72%, which is slightly lower than that of the *System Prompt* plus *Task Prompt* plus *Few shot Examples* and the *Task Prompt* plus *Few shot Examples* configurations at 78%. However, it attains a notably higher no clone recognition rate of 86.40%, resulting in better overall accuracy. In contrast, *GPT-3.5-Turbo* exhibits only minor performance variations across all prompt configurations, indicating relatively low sensitivity to prompt design. This robustness suggests that the model can accurately capture task intent even with concise prompts, reflecting stronger task generalization capability.

Considering overall accuracy, Type-4 detection performance, and prompt conciseness, the *System Prompt* plus *Task Prompt* configuration is selected as the standard prompt for subsequent experiments. This finding indicates that, for code clone detection tasks, a prompt strategy that combines explicit task definition, Chain of Thought guidance, and clear specification of task background and model role is the most effective. By contrast, incorporating *Few shot Examples* is unnecessary and may even negatively affect overall detection performance.

2) *RQ2.2: Do informative prompts outperform simple prompts in LLM-based code clone detection?*

Building on the findings of RQ2.1, the combination of *System Prompt* and *Task Prompt* is adopted as the optimized *Informative Prompt*. This prompt configuration is further evaluated across all models, with detailed results reported in Table VI.

The results show that the *Informative Prompt* significantly improves code clone detection performance for most models, with particularly pronounced gains on clone types where baseline performance under the *Simple Prompt* is weak. The effectiveness of the Informative Prompt primarily stems from its provision of explicit task definitions, structured analytical guidance, and Chain of Thought reasoning cues, which enable models to better capture complex clone transformation patterns and deep semantic relationships. Notably, Type-2 clone detection exhibits the most substantial improvement. The detection rates of *Gemini-2.5-Flash* and *Qwen3-Coder-30B-Instruct* increase from 16.00% to 100.00%, while *StarChat2-15B* improves from 10.20% to 73.80%, and *Deepseek-Coder-7B-Instruct* from 9.80% to 76.80%. Similar improvements are observed for Type-3 clones. For example, substantial performance gains on MT3 are achieved, with *Gemini-2.5-Flash* improving from 34.00% to 96.80% and *Qwen3-Coder-30B-Instruct* reaching 99.40%. In addition, *StarChat2-15B* transitions from near ineffectiveness to practical detection capability. Even for the most challenging Type-4 semantic clones, most models achieve noticeably better performance under the *Informative Prompt* than under the *Simple Prompt*.

However, the performance gains introduced by the *Informative Prompt* are not universal, and several models with strong baseline performance exhibit a degradation in detection accuracy. This degradation can be attributed to limitations in instruction following and the integration of extensive contextual information, which reduce adaptability to more so-

TABLE VII: FP and FN Analysis Results of LLM Code Clone Detection Using *Informative Prompt*

Methods Methods	Model-Cap. Errors			Instr.-Level Errors		
	FP	FN	Total	FP	FN	Total
Gemini-2.5-Flash	378	108	486	0	0	0
GPT-4.1	629	283	912	1	0	1
GPT-3.5-turbo	267	292	559	11	22	33
Deepseek-V3	724	547	1271	0	0	0
Claude-sonnet-4-5	359	129	488	0	0	0
Deepseek-R1	434	202	636	12	8	20
Starchat2-15B	128	791	919	404	289	693
Deepseek-Coder-7B-Instruct	89	812	901	404	248	652
Qwen3-coder-30B-Instruct	392	84	476	90	45	135

phisticated prompt structures. For instance, *GPT-4.1* shows noticeable performance declines on ST3, MT3, and Type-4 clones, while *Deepseek-V3* also exhibits varying degrees of degradation across clone types. These observations suggest that overly complex prompts may introduce information overload, example interference, or reasoning misalignment, thereby disrupting previously stable decision processes.

Bad Case Analysis. As illustrated in Table VII, corresponding to the two-category error taxonomy identified under *Simple Prompts*, model errors under *Informative Prompts* can still be classified into the same two categories; however, they exhibit distinctly different characteristics.

First, errors stemming from insufficient model capability emerge as the primary source of failure under the Informative Prompt setting. By guiding models to explicitly decompose the judgment logic of code clone detection in a step-by-step manner, the *Informative Prompt* effectively suppresses instruction-following errors. Consequently, the majority of observed failures are no longer attributable to the inability to extract valid clone judgments due to output format violations. Instead, they predominantly manifest as cases in which models correctly follow the instruction to produce binary "yes/no" judgments (or derive such judgments after structured reasoning), yet the resulting decisions remain inconsistent with the ground-truth labels, leading to false positives or false negatives. This error type is consistently observed across both closed-source models (e.g., *GPT-4.1* and *GPT-3.5-Turbo*) and open-source models that still exhibit non-negligible error rates (*e.g., Deepseek-R1* and *Qwen3-Coder-30B-Instruct*). The underlying cause remains the limited mastery of the professional and domain-specific judgment logic required for accurate code clone detection, with only a subset of models demonstrating genuine capability improvements through step-by-step reasoning.

Second, instruction-following errors are substantially reduced, and in most cases fully eliminated, under the *Informative Prompt*. By providing explicit step-by-step reasoning guidance and clearly constraining the expected output structure, the *Informative Prompt* effectively aligns model behavior with the task objective of binary clone judgment. As a result, compared with the *Simple Prompt* setting, instruction deviations such as returning derivative code fragments or unrelated analytical content occur far less frequently in the experimental results. Even for models with high overall error counts, such as *Deepseek-Coder-7B-Instruct* and *Starchat2-15B*, the dominant source of failure shifts under the *Informative Prompt*. Instruction-following errors, which were prevalent under the *Simple Prompt*, are markedly reduced and become a minority of error cases. The remaining failures are mainly caused by limited judgment capability, rather than deviations from the required output format or task objective. Furthermore, the refusal-to-respond behavior previously exhibited by *Deepseek-Coder-7B-Instruct* under the Simple Prompt setting is completely eliminated.

TABLE VIII: Total Cost and Average Cost per Task for LLM Methods under *Simple Prompts* and *Informative Prompts*

Method	Simple Prompt		Informative Prompt	
	Total ($)	Single Task (¢)	Total ($)	Single Task (¢)
GPT-3.5-turbo	3.91	0.08	6.42	0.13
GPT-4.1	13.26	0.27	27.64	0.55
Gemini-2.5-Flash	4.20	0.08	31.35	0.63
Deepseek-V3	1.92	0.04	3.95	0.08
Deepseek-R1	9.60	0.19	13.52	0.27
Claude-sonnet-4-5	27.85	0.56	51.84	1.04

***Summary:** The combination of System Prompt and Task Prompt constitutes the most effective Informative Prompt for code clone detection, improving performance for most models, particularly on challenging clone types. While some models, such as Gemini-2.5-Flash, benefit significantly, models with strong baseline performance may experience slight performance degradation.*

C. RQ3: How can an effective balance between the costs and benefits of using LLMs be achieved for code clone detection?

To evaluate the computational costs and deployment overheads of various models in code clone detection tasks, this study conducts targeted tests on 4 closed-source commercial models and 2 commercialized deployment versions of open-source models, with the relevant experimental data presented in Table VIII.

From the perspective of the relationship between model cost and detection performance, the cost effectiveness of different models exhibits variation and shows no clear correlation with clone detection accuracy. Among the evaluated models, *Claude-Sonnet-4-5* incurs the highest cost under both the Simple Prompt and Informative Prompt settings, with per task costs of 0.56 cents and 1.04 cents, respectively. This high cost can be attributed to two factors. First, the model adopts a relatively high base pricing scheme. Second, it autonomously generates lengthy intermediate analysis content that exceeds the experimental requirement of producing only binary yes or no judgments, resulting in persistently high token consumption. In contrast, *GPT-4.1* achieves the most favorable cost performance trade off, with a per task cost of 0.27 cents while delivering the highest clone detection performance among all tested models. *GPT-3.5-Turbo* exhibits the lowest cost, averaging 0.08 cents per task, yet its detection performance still surpasses that of *Gemini-2.5-Flash*. These results demonstrate that higher monetary cost does not necessarily translate into

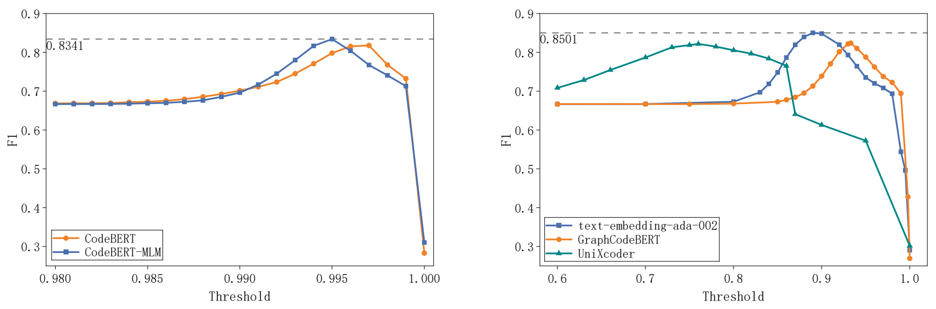


Fig. 1: The left figure shows the F1 performance of *CodeBERT* and *CodeBERT-MLM* at different thresholds. The right figure shows the performance of *Text-embedding-ada-002* ,*GraphCodeBERT* and *Unixcoder*.

better detection performance. Instead, the inherent capability and task adaptability of the model play a more critical role.

From the perspective of prompt switching, the *Informative Prompt* generally leads to increased token consumption, although the magnitude of this increase varies across models depending on their reasoning behavior and output characteristics. The primary reason for this increase is that the *Informative Prompt* guides models to perform clone detection following explicit reasoning steps, which requires generating complete intermediate analysis content. Compared with the concise binary outputs required by the *Simple Prompt*, this reasoning process significantly increases token usage. Among the evaluated models, *Gemini-2.5-Flash* exhibits the largest cost increase, rising from 0.08 cents per task under the *Simple Prompt*, to 0.63 cents under the *Informative Prompt*. This behavior can be explained by the fact that under the *Simple Prompt*, the model produces only brief judgments without exposing its reasoning process, which leads to poor detection performance. When guided by the Informative Prompt, the model is encouraged to generate structured reasoning, enabling it to better utilize its capabilities, which substantially increases token consumption while yielding the largest performance improvement. In contrast, *Deepseek-R1* shows the smallest cost increase, with token consumption under the *Informative Prompt* only 1.4 times that under the *Simple Prompt*. This is because *Deepseek-R1* already produces explicit reasoning content under the *Simple Prompt*, so switching to the *Informative Prompt* requires only minor adjustments, resulting in limited additional token usage.

Summary: *Informative Prompt generally increase token consumption in code clone detection tasks, with the magnitude of this increase depending on each model's reasoning and output behavior. Moreover, higher model cost does not necessarily lead to better detection performance.*

D. RQ4: Can embeddings generated by LLMs outperform those produced by pre-trained code embedding models in code clone detection?

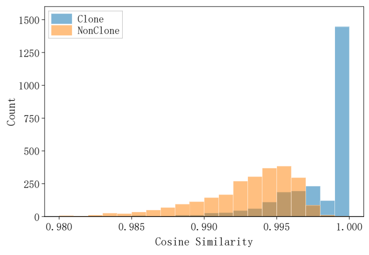
This section offers a comparative analysis of the performance of various LLMs, specifically focusing on their usage of code embedding. This is done by contrasting their results with established PLMs, including *CodeBERT*, *CodeBERT-MLM*, *Unixcoder*, *GraphCodeBERT* and *text-embedding-ada-002*. In our study, the performance of five models is evaluated based on their ability to identify cloned code pairs in the Java dataset. The models were trained to predict similarity between pairs of code, which was computed as the cosine similarity between the vector representations of the code pairs.

In order to capture the nuances of model's performance, we varied the probability thresholds and measured the precision, recall, and F1 scores at each level. Each model was analyzed at its respective threshold which corresponded to its optimal performance. The comparative F1-score across the different thresholds is graphically represented in Figure 1.

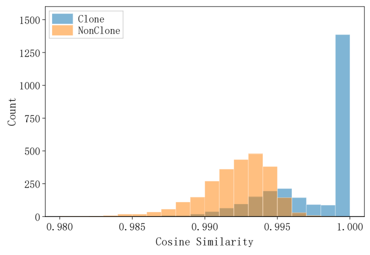
In terms of F1-score performance, *text-embedding-ada-002* attains its optimal performance at a threshold of 0.89, with an F1-score as high as 0.8501, ranking the highest among all models. Among traditional pre-trained models, *CodeBERT-MLM* outperforms other models of the same category in peak performance, achieving an F1-score of 0.8341. The optimal thresholds vary across models: 0.997 for *CodeBERT*, 0.995 for *CodeBERT-MLM*, 0.76 for *Unixcoder*, and 0.93 for *GraphCodeBERT*. These threshold differences are essentially derived from the distinct vector generation characteristics of each model, and the specific manifestations of these characteristics can be further clearly observed in Figure 2.

As depicted in Figure 2, the similarity scores generated by *text-embedding-ada-002* feature a broader distribution range and higher discriminative precision, a characteristic that enables it to more effectively distinguish between true positive and false positive samples. Nevertheless, the broader distribution also leads to occasional erroneous predictions in the high similarity score interval. Despite these occasional errors, the experimental results fully demonstrate that *text-embedding-ada-002* delivers the most robust performance in code clone pair detection. Its superior similarity score distribution characteristics further confirm its reliability and effectiveness in differentiating between clone and non-clone code pairs, highlighting the robustness of the model.

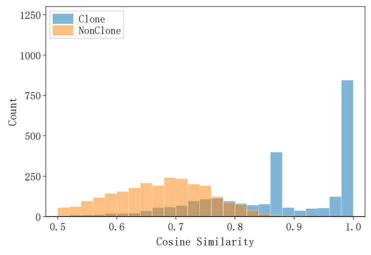
Summary: *Text-embedding-ada-002 is more effective than dedicated code pre-trained models in identifying code clones, demonstrating superior performance. Its core advantage lies in its ability to generate similarity scores with a broader range, thereby enabling more accurate differentiation between true positives and false positives.*



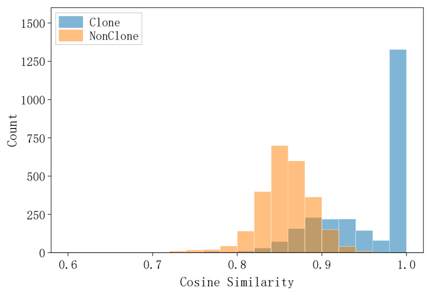
(a) CodeBERT



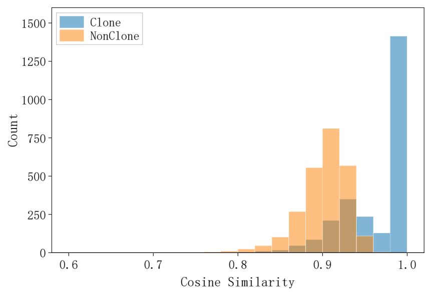
(b) CodeBERT-MLM



(c) UnixCoder



(d) Text-embedding-ada-002



(e) GraphCodeBERT

Fig. 2: The similarity distribution between the two codes embedding.

TABLE IX: Recall and Precision on Java, Python, and C++ Code Clone Detection

Method	**Language**	**Recall (%)**	**Precision (%)**
Qwen3-Coder-30B-Instruct	Python	58.60	100.00
	Java	20.40	100.00
	C++	21.00	100.00
GPT-4.1	Python	86.40	100.00
	Java	89.00	100.00
	C++	90.20	100.00

E. RQ5: How does the performance of LLMs in code clone detection vary across different programming languages?

This section evaluates the performance of large language models in cross programming language code clone detection. We select *GPT-4.1* and *Qwen3-coder-30B-Instruct* as representative commercial and open source models. Experiments are conducted on a cross language dataset to assess their code clone detection performance in multilingual scenarios.

As shown in Table IX, *GPT-4.1* significantly outperforms *Qwen3-coder-30B-Instruct* across all evaluation metrics. Its advantages remain stable across different clone types and target languages, with particularly strong recall performance on Python and C++. These results indicate that a model's ability to understand the syntax and structural characteristics of multilingual code has a direct impact on cross language code clone detection performance.

Further examination of the detection results across different programming languages reveals that, for the open source model *Qwen3-coder-30B-Instruct*, the detection performance on Python is significantly better than that on Java and C/C++. This observation indicates that even under the same model and identical detection settings, intrinsic characteristics of programming languages can still exert a substantial influence on code clone detection performance.

On this basis, code complexity at the language level emerges as a key factor in explaining this discrepancy. Compared with Java and C/C++, Python exhibits lower overall complexity in terms of syntactic structure, type system, and control flow organization, and can typically implement the same functionality with fewer lines of code. In code clone detection tasks, shorter code snippets with lower complexity are easier for models to process. Such code involves fewer type constraints, control flow branches, and dependency relationships. As a result, the difficulty of semantic modeling and similarity alignment is substantially reduced.

Further, Python's dynamic typing mechanism and high level abstractions partially conceal low level implementation details, causing code logic to be primarily expressed through function call relationships and control structures. This low complexity characteristic enables models to more directly extract function level semantic features, leading to greater stability, particularly in challenging clone scenarios such as Type-3 and Type-4. In contrast, explicit type declarations, generic or template mechanisms, as well as low level operations such as pointer manipulation and memory management in Java and C/C++ significantly increase structural and semantic complexity. These elements, which are weakly related to core functional semantics but exhibit substantial structural variation, introduce additional noise in clone determination and thereby interfere with accurate identification of functionally equivalent code.

From the perspective of cross language clone detection, higher code complexity increases the structural differences between functionally equivalent code. This makes it harder for models to align control flow, data flow, and call relationships. In contrast, Python has lower overall complexity. Its implementations are more likely to share similar logical structures and invocation patterns. This allows models to build more stable and consistent semantic correspondences across samples. This characteristic provides a plausible explanation for the higher recall and greater detection stability observed for Python.

In contrast, *GPT-4.1* demonstrates consistently strong performance across languages, which can be attributed to its larger parameter scale and superior capacity for long text modeling and comprehension. Even when handling scenarios involving Java and C++ with longer code snippets and higher structural and semantic complexity, its detection performance remains robust and significantly outperforms comparable open source models. This indicates that large scale models handle high complexity code more effectively. They can better reduce redundant syntactic information and model complex control flow and type dependencies. As a result, they extract core semantic features for clone detection more accurately. In addition, the abundance of Python related training data may further enhance the model's semantic alignment capability and recall performance for this language.

Summary: *Open source models perform well mainly on Python due to its lower code complexity and more concise structural representation, while commercial models achieve strong and stable performance across all languages thanks to their larger scale and stronger modeling capability.*

V. Discussions and Limitations

A. Lessons

Based on the comprehensive evaluation, we summarize several key lessons and suggestions for code clone detection using LLMs:

For Researchers: LLMs have demonstrated tremendous potential in the field of code clone detection, and they outperform traditional detection approaches particularly in detecting complex semantic clones. However, to fully unlock this potential, meticulous prompt engineering is crucial. Information prompting strategies can effectively boost the performance of most LLMs, yet this effect is subject to significant model dependency. Furthermore, in the process of exploring the application of LLMs in clone detection, open-source and commercial LLMs exhibit notable differences in terms of cost-effectiveness: lightweight open-source models feature convenient deployment but may suffer from insufficient reliability; while advanced commercial LLMs can deliver superior detection results, they incur higher computational and financial costs. Notably, there is no strict positive correlation between the performance of LLMs and the level of cost investment. It is still necessary to conduct targeted analysis and selection for specific models.

For Users: LLMs can serve as valuable adjuncts to existing tools in the realm of code clone detection. They are particularly adept at identifying complex clone pairs such as Type-3 and Type-4, effectively capturing deep semantic similarities that are easily overlooked by traditional syntactic analysis methods, thus significantly improving the recall rate in complex cloning scenarios. However, it should be noted that LLMs are slightly inferior to traditional tools in terms of detection accuracy for simple clone pairs like Type-1 and Type-2. Therefore, the combined scheme of preliminary filtering by traditional tools and precise verification by LLMs can achieve complementary advantages, making it the optimal choice for balancing detection efficiency and accuracy at present, especially suitable for large-scale and high-precision code clone detection requirements.

B. Discussions

Why can informative prompts enhance the clone detection performance of high-performance LLMs? The core lies in expanding the context range for model prediction. Without such prompts, the model responds only based on the two given code samples; after implementation, the prediction context includes not only the two code samples but also the task guidance and thinking process contained in the informative prompts, thereby enabling a more comprehensive analysis of code pairs and improving detection performance.

How feasible is it to provide an entire codebase to LLMs and obtain all possible clone pairs or clone classes? Directly performing exhaustive clone detection on an entire codebase using LLMs is infeasible, and an efficient approach requires combining LLMs with traditional tools in a staged process. This is primarily because LLMs have strict input length limitations, and the computational cost of pairwise or class-wise comparison grows quadratically with code size, making full-scale scanning impractical in both computational and economic terms. To validate this strategy, we conducted an experiment on a GitHub Java project. We first used SourcererCC to scan the project, generating approximately 1 million clone pairs, and then lowered the similarity threshold by 50% to add around 300,000 additional candidate clones. We randomly sampled 400 of these candidates and submitted them to the LLM for detection. After manual verification, 98 out of the 110 clone pairs identified by the LLM were confirmed as true clones. These results demonstrate that a two-stage approach, where traditional tools first filter candidates and the LLM then performs precise detection, is feasible. This indicates that LLMs are better suited for high-value detection on candidate sets rather than exhaustive scanning of the entire codebase..

Why does code embedding perform better than LLMs chat in clone detection tasks? The success of code embedding over LLMs chat is attributed to its different approach to detecting cloned code. Code embedding creates an individual representation for each code through a text encoder, which is then compared using cosine similarity. This process does not involve a comparative analysis of the two codes, thereby simplifying the task as compared to directly performing clone detection on two codes. Although LLMs with CoT can provide an analysis for each code and then compare the results, the output in the form of natural language text makes this process more complex compared to direct encoding to obtain representations. Also, the final comparison stage still demands a strong context-understanding capability from LLMs to compare the longer code segments. As a result, the code embedding task appears simpler both in terms of code analysis and code pair comparison, thereby leading to better performance.

C. Limitations

Limitations in dataset construction. Although we attempted to mitigate data leakage through dataset filtering, this guarantee mainly relies on conclusions from prior studies. In practice, we adopted their recommended default settings without conducting task-specific parameter tuning. Moreover, the cross-language dataset only covers Java, C/C++, and Python, and does not include other widely used programming languages such as Go and JavaScript. Additionally, this study does not consider the impact of syntactic differences across different versions of the same programming language on clone detection performance. Consequently, the generalizability of our experimental findings across programming languages remains limited.

Limitations in constructing the instruction set. Based on the core requirements of the code clone detection task, we designed an instruction set consisting of system prompts, task prompts, and a small number of examples. However, the optimization of these instructions did not cover the characteristics of all evaluated models. Different models, such as the open-source *Qwen3-Coder-30B-Instruct* and the closed-source *GPT-4.1*, exhibit markedly different sensitivities to instruction structure: some models tend to experience information overload under complex instructions, while others suffer from insufficient reasoning due to overly concise instructions. In summary, the current instruction set has not achieved adaptive optimization across all 14 evaluated models.

Enforcing a response structure during detection. Although the models were explicitly instructed to output a binary judgment of "Yes" or "No," limitations remain in practice. On the one hand, significant differences in instruction-following capabilities among open-source models led some models to produce irrelevant analyses or format deviations, requiring a combination of regular expressions and manual verification, which increased evaluation costs and may have introduced subjective bias. On the other hand, when processing overly long code fragments, some models refused to respond or generated invalid outputs, while our experimental setup lacked explicit rules for handling such abnormal cases, potentially affecting the evaluation accuracy of these samples. Overall, the current evaluation framework is not fully aligned with the output characteristics of different models, and the handling mechanisms for anomalous outputs require further improvement.

VI. Conclusion

This study provides a systematic empirical evaluation of the application of LLMs in automated code clone detection, covering clone types, programming languages, prompt strategies, the cost–benefit balance of LLMs, and LLM-based embedding schemes. The results show that advanced LLMs outperform traditional methods in detecting complex semantic clones, but are slightly weaker in Type-1 and Type-2 clone detection; informative prompts can improve the performance of most LLMs, although their effectiveness is model-dependent; LLM-based text embedding schemes outperform specialized pre-trained code embedding models; and open-source and commercial LLMs exhibit performance differences in cross-language clone detection, with Python achieving the best overall results. Overall, this study provides strong evidence that LLMs, with their powerful natural language and code semantic understanding capabilities, hold great potential for code clone detection. Moreover, the prompt strategies and evaluation methodologies proposed herein offer valuable benchmarks for future research in this emerging field.

References

[1] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based code clone detection: incremental, distributed, scalable," in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–9.

[2] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 595–614.

[3] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on Software Engineering*, 2007.

[4] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queen's School of Computing TR*, 2007.

[5] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: scaling code clone detection to big code," in *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*, 2016.

[6] C. K. Roy and J. R. Cordy, "Nicad: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Proceedings of the 2008 International Conference on Program Comprehension (ICPC'08)*, 2008.

[7] T. Nakagawa, Y. Higo, and S. 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*, 2021, pp. 830–841.

[8] M. Lei, H. Li, J. Li, N. Aundhkar, and D.-K. Kim, "Deep learning application on code clone detection: A review of current knowledge," *Journal of Systems and Software*, vol. 184, p. 111141, 2022.

[9] Y. Wu, D. Zou, S. Dou, S. Yang, W. Yang, F. Cheng, H. Liang, and H. Jin, "Scdetector: Software functional clone detection based on semantic tokens analysis," in *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*, 2020, pp. 821–833.

[10] J. D. M.-W. C. Kenton and L. K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of NAACL-HLT*, 2019, pp. 4171–4186.

[11] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, "Improving language understanding by generative pre-training."

[12] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[13] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. Hoi, "Codet5+: Open code large language models for code understanding and generation," *arXiv preprint arXiv:2305.07922*, 2023.

[14] C. Sun, X. Qiu, Y. Xu, and X. Huang, "How to fine-tune bert for text classification?" in *Chinese Computational Linguistics: 18th China National Conference, CCL 2019, Kunming, China, October 18–20, 2019, Proceedings 18*. Springer, 2019, pp. 194–206.

[15] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, 2023.

[16] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," *arXiv preprint arXiv:2001.08361*, 2020.

[17] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[18] M. N. Team. (2023) Introducing mpt-30b: Raising the bar for open-source foundation models. Accessed: 2023-06-22. [Online]. Available: www.mosaicml.com/blog/mpt-30b

[19] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[20] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[21] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler *et al.*, "Emergent abilities of large language models," *arXiv preprint arXiv:2206.07682*, 2022.

[22] S. Feng and C. Chen, "Prompting is all your need: Automated android bug replay with large language models," *arXiv preprint arXiv:2306.01987*, 2023.

[23] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 423–435.

[24] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei *et al.*, "Starcoder 2 and the stack v2: The next generation," *arXiv preprint arXiv:2402.19173*, 2024.

[25] A. Yang, A. Li, B. Yang, B. Zhang *et al.*, "Qwen3 technical report," 2025. [Online]. Available: https://arxiv.org/abs/2505.09388

[26] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang, "Deepseek-coder: When the large language model meets programming – the rise of code intelligence," 2024. [Online]. Available: https://arxiv.org/abs/2401.14196

[27] DeepSeek-AI, "Deepseek-v3 technical report," 2024. [Online]. Available: https://arxiv.org/abs/2412.19437

[28] D. Guo, D. Yang, H. Zhang *et al.*, "Deepseek-r1 incentivizes reasoning in llms through reinforcement learning," p. 633–638, Sep. 2025. [Online]. Available: http://dx.doi.org/10.1038/s41586-025-09422-z

[29] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, "Training language models to follow instructions with human feedback," *Advances in Neural Information Processing Systems*, vol. 35, pp. 27 730–27 744, 2022.

[30] OpenAI, "Gpt-4.1 technical report," 2023.

[31] "Gemini-2.5-flash technical report," https://ai.google.dev/gemini-api/docs/models/gemini-2.5, 2025.

[32] "claude-sonnet-4-5 technical report," https://www.anthropic.com/news/claude-sonnet-4-5, 2025.

[33] Y. Wu, S. Feng, D. Zou, and H. Jin, "Detecting semantic code clones by building ast-based markov chains model," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3560426

[34] Y. Hu, D. Zou, J. Peng, Y. Wu, J. Shan, and H. Jin, "Treecen: Building tree graph for scalable semantic code clone detection," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3556927

[35] S. Feng, W. Suo, Y. Wu, D. Zou, Y. Liu, and H. Jin, "Machine learning is all you need: A simple token-based approach for effective code clone detection," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639114

[36] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, 2002.

[37] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "Cclearner: a deep learning-based clone detection approach," in *Proceedings of the 2017 International Conference on Software Maintenance and Evolution (ICSME'17)*, 2017.

[38] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy, "Ccaligner: a token based large-gap clone detector," in *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*, 2018.

[39] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, 2007.

[40] H. Liang and L. Ai, "Ast-path based compare-aggregate network for code clone detection," in *Proceedings of the 2021 International Joint Conference on Neural Networks (IJCNN'21)*, 2021.

[41] Y. Golubev, V. Poletansky, N. Povarov, and T. Bryksin, "Multi-threshold token-based code clone detection," in *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'21)*, 2021.

[42] J. Krinke, "Identifying similar code with program dependence graphs," in *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01)*, 2001.

[43] Y. Zou, B. Ban, Y. Xue, and Y. Xu, "Ccgraph: a pdg-based code clone detector with approximate graph matching," in *Proceedings of the 35th International Conference on Automated Software Engineering (ASE'20)*, 2020, pp. 931–942.

[44] B. Hu, Y. Wu, X. Peng, C. Sha, X. Wang, B. Fu, and W. Zhao, "Predicting change propagation between code clone instances by graph-based deep learning," in *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*, 2022, pp. 425–436.

[45] M. Khajezade, J. J. Wu, F. H. Fard, G. Rodriguez-Perez, and M. S. Shehata, "Investigating the efficacy of large language models for code clone detection," in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, ser. ICPC '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 161–165. [Online]. Available: https://doi.org/10.1145/3643916.3645030

[46] "Bigclonebench," https://github.com/clonebench/BigCloneBench, 2020.

[47] D. Dai, Y. Sun, L. Dong, Y. Hao, Z. Sui, and F. Wei, "Why can gpt learn in-context? language models secretly perform gradient descent as meta optimizers," *arXiv preprint arXiv:2212.10559*, 2022.

[48] Q. Dong, L. Li, D. Dai, C. Zheng, Z. Wu, B. Chang, X. Sun, J. Xu, and Z. Sui, "A survey for in-context learning," *arXiv preprint arXiv:2301.00234*, 2022.

[49] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in Neural Information Processing Systems*, vol. 35, pp. 24 824–24 837, 2022.

[50] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, 2023.

[51] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 1–79, 2024.

[52] H. Pearce, B. Ahmad, B. Tan, B. Dolos, and R. Chandra, "Asleep at the keyboard? assessing the security of github copilot's code contributions," *IEEE Symposium on Security and Privacy*, pp. 754–768, 2022.

[53] "Openai," 2023. [Online]. Available: https://openai.com/

[54] Y. Dong, X. Jiang, H. Liu, Z. Jin, B. Gu, M. Yang, and G. Li, "Generalization or memorization: Data contamination and trustworthy evaluation for large language models," in *Findings of the Association for Computational Linguistics: ACL 2024*, L.-W. Ku, A. Martins, and V. Srikumar, Eds. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 12 039–12 050. [Online]. Available: https://aclanthology.org/2024.findings-acl.716/

[55] "Google code jam," https://code.google.com/codejam/past-contests., 2017.

[56] J. Krinke and C. Ragkhitwetsagul, "Bigclonebench considered harmful for machine learning," in *2022 IEEE 16th International Workshop on Software Clones (IWSC)*, 2022, pp. 1–7.

[57] ——, "How the misuse of a dataset harmed semantic clone detection," *arXiv preprint arXiv:2505.04311*, 2025.

[58] D. Zou, S. Feng, Y. Wu, W. Suo, and H. Jin, "Tritor: Detecting semantic code clones by building social network-based triads model," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 771–783. [Online]. Available: https://doi.org/10.1145/3611643.3616354

[59] R. Puri, D. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker, V. Thost, L. Buratti, S. Pujar, S. Ramji, U. Finkler, S. Malaika, and F. Reiss, "Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks," 2021.

[60] M. Qi, Y. Huang, Y. Yao, M. Wang, B. Gu, and N. Sundaresan, "Is next token prediction sufficient for gpt? exploration on code logic comprehension," *arXiv preprint arXiv:2404.08885*, 2024.

[61] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[62] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," *arXiv preprint arXiv:2203.03850*, 2022.

[63] R. Greene, T. Sanders, L. Weng, and A. Neelakantan, Dec 2022. [Online]. Available: https://openai.com/blog/new-and-improved-embedding-model

[64] "Statchat2," https://huggingface.co/HuggingFaceH4/starchat2-15b-sft-v0.1, 2025.

[65] C. Zhang, J. Wang, Q. Zhou, T. Xu, K. Tang, H. Gui, and F. Liu, "A survey of automatic source code summarization," *Symmetry*, vol. 14, no. 3, p. 471, 2022.

[66] A. Neelakantan, T. Xu, R. Puri, A. Radford, J. M. Han, J. Tworek, Q. Yuan, N. Tezak, J. W. Kim, C. Hallacy *et al.*, "Text and code embeddings by contrastive pre-training," *arXiv preprint arXiv:2201.10005*, 2022.

[67] M. Wu, P. Wang, K. Yin, H. Cheng, Y. Xu, and C. K. Roy, "Lvmapper: A large-variance clone detector using sequencing alignment approach," *IEEE access*, vol. 8, pp. 27 986–27 997, 2020.

[68] F.-M. Lazar and O. Banias, "Clone detection algorithm based on the abstract syntax tree approach," in *2014 IEEE 9th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI)*. IEEE, 2014, pp. 73–78.

[69] J. Zhao, K. Xia, Y. Fu, and B. Cui, "An ast-based code plagiarism detection algorithm," in *2015 10th International conference on broadband and wireless computing, communication and applications (BWCCA)*. IEEE, 2015, pp. 178–182.

[70] Y. Yang, Z. Ren, X. Chen, and H. Jiang, "Structural function based code clone detection using a new hybrid technique," in *2018 IEEE 42nd annual computer software and applications conference (COMPSAC)*, vol. 1. IEEE, 2018, pp. 286–291.

[71] T. F. Smith, M. S. Waterman *et al.*, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[72] W. Amme, T. S. Heinze, and A. Schäfer, "You look so different: Finding structural clones and subclones in java source code," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 70–80.

[73] N. Marastoni, A. Continella, D. Quarta, S. Zanero, and M. D. Preda, "Groupdroid: Automatically grouping mobile malware by extracting code similarities," in *Proceedings of the 7th Software Security, Protection, and Reverse Engineering/Software Security and Protection Workshop*, 2017, pp. 1–12.

[74] X. Zhan, L. Fan, S. Chen, F. We, T. Liu, X. Luo, and Y. Liu, "Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1695–1707.

[75] S. W. H. Z. M. W. Y. T. M. Z. M. C. J. F. Z. X. R. Z. Y. W. M. W. T. G. Q. Z. X. Q. X. H. Shihan DOU, Haoxiang JIA, "What is wrong with your code generated by large language models? an extensive study," *SCIENCE CHINA Information Sciences*, vol. 69, no. 1, pp. 112 107–, 2026. [Online]. Available: http://www.sciengine.com/publisher/ScienceChinaPress/journal/SCIENCECHINAInformationSciences/69/1/10.1007/s11432-025-4632-8,doi=

[76] H. Jia, R. Morris, H. Ye, F. Sarro, and S. Mechtaev, "Automated repair of ambiguous natural language requirements," *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering*, 2025.

[77] F. Mu, L. Shi, S. Wang, Z. Yu, B. Zhang, C. Wang, S. Liu, and Q. Wang, "Clarifygpt: Empowering llm-based code generation with intention clarification," *arXiv preprint arXiv:2310.10996*, 2023.

[78] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski, "A learning algorithm for boltzmann machines," *Cognitive science*, vol. 9, no. 1, pp. 147–169, 1985.

[79] J. Ficler and Y. Goldberg, "Controlling linguistic style aspects in neural language generation," in *Proceedings of the Workshop on Stylistic Variation*. Copenhagen, Denmark: Association for Computational Linguistics, Sep. 2017, pp. 94–104. [Online]. Available: https://aclanthology.org/W17-4912

[80] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The curious case of neural text degeneration," *arXiv preprint arXiv:1904.09751*, 2019.

[81] J. Shan, S. Dou, Y. Wu, H. Wu, and Y. Liu, "Gitor: Scalable code clone detection by building global sample graph," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 784–795. [Online]. Available: https://doi.org/10.1145/3611643.3616371