Automated Type-IV Clone Generation via LLMs and Deterministic Validation

LUCIANO MARCHEZAN, Université de Montréal, Canada
EUGENE SYRIANI, Université de Montréal, Canada
KÉVIN DELCOURT, Université de Montréal, Canada
HOUARI SAHRAOUI, Université de Montréal, Canada

Detecting Type-IV code clones, functionally equivalent fragments with different syntax, remains a major challenge for quality assurance. Existing datasets are limited in supporting semantic clone detection due to class imbalance, lack of verified functional equivalence, and data redundancy. We present an automated approach for generating Type-IV clones by leveraging large language models (LLMs) with deterministic testing and filtering. The approach normalizes input code, produces diverse clone candidates through customizable prompts, and ensures semantic equivalence via automated testing and syntactic diversity through CodeBLEU-based filtering. Representative unique clones are then selected by clustering. We evaluate the extent to which LLMs generate diverse Python Type-IV clones, how prompt and generation factors affect quality and efficiency, the retention of only Type-IV clones at the final dataset, and the usefulness of the resulting dataset for fine-tuning embedding models. Results show that the generated clones improve Type-IV clone detection across different programming languages.

CCS Concepts: • **Software and its engineering → Maintaining software**; **Software evolution**.

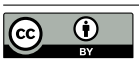
Additional Key Words and Phrases: Semantic code similarity; Code generation; Software maintenance

ACM Reference Format:
Luciano Marchezan, Eugene Syriani, Kévin Delcourt, and Houari Sahraoui. 2026. Automated Type-IV Clone Generation via LLMs and Deterministic Validation. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA062 (October 2026), 23 pages. https://doi.org/10.1145/3832153

1 Introduction

Software clones are a well-known phenomenon in software development [1] and detecting them is critical for software maintenance tasks such as reuse, refactoring, and quality assurance [44]. Types I–III clones, which involve textual or syntactic similarity, can often be detected using straightforward comparison methods. Detecting Type-IV clones, semantically equivalent fragments with potentially entirely different syntax, requires reasoning about program behavior rather than structure, making traditional approaches insufficient [8, 39]. For instance, text- or token-based methods fail when clones exhibit differing control-flow structures, such as recursion versus iteration [46]. Approaches targeting semantic clones have been explored for many years, starting with the work of Gabel et al. [18], who studied behavioral equivalence via program transformations. More recent studies leverage machine learning (ML) techniques [7, 41], yet these methods often underperform [51], partly due to the scarcity of high-quality datasets for training and evaluation [53]. The growing

Authors' Contact Information: Luciano Marchezan, Université de Montréal, DIRO, Montreal, Canada, luciano.augusto.marchezan.de.paula@umontreal.ca; Eugene Syriani, Université de Montréal, DIRO, Montreal, Canada, syriani@iro.umontreal.ca; Kévin Delcourt, Université de Montréal, DIRO, Montreal, Canada, kevin.delcourt@umontreal.ca; Houari Sahraoui, Université de Montréal, DIRO, Montreal, Canada, sahraouh@iro.umontreal.ca.



ACM 2994-970X/2026/10-ARTISSTA062
https://doi.org/10.1145/3832153

adoption of LLM-generated code further increases the importance of Type-IV clone detection, as semantically equivalent implementations can introduce maintainability, redundancy, and licensing concerns while remaining difficult to detect through syntactic analysis alone [50]. Existing datasets, including *BigCloneBench* [43] and *CodeSearchNet* [20], are generally insufficient for Type-IV clone research. For example, the former emphasizes recall for syntactic clones, resulting in class imbalance, sparse negative examples, and a lack of explicit semantic ground truth [27]. Creating high-quality Type-IV datasets manually is costly as it requires verifying correctness through testing or formal analysis [5].

To address the problem of the lack of a proper dataset, we propose a hybrid approach that employs large language models (LLMs) to systematically *produce* diverse Type-IV clones. To deal with the non-deterministic nature of LLMs and to avoid producing low-quality clones, we also employ *deterministic validation* by automatically testing and syntactically filtering clone candidates. Our method generates clone candidates from multiple configurations, including different LLMs, prompt contexts, prompting strategies, and refactoring actions. Candidates are filtered syntactically using a CodeBLEU-based metric [37], validated via automated tests, optionally repaired if partially correct, and clustered to identify representative clones. We implemented our approach as a fully automated pipeline called *Kamino*.

We evaluate Kamino based on 224 unique prompt configurations to generate clones using 927 Python entries from *BigCodeBench* [54]. The results show that LLMs can effectively generate Type-IV clones, with the context and choice of LLM being the primary factors influencing efficiency. To further assess clone quality, we used CloneCognition [34], which considered 99.9% of pairs as Type-IV clones, and complemented this with a manual inspection of 231 randomly sampled pairs by two annotators, achieving substantial agreement ($\kappa = 0.74$) and confirming that the majority of cases (83%) are Type-IV clones.

Furthermore, to demonstrate the applicability of our dataset, we use it to fine-tune code embedding models and evaluate their ability to detect Type-IV clones using unseen Python, Java, and C# Type-IV clone pairs from *SemanticCloneBench* [2], and *GPTCloneBench* [3]. Results show that fine-tuning `CodeBERT` and `CodeT5` improves F1 and the Matthews Correlation Coefficient (MCC) [32] scores, especially for Java, Python, and C#. Lastly, we derive three variants of our dataset considering low, medium, and high syntactic diversity among clones, training the embedding models independently in each variant. The results show that high-diversity clones can improve the results on detecting human-written Type-IV clones (*SemanticCloneBench*) while having mixed results for *GPTCloneBench*.

The main contributions of this paper are: (1) A **hybrid approach for generating high-quality Type-IV clones** by combining LLM-based generation with deterministic validation and clustering (2) An **automated pipeline** that executes this approach with customizable input datasets, prompt configurations, and filtering thresholds (3) A **publicly available dataset** of 42 698 Type-IV clone pairs for training and evaluating ML-based clone detectors (4) **Empirical evidence** of the utility of the generated dataset **for fine-tuning code embedding models**, demonstrating its potential to improve Type-IV clone detection.

The rest of the paper is organized as follows. Section 2 presents the background, challenges, and motivation for our work. Section 3 describes the hybrid Type-IV clone generation approach. Section 4 presents the empirical evaluation, followed by discussion and limitations in Section 5. Section 6 reviews related work, and Section 7 concludes the paper.

2 Background and Motivation

Over the years, different definitions and classifications of code clones have been proposed [8, 39]. The most widely accepted taxonomy distinguishes four types of clones: (1) *Types I–III (textual and*

syntactic clones): near-identical or slightly modified code fragments that share substantial syntactic similarity, ranging from exact copies to variations through insertions or deletions; and (2) *Type-IV (semantic clones):* functionally equivalent fragments that may differ entirely in syntax or structure, making them the most challenging to detect.

Type-IV clones are difficult to identify because their similarity lies in *behavior* rather than structure. This challenge is further worsened as determining whether two arbitrary programs implement exactly the same functionality is undecidable [38]. Consequently, semantic clone detection techniques must rely on practical approximations of behavioral equivalence rather than formal guarantees [1]. Fig. 1 illustrates two Python implementations of a function that samples random numbers based on a given weighted distribution, returning a histogram of the samples. Although both return the same result given the same input, their syntactic representations share almost no overlap (e.g., only one uses Counter), rendering them invisible to traditional text- or token-based clone detection methods [46].

```python
import random
from collections import Counter
def task_func(values, weights, n_samples):
  samples = random.choices(values, weights,
              k=n_samples)
  histogram = dict(Counter(samples))
  return histogram
```

(a) Primary Solution

```python
import random
from collections import defaultdict
def task_func(values, weights, n_samples):
  samples = [random.choices(values, weights=weights)[0]
             for _ in range(n_samples)]
  histogram = defaultdict(int)
  for sample in samples: histogram[sample] += 1
  return dict(histogram)
```

(b) Type-IV clone

Fig. 1. Two functionally equivalent but syntactically different functions in Python

Early work, such as Gabel et al. [18], explored semantic clone detection through program transformations. More recent studies have adopted ML techniques to capture deeper semantic relationships between code fragments [7, 41]. However, ML-based methods often underperform [51], largely due to the scarcity of high-quality datasets with verified examples of behavioral equivalence [53]. The growing adoption of LLM-assisted software development further increases the importance of semantic clone detection. While LLMs can improve developer productivity, generated code may reproduce patterns, logic, or entire solutions observed during training. This raises concerns related to code maintainability, redundancy, and licensing compliance when generated code unintentionally mirrors existing implementations [50]. Consequently, robust Type-IV clone detection is becoming increasingly important for identifying semantic duplication across diverse implementations, coding styles, and programming languages, especially for LLM-generated code.

BigCloneBench remains the most used benchmark for clone detection [53]. Yet, recent analyses have revealed several limitations that make it less suitable for semantic clone research [26, 27, 43]. First, *BigCloneBench* was primarily designed to evaluate recall in syntactic clone detection, resulting in a dataset dominated by true clone pairs. It lacks negative examples, an imbalance that hampers ML model training. Second, it does not provide explicit ground truth for behavioral equivalence, especially for weak Type-III and Type-IV pairs, which together represent about 95% of its entries [27]. Consequently, the dataset offers limited support for approaches seeking to capture semantic similarity.

Several large-scale datasets have been developed, including *BigCodeBench* [54], *CodeNet* [36], and *CodeSearchNet* [20]. These datasets have significantly contributed to progress in ML-for-code research by supporting tasks such as code generation, summarization, and search. However, they were not specifically designed for clone detection. For instance, *CodeNet* links programming tasks to multiple user submissions, which may approximate Type-IV similarity but without explicit verification of behavioral equivalence. In addition, redundancy, inconsistent coding styles, and the

absence of explicit similarity labels limit their applicability to semantic clone detection. Likewise, *CodeSearchNet* emphasizes code-text alignment rather than code-code relationships, making it unsuitable for clone detection research. A recent study leveraged LLMs to construct such datasets, resulting in *GPTCloneBench* [3]. This dataset contains Type-IV clones for Python, Java, C#, and C, and it is an extension of *SemanticCloneBench* [2], which contains manually validated clone pairs extracted from Stack Overflow answers. While valuable, both datasets remain relatively small (e.g., around 7 000 Type-IV clone pairs for Java) when compared to larger benchmarks such as *BigCloneBench*, with over 50 000 pairs. Moreover, their approach relies on manual user intervention to validate the generated clones, thereby not eliminating the manual effort required.

In summary, three main challenges motivate our work: (1) *Complexity of semantic similarity:* detecting Type-IV clones requires reasoning about control flow, data flow, and program behavior, tasks that go beyond syntactic comparison. ML-based models show promise but require rich, labeled data to learn from; (2) *Limitations of existing datasets:* current datasets such as *BigCloneBench*, *CodeNet*, and *CodeSearchNet* lack verified functional ground truth, suffer from imbalance, or contain noise that limits their reliability for training and evaluation; and (3) *Effort in manual curation:* constructing large-scale semantic clone datasets manually demands implementing behaviorally equivalent yet syntactically distinct fragments and verifying them via testing or formal analysis, a process that is slow, error-prone, and difficult to scale. Modern ML methods such as self-supervised learning [5] further require diversity and volume in training data, amplifying the challenge.

To overcome these challenges, several directions can be pursued to generate high-quality Type-IV clone detection datasets. While existing datasets may not directly support this goal, they can serve as foundations. For example, *BigCodeBench* [54] provides Python functions, associated unit tests for correctness, and rich metadata such as function descriptions and dependencies. An example code for an entry is shown in Fig. 1a. However, as each task contains only one implementation, it cannot be directly used for clone detection without generating alternative solutions. A promising strategy for creating such data is to leverage LLMs [35]. LLMs can automatically generate diverse variants of a given code fragment, potentially producing syntactic alternatives that preserve functional behavior. Nevertheless, LLMs are non-deterministic and may produce erroneous or inconsistent outputs [4].

For generating Type-IV clones, two key requirements must be satisfied: (1) generated code fragments must exhibit identical behavior across all valid inputs; and (2) the syntactic structure of the generated code must differ significantly from the original to ensure semantic, not superficial, similarity. Ensuring both properties solely through LLM prompting is difficult. Even when carefully guided, LLMs may fail to follow strict behavioral constraints [29]. Additionally, including all previously generated clones in prompts to maintain diversity can lead to exponential resource growth. Hence, while datasets such as *BigCodeBench* can serve as inputs, additional mechanisms are needed to guarantee both behavioral correctness and syntactic diversity. This motivates our automated Type-IV clone generation approach.

3 Hybrid Type-IV Clone Generation Approach

Fig. 2 illustrates our hybrid clone generation approach that combines LLM-based generation of diverse code clones with deterministic validation through testing and filtering to ensure semantic correctness and syntactic differences. We present each step of the approach next.

3.1 Normalization

This is needed to ensure that all input data conforms to a consistent and machine-readable structure, enabling reliable prompting and validation in the later stages of the pipeline. Without a standardized representation, it would be difficult to automate clone generation and testing. To achieve this, the

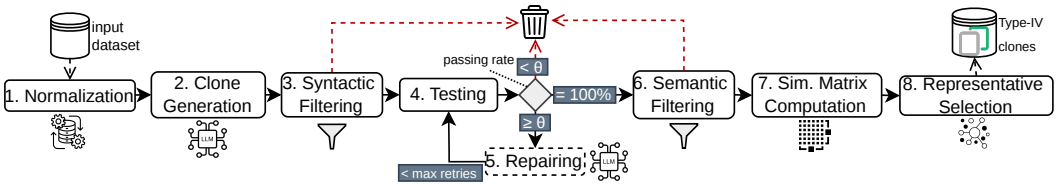


Fig. 2. Overview of our Type-IV clone generation approach.

normalization process extracts and organizes only the elements relevant for clone generation: the original source code of a given task, its corresponding test cases, textual descriptions, and metadata such as expected return values and input parameters. Notice that for our approach, the only mandatory artifact is the test cases, since they are needed for the semantic filtering (see Section 3.3). In addition, our approach generates an abstract syntax tree (AST) for each source function and stores it alongside the normalized data. The AST serves as an alternative representation of the code, which can be used as input to LLMs during clone generation to diversify the produced candidates.

3.2 Clone Generation

The goal of this step is to produce behaviorally equivalent but syntactically diverse implementations of a given source function, i.e., Type-IV clones. This step is central to the approach, as it directly determines the semantic diversity and coverage of the resulting dataset. Two main design requirements guide this stage: (1) ensuring *variety* in the generated code so that ML models trained on the dataset are exposed to diverse realizations of the same functionality; and (2) accommodating *heterogeneous input data*, since not all datasets provide complete metadata such as documentation or test cases. To meet these requirements, a set of prompts is systematically constructed for each normalized dataset entry. Each prompt specifies how an LLM should generate a clone under a given configuration, enabling both controlled experimentation and reproducibility. We define a **prompt configuration** as the combination of (i) the LLM, (ii) the input context, (iii) the prompting strategy, and (iv) a set of refactoring instructions that are provided to the LLM as part of the prompt.

Context: defines the main input provided to the LLM. Contexts are mutually exclusive and include: the original `code`, its `test` cases, a combination of both (`complete`), or the corresponding `AST`. *Prompt strategy:* specifies the prompting style used with the LLM, such as *zero-shot* or chain-of-thought (`CoT`) with few-shot examples [48]. These strategies are mutually exclusive and control the degree of guidance given to the model.

Refactoring: describes transformations that modify the code's syntax or structure without changing its semantics [17]. Refactorings are specified during prompt construction as language-agnostic constraints and are directly provided to the LLM as instructions, which it incorporates when generating the alternative implementation in the target programming language. We rely on seven refactorings inspired by established practices in software engineering [12, 17, 33]: (1) *algorithmic shift*, e.g., using "sorted()" with a key function instead of a manual bubble sort; (2) *library exchange*, e.g., Using 'deque' from 'collections' instead of a list for efficient queue operations; (3) *data shift*, e.g., using a dictionary instead of two separate lists for key-value pairs; (4) *formatting variation*, e.g., adjust indentation or line breaks without changing behavior; (5) *side-effect isolation*, e.g., move printing outside a function that calculates the sum; (6) *security enhancement*, e.g., adding null checks before dereferencing an object; and (7) *bad smell introduction*, e.g., using long, monolithic

Table 1. Prompt template using the code context.

System prompt content
You are a Python generation engine. You produce Python code based on the information given.
User prompt content
You will be shown: 1) A short description. 2) The original function. - Your solution must correspond to this description: `{description}`. - Example solution (you must not use it): `{original_code}` Your task: - Generate an alternative solution named: `{function_name}` - With the following arguments: `{params}`. - Make sure the syntax and structure of your solution are different from the Example solution BODY. - Your solution must return something based on this text: `{return_text}`. - To ensure syntactic and structural differences in your solution, you MUST: `{[refacs]}` - In addition, make sure that: `{mandatory_hints}` `{strategy}`

functions or unclear variable names. Refactorings can be combined to further increase diversity in the generated clones.[1]

Each configuration combines one choice from each dimension, e.g., using the code as context, a CoT strategy, and a subset of refactorings. Table 1 shows the prompt template for the code-based context. Notice that in the prompt content (Table 1), we do not use the term "clone", using instead "alternative solution". The reason for this is our observation during prompt testing: when presented with the term "clone", the LLMs often interpreted it literally (despite instructions to generate Type-IV clones) and produced code that was similar to the original in syntax. Although the approach currently supports a fixed set of contexts, strategies, and refactorings, new options can be easily added to extend coverage.

The `mandatory_hints` section enforces consistent prompt behavior across all configurations, such as restricting the output to valid code only. Despite these constraints, some LLMs may still produce additional text or incomplete snippets. To address this, a post-processing step extracts valid code segments and ensures signature consistency with the original function. These post-processing operations are also applied during the repairing step (see Section 3.4). Finally, the approach supports the use of multiple LLMs. During the generation phase, each LLM is applied across all prompt configurations to encourage diversity. In the repairing step, however, LLMs are selectively used to refine incorrect or incomplete outputs, as described in Section 3.4.

3.3 Syntactic Filtering and Testing

The goal of these steps is to ensure that the generated clones are *syntactically diverse* and *semantically equivalent* to their original counterparts, to be valid Type-IV clones.

Syntactic filtering. Immediately after clone generation, a preliminary filtering mechanism is applied to discard clones that are too similar to the original code. This prevents the inclusion of near-miss (Type-I–III) clones, which would otherwise reduce the diversity and usefulness of the dataset. To quantify syntactic similarity, we adopt CodeBLEU [37], a metric specifically designed for source code that extends BLEU by incorporating structural information derived from program syntax.[2] CodeBLEU decomposes similarity into four components: *n-gram* match, *weighted n-gram* match (emphasizing keywords), *syntax* match based on ASTs, and *data-flow* match capturing variable usage patterns. We focus on the syntactic aspects of the metric by considering only the

[1]Details about the refactorings and prompt templates are available in our replication package [31]

[2]We use the official Python implementation, which derives AST representations using its built-in parser.

n-gram, *weighted n-gram*, and *syntax* components, while excluding the data-flow score. Hence, we call our version CodeBLEU* to differentiate from the standard metric. This choice ensures that the resulting similarity measurement primarily reflects structural information rather than semantic information. The syntactic comparison is performed solely on the function body, as the signature remains fixed across all generated clones. The resulting CodeBLEU* score is appended to each clone's metadata and used to filter out those that exceed a similarity threshold, indicating insufficient syntactic variation.

```python
import random
import collections
def task_func(values, weights, n_samples):
    samples = random.choices(values, weights=weights, k=n_samples)
    histogram = collections.Counter(samples)
    return dict(histogram)
```

Fig. 3. Solution with high CodeBLEU* score compared to Fig. 1a.

Clones with a CodeBLEU* above the threshold are excluded from subsequent stages. Fig. 3 illustrates an example solution with a high CodeBLEU* compared to the primary solution from Fig. 1a. Thus, it would be filtered out despite having minor structural differences. When the original code is unavailable in the input dataset, the syntactic filtering is skipped, and syntactic diversity is later assessed during the representative selection (see Section 3.5). However, if textual descriptions or tests are missing from the dataset, these entries would not be considered.

Testing. Behavioral validation through automated testing is essential to confirm semantic equivalence between the original and generated code. Only clones that pass the syntactic filter proceed to this stage. This process requires the dataset to include executable test cases, which represent the minimum mandatory artifact for applying our approach. Without tests, semantic correctness cannot be automatically verified. The testing phase is fully automated. For each generated clone, a dedicated test file is created that includes all unit tests from the original dataset, which are then executed against the clone implementation. The *semantic filtering* step (step 6) will retain clones that pass all tests, while clones that pass a certain number of tests will be repaired, as described next.

3.4 Repairing Clone Candidates

The goal of this step is to recover near-valid clones that almost satisfy the behavioral equivalence requirement but fail some test cases. These clones already exhibit partial correctness and adequate syntactic diversity, making them valuable candidates for refinement rather than direct rejection. Repairing is an optional step executed when a clone fails some tests but still achieves a test pass rate above a predefined threshold. By focusing on these candidates, the approach iteratively improves clones to increase the number of valid Type-IV clones without regenerating the entire dataset from scratch. This selective strategy not only preserves near-miss solutions that could be corrected but also optimizes computational resources by limiting additional LLM calls to clones with potential. Unlike the clone generation step, repairing uses a single standardized prompt. The prompt[3] includes the clone's current code, the tests list, and the subset of failing tests. Here, prompting strategies or refactoring instructions are not applied, as the objective is not to induce additional diversity but to correct the existing code.

In this step, multiple LLMs are used *collaboratively* rather than independently, in order to leverage their complementary strengths in reasoning and correction. This process consists of two main stages. First, candidate clones are identified based on whether their test execution satisfies a predefined

[3]The prompt used for repairing is available in our replication package [31].

pass-rate threshold. Then, these candidates are refined using available LLMs, where each model iteratively attempts to improve the clone within a limited number of retries. For each repairing attempt, the system (1) generates a new version of the clone using an alternative model, (2) executes the unit tests on the new version, and (3) computes its CodeBLEU* score. If a clone passes all tests and its CodeBLEU* score remains below the syntactic similarity threshold, the new clone replaces the previous version. Otherwise, retries continue until either success is achieved or all models have been exhausted. This process prioritizes efficiency and variety, as once a valid Type-IV clone is obtained, no further repairing is performed for that candidate.

3.5 Representative Selection

The goal of this step is to ensure that the final dataset captures diverse yet non-redundant Type-IV clones. Although all clones at this stage are syntactically distinct from the original code and semantically equivalent, many of them may still be highly similar to each other. Without an explicit mechanism to reduce such redundancy, the resulting dataset could overrepresent certain syntactic patterns, limiting its effectiveness for training and evaluating ML-based clone detectors. To address this issue, this step identifies, for each original code entry, a subset of clones that are mutually distinct while still representative of the overall diversity of syntactic transformations. This process is carried out in two main phases. First, a *syntactic similarity matrix* A is constructed for each set of candidate clones (step 7 in Fig. 2) using their pairwise syntactic CodeBLEU* scores. Each element A_{ij} represents the syntactic similarity between clones i and j:

$$A_{ij} = \begin{cases} 1 & \text{if } i = j, \\ \text{CodeBLEU*}(\text{clone}_i, \text{clone}_j) & \text{otherwise.} \end{cases}$$

Fig. 4 illustrates a clone candidate that passes the syntactic filtering when compared to the original code (Fig. 1a), but it is similar to other clone candidates (Fig. 1b).

```python
import random
from collections import Counter
def task_func(values, weights, n_samples):
    samples = [random.choices(values, weights=weights)[0] for _ in   range(n_samples)]
    histogram = {}
    counts = Counter(samples)
    for value in values:
        histogram[value] = counts.get(value, 0)
    return histogram
```

Fig. 4. Solution with low CodeBLEU* scores compared to Fig. 1a, but high compared to Fig. 1b.

Second, we apply *hierarchical agglomerative clustering* (HAC) [10] to cluster syntactically similar clones based on their pairwise similarities. Given an entry containing n clones, we use the similarity matrix $A \in [0, 1]^{n \times n}$, where each element A_{ij} denotes the syntactic similarity between clones i and j. The matrix is converted into a distance representation $D = 1 - A$, which serves as input to an HAC algorithm using average linkage. Clusters are then formed by cutting the dendrogram at a distance threshold $\tau = 1 - \theta$, where θ corresponds to the CodeBLEU* similarity threshold used during syntactic filtering. Two clones i and j belong to the same cluster C_k if and only if $D_{ij} \leq \tau \iff A_{ij} \geq \theta$. For each cluster C_k, the *representative clone* r_k is defined as the medoid, i.e., the clone with the highest average similarity to all other members of the cluster:

$$r_k = \arg\max_{i \in C_k} \frac{1}{|C_k| - 1} \sum_{\substack{j \in C_k \\ j \neq i}} A_{ij}.$$

Since the syntactic filtering phase already excludes any clone pairs with a CodeBLEU* score above θ, HAC should group only clones whose syntactic similarity meets or exceeds the minimum level we consider meaningful with regard to θ. Thus, the selected representative reflects the most central syntactic form within its cluster while remaining semantically equivalent to the original function. For example, the solutions from Fig. 1b and Fig. 4 would be part of the same cluster. In case these were the only two solutions in this cluster, one would be selected as the representative. By combining CodeBLEU*-based clustering with medoid selection, this step produces a final set of *representative and diverse Type-IV clones* for each original code entry. This strategy minimizes redundancy and ensures that the resulting dataset maintains both behavioral correctness and syntactic diversity, making it a robust foundation for training and fine-tuning ML models for clone detection.

4 Empirical Evaluation

We implemented the approach in Python as a fully automated pipeline, named *Kamino*. The *Kamino* pipeline [31] executes all steps of the approach automatically, given: (1) an input dataset that contains at least a set of tests and function descriptions; (2) the LLMs to be prompted; (3) syntactic and semantic thresholds; and (4) the selected prompt configurations (see Table 2). While the pipeline could have been implemented in other languages, Python was chosen due to its libraries that facilitate working with LLMs (e.g., Ollama), filtering, and clustering methods. The implemented pipeline was used to carry out an empirical evaluation.

4.1 Study Design

The goal of our evaluation is to assess how effectively LLMs generate Type-IV clones across different contexts and to examine the utility of the resulting dataset for fine-tuning embedding models for Type-IV clone detection. The evaluation is guided by research questions (RQ):

RQ1: To what extent can LLMs be leveraged to generate Type-IV clones considering different contexts, prompt strategies, and refactorings?

Rationale: We systematically explored multiple generation configurations to identify those that maximize the syntactic and structural diversity of the produced clones. *Method:* We explored all prompt configurations presented in Table 2 to assess their efficiency in Type-IV clone generation across multiple LLMs (`deepseek-r1:14B`, `gemma3:4B`, `llama3.1:8B`, and `gpt-oss:20B`). These models are open-source and relatively small-scale, thereby supporting reproducibility under practical resource constraints. To induce controlled syntactic diversity, we considered all refactoring categories described in Section 3.2. We constructed seven distinct refactoring combinations, each comprising three refactoring categories, such that all categories are uniformly represented across combinations. Overall, the prompt configurations systematically combined four contexts, two prompting strategies, and the seven refactoring combinations, and were applied to each of the four LLMs, resulting in a total of 224 unique prompt configurations.

The input dataset comprised 992 Python entries from the *easy* split of *BigCodeBench*. We selected *BigCodeBench* because its test suites are widely used to evaluate LLMs [21] and because our approach requires the availability of executable test cases. Moreover, since our objective is to generate a large and diverse set of semantic clones, we focused on the *easy* split, which is obtained by excluding the *hard* subset from the standard *BigCodeBench* and is more amenable to large-scale generation.[4] Before executing our pipeline, we validated that all original implementations pass their associated tests. After this verification step, 927 of the 992 entries remained. For syntactic filtering,

[4]BigCodeBench leaderboard: https://bigcode-bench.github.io/

Table 2. Prompt configurations used in the evaluation.

Configurations.*	Description
Context	
Code	Original code (Table 1).
Test	Unit tests.
Complete	Code and unit tests.
AST	Generated abstract syntax tree (AST).
Strategy	
Zero-shot	No additional information.
CoT	Reasoning with examples of Type-IV clones outside the dataset.
Refactoring Actions	
Refacs.	Three of seven refactoring actions (Section 3.2) randomly chosen. Seven distinct groups ensure all actions appear equally.

* Descriptions, parameters, and return texts are included in all configurations.

we applied a CodeBLEU* threshold of 0.4, which has been shown to be effective for enforcing syntactic divergence [28] after removing function signatures (identical across clones). Moreover, we required at least 75% of the test cases to pass for a generated clone to be considered for repair. This threshold was informed by a pilot study conducted on 50 randomly sampled entries.[5] The same pilot study further indicated that allowing up to two repair attempts yields the best trade-off between success rate and computational cost.

Metrics: We analyzed clone statistics considering CodeBLEU* scores and the percentage of passing tests. We also considered the cluster sizes (bigger clusters mean more similar clones were produced) per context, strategy, refactorings, and LLM. Based on these statistics, we analyzed the most efficient prompt configurations (c.f. Table 2) by calculating the survival rate: Let $N_c^{(k)}$ denote the number of clones surviving after step k. Each configuration c generated an initial set of $N_c^{(0)}$ clones, which were progressively filtered and clustered. Thus, $N_c^{(5)}$ denotes the final representative clones (five steps considered: generation, syntactic filter, semantic filter, repairing, and representative selection). Formally, we define the efficiency of configuration c as the ratio of clones that survived the entire process $E_c = N_c^{(5)} / N_c^{(0)}$.

RQ2: To what extent do test cases and CodeBLEU effectively retain only Type-IV clones?*

Rationale: We examined whether the combination of syntactic filtering and test-based semantic validation is sufficient to reliably exclude non–Type-IV clones from the final dataset. *Method:* To assess whether the final dataset predominantly contained Type-IV clones, we applied *CloneCognition*, an ML-based clone classifier that detects syntactic clones (Types I–III) with an accuracy of up to 87.4% [34]. CloneCognition relies on surface-level similarity features, including syntactic similarity, function name similarity, and lines of code. CloneCognition was not used within the generation pipeline (i.e., instead of CodeBLEU* at Step 3) because the pipeline required a *continuous and lightweight* syntactic similarity measure to enforce controlled syntactic divergence during filtering and repair; CodeBLEU* provides such a score, whereas CloneCognition is a binary classifier designed for clone detection rather than iterative filtering. We adopted a detection threshold of 0.76, which has been shown to be optimal in prior work using this tool [3].

In addition, we performed a manual validation using a sample of clone pairs derived from the dataset. To construct the evaluation sample, we randomly selected 10% of all dataset entries. For each selected entry, we retained the original code and sampled 30% of its associated clones (rounded up, with at least one clone per entry). Each sampled clone was paired with its corresponding

[5] Pilot results are available in our replication package [31]

original implementation to form the evaluation pairs. Each pair was independently inspected by two annotators, who classified the relationship into one of four categories: non-clone, Type I–II, Type III, or Type IV. To reduce ordering effects, each annotator reviewed the full sample in a randomized sequence. Agreement was measured using Cohen's Kappa coefficient, after which disagreements were resolved through discussion to obtain a final consensus.

Metrics: We analyzed the results produced by *CloneCognition*, aiming at the absence of Types I–III clones in the dataset. For the manual validation, we report the distribution of clone categories assigned by annotators (non-clone, Type I–II, Type III, and Type IV), the number of agreements and disagreements between annotators, and Cohen's Kappa coefficient to assess inter-rater reliability.

RQ3: To what extent does fine-tuning embedding models on the generated dataset improve Type-IV clone detection performance and cross-dataset generalization?

Rationale: To assess the practical utility of our pipeline and the resulting dataset, we evaluated whether models fine-tuned on our dataset generalize to unseen Type-IV clones originating from different sources and programming languages. *Method:* For clone detection, we fine-tuned code embedding models using only the generated *Kamino* dataset and evaluated them on two independent benchmarks: *SemanticCloneBench* [2], which contains manually validated clone pairs extracted from Stack Overflow answers in Python, Java, C#, and C, and *GPTCloneBench* [3], which contains LLM-generated Type-IV clone pairs derived from *SemanticCloneBench*. This setup enables us to assess whether the generated dataset supports clone detection for both human-written and LLM-generated code.

For fine-tuning, we used the *Kamino* dataset to construct an equal number of *positive pairs* (clone pairs originating from the same entry) and *negative pairs* (pairs formed from clones belonging to different entries). In total, 72 084 pairs were used for fine-tuning. Entries containing only a single clone were excluded, and only function bodies were considered.

We fine-tuned code embedding models (`CodeBERT` [45] and `CodeT5` [16]) using the *CosineSimilarityLoss* objective from the *SentenceTransformers* framework, which optimizes cosine similarity between embeddings of positive and negative code pairs. This objective pulls embeddings of positive pairs closer while pushing negative pairs further apart. We evaluated different cosine-similarity classification thresholds (θ), only reporting the best-performing results, which were at 0.5. Fine-tuning was performed for 3 epochs with a batch size of 8, a maximum sequence length of 256 tokens, and a warmup phase of 10% of training samples. These values were determined based on tests performed with a sample of the dataset [31]. Since the selected models were not pre-trained on the C programming language, we excluded these pairs from the evaluation.

Metrics: We collected the precision (predicted Type-IV clone pairs that originate from the same entry), recall (true Type-IV clone pairs correctly identified), and F1-score (harmonic mean of precision and recall). Additionally, we compute the Matthews Correlation Coefficient (MCC) [32], which has been shown to provide advantages over F1-score for binary classification tasks as it accounts for true and false positives and negatives [11].

RQ4: To what extent does syntactic diversity impact the usefulness of the generated dataset for downstream Type-IV clone detection?

Rationale: We investigate how datasets composed of more or less syntactically diverse clones impact downstream detection performance. *Method:* We conducted an ablation study using three variants of the generated *Kamino* dataset, each constructed by applying different CodeBLEU*-based similarity constraints during the clustering process. Since lower CodeBLEU* scores indicate stronger syntactic divergence, the variants correspond to increasing levels of syntactic diversity. For this, we derive thresholds empirically from the distribution of pairwise CodeBLEU* scores across all clones, using the 25th and 75th percentiles ($p_{25} = 0.23$, $p_{75} = 0.46$). Thus, the

variants are: (i) `Kamino-LD`: high-similarity range ($0.46 < \texttt{CodeBLEU*} \leq 1.0$); (ii) `Kamino-MD`: medium-similarity range ($0.23 < \texttt{CodeBLEU*} \leq 0.46$); and (iii) `Kamino-HD`: low-similarity range ($0.0 \leq \texttt{CodeBLEU*} \leq 0.23$). All other settings, including negative pairs building, train/test splitting, and hyperparameters, remain identical to RQ3. The embedding models (`CodeBERT` and `CodeT5`) are then fine-tuned independently on each variant and tested on *SemanticCloneBench* and *GPT-CloneBench*.

Metrics: We report F1-score and MCC. In addition, we analyze relative performance variations across the three diversity levels to assess how increasing syntactic divergence affects semantic clone detection performance.

4.2 Evaluation Results

In this section, we present the results for each RQ. All experiments were performed using a Linux server with one NVIDIA RTXA5000 GPU using Ollama v0.14.2 and temperature=0.1, as shown to be an optimal value for clone generation [13]. A replication package is available [31].

RQ1. Type-IV clone generation: Fig. 5 summarizes the CodeBLEU* scores by context, strategy, and refactorings (refacs) per LLM from syntactic filtering (step 3). Context and LLM had the most impact on the scores. Notice that for the `test` context, the scores (including the quartiles) are the lowest, with a median≈0.2. Considering the LLMs, `gpt-oss:20B` achieved the best results with the max CodeBLEU* score never going beyond 0.6. This means that, considering only the results from syntactic filtering, the `test` context and `gpt-oss:20B` would be part of the most efficient combinations. Regarding the results for passing tests (step 4), however, `deepseek-r1:14B` was the best LLM (illustrated in Fig. 6) on most cases. Another impact factor is the `AST` context, which performed poorly with around 40% clones passing the tests in the best cases. The `complete` context was the most efficient in this case, with passing tests above 60% for all LLMs, except `llama3.1:8B`. A total of 31 696 clone candidates passed all tests (semantic filtering). During the representative selection (step 8), cluster sizes ranged from 1 to 111 clones, averaging 5.17 per entry. This lower number indicates that few redundant clones remained at this stage (i.e., clones syntactically similar to each other). Regarding the clones' survival ratio, `gpt-oss:20B` achieved the best overall result (0.364) when combined with the `test` context, `zero-shot`, and the refactorings `algorithmic shift`, `data shift`, and `bad smells`.

Considering each aspect individually, `deepseek-r1:14B` and the `test` context were more often among the most efficient configurations. In contrast, `gemma3:4B` and the `AST` context were among the worst results. While the survivor ratio is low even for the best cases, in the end, the final number of retained clones is acceptable. Considering that from the original 927 *BigCodeBench* entries, 177 401 clones were generated. During the syntactic filtering (step 3) 120 273 clones remained (67.8%). During semantic filtering (step 6) 31 696 clones were kept (17.9%). Ultimately, 6, 123 unique Type-IV clones remained in the dataset (3.5% of the first generation). Still, for 65 entries, no clones remained in the final dataset due to either syntactic or semantic filtering. This led to a total of 862 entries with clones (avg. of ≈7 clones per entry).

RQ1: LLMs can effectively generate Type-IV clones, with efficiency driven by context and the LLM used. The *test* context yielded the best results. `gpt-oss:20B` achieved the best CodeBLEU* scores, while `deepseek-r1:14B` produced the most passing clones.

RQ2. Retention of Type-IV clones: After the clone generation, we constructed all possible combinations of clone pairs for each entry (excluding the original code) and used them as input to CloneCognition. This resulted in a total of 36 042 clone pairs, with an average of 42.43 pairs per

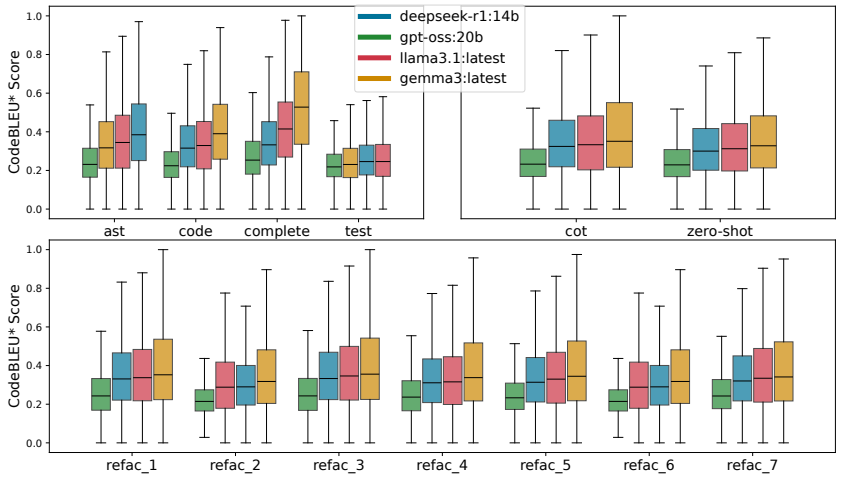


Fig. 5. CodeBLEU* scores (context, strategy, and refacs.).

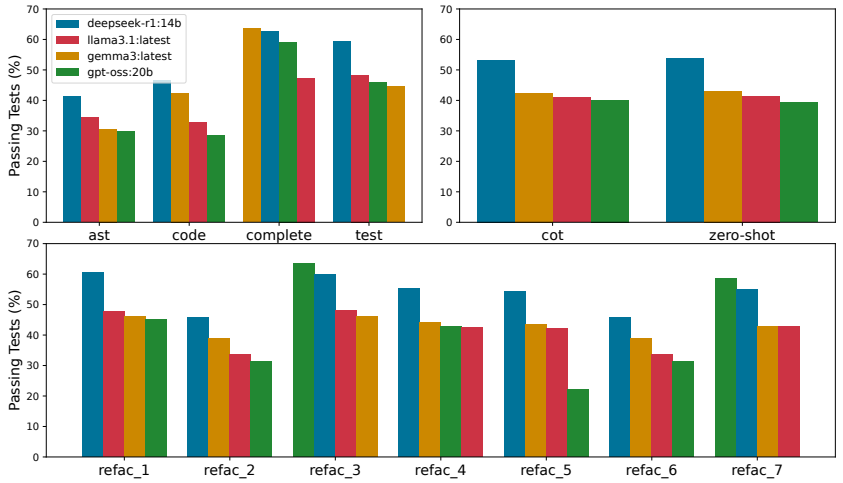


Fig. 6. Passing tests %. (context, strategy, and refacs.)

BigCodeBench entry. Among these pairs, CloneCognition identified 21 as clones (0.06%). Entry 911 from *BigCodeBench* was the most problematic, accounting for 9 non–Type-IV clone pairs. The `algorithm shift` refactoring and `deepseek-r1:14B` were the primary causes, leading to 13 cases each. Nevertheless, the results indicate that the proportion of non–Type-IV clones in the final dataset is negligible.

Considering the manual inspection, the resulting sample contained 231 clone pairs derived from 86 unique entries. On average, 2.69 clones per entry were included (min=1, max=11, std=1.96). Across all the inspected code pairs, annotators agreed directly on 216 cases, indicating a *substantial* level of agreement ($\kappa = 0.74$). Considering the agreement answers, the majority of 192 pairs were labeled as Type-IV, 12 pairs were labeled as Type-III, 10 pairs as Type-I–II, and 2 cases as non-clones. Considering the disagreements, most cases occurred between Type-IV and Type-III clones (13),

Table 3. Clone detection before and after fine-tuning on the *Kamino* dataset (θ = 0.5).

Model	DS	Lang.	Pairs*	Pre-trained				Fine-tuned			
				Prec.	Rec.	F1	MCC	Prec.	Rec.	F1	MCC
CodeBERT	GCB	Java	14,192	0.50	1.00	0.67	0.01	0.93	0.92	**0.93**	**0.86**
		C#	9,816	0.50	1.00	0.67	0.00	0.99	0.94	**0.96**	**0.93**
		Py	5,640	0.50	1.00	0.67	0.00	0.94	0.91	**0.92**	**0.85**
	SCB	Py	2,000	0.50	1.00	0.67	0.00	0.95	0.74	**0.83**	**0.72**
		Java	1,992	0.50	1.00	0.67	0.02	0.96	0.71	**0.81**	**0.70**
		C#	1,870	0.50	1.00	0.67	0.00	0.99	0.71	**0.82**	**0.73**
CodeT5	GCB	Java	14,192	0.53	0.91	0.67	0.14	0.90	0.92	**0.91**	**0.81**
		C#	9,816	0.53	0.91	0.67	0.13	0.97	0.95	**0.96**	**0.92**
		Py	5,640	0.52	0.92	0.67	0.11	0.85	0.91	**0.88**	**0.75**
	SCB	Py	2,000	0.51	1.00	0.67	0.13	0.87	0.81	**0.84**	**0.69**
		Java	1,992	0.53	0.95	0.68	0.16	0.91	0.72	**0.80**	**0.66**
		C#	1,870	0.51	0.95	0.66	0.07	0.96	0.74	**0.83**	**0.72**

* 50/50 negative/positive. GCB = GPTCloneBench; SCB = SemanticCloneBench.

reflecting the inherent difficulty in distinguishing higher-level semantic similarity in borderline cases.

RQ2: Our pipeline successfully generated 99.9% Type-IV clone pairs (according to CloneCognition results). Manual inspection of 231 sampled pairs further supports this finding, showing substantial agreement between annotators (κ = 0.74) and confirming that 192 pairs (83%) were Type-IV clones.

RQ3. Using the dataset for clone detection: Table 3 shows that fine-tuning embedding models with the *Kamino* dataset consistently improves clone detection performance. First, considering the results for the *GPTCloneBench* (GCB in Table 3), both F1 and MCC results are better for detection on Python, Java, and C# clones. Despite the fact that our training data only had Python clones, the best results were achieved for C#, where both `CodeBERT` and `CodeT5` reach F1 of 0.96 and MCC of 0.92. These results show that fine-tuning on Python clones also benefits the other languages, demonstrating that the generated clones capture language-agnostic semantic properties.

Moreover, considering the results for the *SemanticCloneBench* (SCB), both F1 and MCC values were lower than those obtained for *GPTCloneBench* across all programming languages. This difference is probably due to SCB containing human-written Type-IV clones, which are syntactically more different. Even so, the fine-tuned models still achieved strong performance, with F1 scores above 0.80 for all evaluated languages. Furthermore, the lower MCC values are mainly associated with a larger number of false negatives, indicating that several true clone pairs received cosine similarity values below the selected threshold (θ = 0.5). This suggests that using a lower threshold could improve recall by identifying more true positives, although potentially at the cost of reducing precision due to an increase in false positives.

In summary, the detection results demonstrate that the produced clones can improve and make the predictions of embedding models more consistent when considering semantic similarity, even for programming languages that were not used during fine-tuning. However, the fine-tuned models still struggle with the SCB dataset, suggesting that more syntactically diverse human-written clones may require lower similarity thresholds (θ) to achieve better recall and overall detection performance.

RQ3: Fine-tuning with the generated dataset improves F1 and MCC scores across unseen functionalities in Python, Java, and C#. Improvements, however, are more limited on the more syntactically diverse SemanticCloneBench dataset.

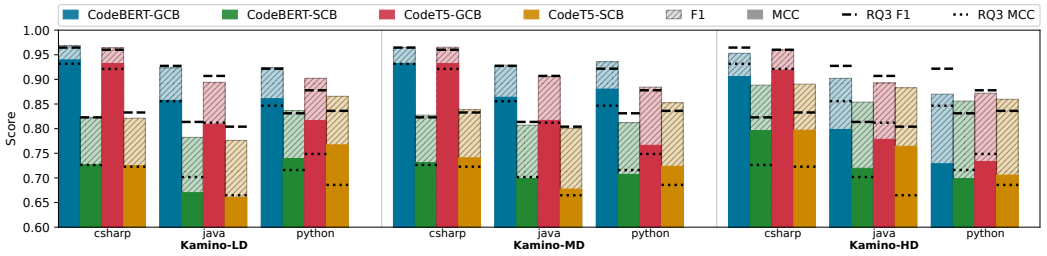


Fig. 7. Performance across syntactic training dataset variants compared to RQ3 ($\theta = 0.5$).

RQ4. Impact of syntactic diversity: Considering the dataset variants, `Kamino-LD` is the densest (5,332 clones, 10.37 per entry, 49,546 pairs), `Kamino-MD` is more balanced (7,119 clones, 8.54 per entry, 43,064 pairs), and `Kamino-HD` is the sparsest (1,477 clones, 1.71 per entry, 1,552 pairs). Figure 7 summarizes the results for clone detection across the three variants, compared against the RQ3 baseline. When analyzing the test datasets separately, the high-diversity variant produced the most consistent gains on *SemanticCloneBench* trained with `CodeT5`, with improvements reaching +15% MCC (0.76 vs. 0.66 from RQ3) and +9.8% F1 (0.88 vs. 0.80 from RQ3) for Java, and +10% MCC (0.79 vs. 0.72 from RQ3) and +7.9% F1 (0.89 vs. 0.83 from RQ3) for C#, suggesting that syntactically diverse training pairs help models better capture semantic equivalence in human-written code. In contrast, results on *GPTCloneBench* were more variable, with some of the largest degradations observed for `Kamino-HD`, especially for `CodeBERT` on Java and Python. These degradations, however, are probably related to the considerably smaller size of `Kamino-HD`. When analyzing the embedding models independently, `CodeT5` benefited more strongly from syntactic diversity than `CodeBERT`. Across all variants, `CodeT5` achieved average gains of +1.18% in F1-score and +3.16% in MCC, whereas `CodeBERT` showed nearly neutral F1 variation (+0.14%) and a slight MCC degradation (-0.45%). This suggests that encoder-decoder architectures such as `CodeT5` may better leverage semantically equivalent but syntactically distant examples.

The detailed comparison further shows that improvements were more frequent than degradations. Considering all configurations, 20 cases improved F1-score while 16 degraded, with average gains (+2.49%) exceeding the average degradation (-1.63%). Similarly, MCC improved in 24 cases and degraded in 12, with average positive changes (+3.68%) slightly larger than the average negative changes (-3.30%). These results indicate that syntactic diversity generally contributes positively to semantic clone detection robustness.

RQ4: Syntactic diversity leads to moderate improvements in Type-IV clone detection, with *Kamino-HD* achieving the strongest gains (up to +15% MCC and +9.8% F1 for Java), although effects are not consistent across datasets, as *GPTCloneBench* shows degradations under *Kamino-HD* (down to -5.61% F1 and -13.91% MCC for Python).

5 Discussion

In this section, we discuss the evaluation results, reflect on their implications, and report threats to validity.

5.1 Observations and Limitations

Main driving forces: To better understand which configuration aspects most influence clone generation success, we analyzed the individual contributions of each dimension, i.e., LLM, context,

refactorings, and prompting strategy, using group-wise aggregation combined with a random forest model. The results [31] indicate that among the LLMs, `deepseek-r1:14B` was the main driver of success overall, followed by `gpt-oss:20B` with comparable scores. The `test` context yielded the best results, closely followed by `complete`, whereas the `AST` context produced significantly lower performance. Interestingly, zero-shot prompting outperformed few-shot CoT, suggesting that providing examples may lead LLMs to reproduce syntactically closer solutions with higher CodeBLEU* scores. Regarding refactorings, most combinations showed limited impact; however, removing them entirely increased syntactic similarity, confirming that they promote diversity. These findings reinforce the results of RQ1 and highlight the central role of prompt composition and LLM choice in effective clone generation.

Moreover, for 65 *BigCodeBench* entries, no clones remained after filtering. In these cases, either the entries contained numerous tests, leading clones to fail at least one, or the tasks were inherently complex. For instance, Entry 566 required extracting a function's name, type, and arguments. Addressing these cases could involve fine-tuning LLMs on validated clones or iteratively repairing unsuccessful candidates after clustering. Another option would be to keep entries failing a low number of tests and provide them to a user for a possible fix.

Number of distinct functionalities: Although the *Kamino* dataset contains 42 698 Type-IV clone pairs (including the original code) derived from 862 *BigCodeBench* [54] entries, these pairs may originate from a limited set of predefined functionalities. For example, *SemanticCloneBench* consists of 4 000 manually validated clone pairs extracted from Stack Overflow answers across Java, C#, C, and Python, covering 2 632 functionality-specific groups (477–838 per language). *GPTCloneBench* extends this dataset by generating additional synthetic Type-IV clone pairs using GPT-3, resulting in 18 269 pairs. A key distinction between these datasets lies in their construction. Kamino allows for controlled and scalable generation of semantically diverse clone pairs with explicit functional grounding. In contrast, *SemanticCloneBench* is derived from real-world developer discussions without an explicit functional taxonomy, while *GPTCloneBench* augments it using LLM-based generation while preserving its structure. Consequently, relying on *Kamino* for training may introduce the risk of models overfitting to task-specific patterns rather than learning general semantic representations [24]. To mitigate potential dataset overlap, we evaluate models fine-tuned on *Kamino* using *SemanticCloneBench* and *GPTCloneBench*, and manual inspection of sampled pairs did not reveal any clear overlap between *Kamino* and either benchmark. Moreover, the performance improvements across these datasets (RQ3, RQ4) suggest that the learned representations generalize beyond the specific functionalities present in the training data. Therefore, we position *Kamino* not as a replacement for real-world datasets such as *SemanticCloneBench*, but as a scalable approach for generating large quantities of behaviorally validated Type-IV clone pairs that complement existing benchmarks.

Extending and improving generation: While exploring all prompt configurations may lead to a larger number of generated clones, an alternative approach is to restrict generation to the most effective configurations in order to optimize performance. We investigate this possibility by selecting the top 50 configurations identified in RQ1 and applying them to 50 entries randomly sampled from *BigCodeBench*. The results [31] show that this strategy yields fewer clones on average (2.68 per entry), with 2 500 clones generated after Step 2, of which 1 559 (62.4%) remain after Step 3, 804 (32.2%) after Step 6, and 134 (5.4%) after Step 8.[6] Although the final number is relatively small, it could be increased by considering a larger set of effective configurations (e.g., the top 100 instead of the top 50). However, using more configurations will increase resource consumption. In RQ1, we used open-source LLMs executed on a single GPU. Running the full pipeline on the 927 *BigCodeBench*

[6]Results for this exploration are available in our replication package [31]

entries required approximately three weeks, which further motivates focusing on the most effective configurations. Investigating an appropriate trade-off remains as future work.

Moreover, expanding the framework with new LLMs, contexts, or prompt strategies can further enhance diversity and coverage. However, relying exclusively on CodeBLEU* as a syntactic filter may limit generalization. Applying the generation at the file level or across programming languages will likely require alternative metrics. Promising options include normalized ASTs with tree edit distances [42], vector-based similarity measures [22], CrystalBLEU [15], which improves CodeBLEU by discounting trivial shared n-grams, and graph-distance metrics on enriched code representation structures [47]. Combining these metrics could provide more robust syntactic filtering, which could, for example, eliminate the non-Type-IV clones that were still identified in the final dataset (RQ2).

Improving clone detection: Generating diverse clones is insufficient if they fail to represent genuine Type-IV equivalence. RQ3's results demonstrate the usefulness of the generated dataset when evaluated with code embeddings: fine-tuning on these data leads to measurable precision gains. RQ4 further refines these findings by showing that the degree of syntactic diversity has an impact on downstream performance. In particular, moderate-to-high diversity improves Type-IV clone detection, especially on *SemanticCloneBench*, where gains are both consistent and substantial (up to +15% MCC and +9.8% F1). This suggests that syntactically divergent examples are particularly effective for improving model generalization on human-written code. However, the benefits are not uniform as the results for *GPTCloneBench* show more variability and occasional degradations. This is also related to the smaller size of the high-diversity dataset compared to the others, indicating that model performance is sensitive to both dataset structure and scale. Overall, RQ4 confirms that diversity is beneficial but must be balanced with dataset size and stability to maximize effectiveness.

Further improvements could stem from extending the framework to multi-language clone generation, which would enrich the syntactic and semantic diversity of training data. Such diversity is essential for using non-contrastive self-supervised learning approaches, e.g., VICReg [5], enabling better invariance learning and more accurate cross-language clone detection. Furthermore, since the primary contribution of this paper is the generation of a large and diverse set of Type-IV clones, we intentionally do not emphasize baseline comparisons in RQ3. Hence, the objective of RQ3 is not to claim that embedding models represent the most effective mechanism for clone detection. Rather, our aim is to demonstrate that access to a large, high-quality dataset of verified Type-IV clones can substantially improve the performance of ML-based approaches on the clone detection task. Recent studies further indicate that even LLMs struggle with Type-IV clone detection, in part due to the lack of sufficiently large and diverse training data [23, 52]. Our results suggest that the dataset generated by our approach can help mitigate this limitation. Improvements observed for Java and C# suggest that cross-language transfer is more effective when languages are syntactically and semantically similar. Programming paradigms may also have an impact. Beyond benchmarking, we also envision practical applications of the proposed dataset for training models aimed at detecting semantic duplication in LLM-generated code. In this context, our dataset can support the development of models capable of identifying semantic equivalence across diverse implementations and programming styles.

Applications beyond Type-IV clones: Although our work focuses on Type-IV clones due to their complexity and scarcity, the same framework can be adapted to other clone categories. Increasing the CodeBLEU* threshold allows Types I–III to be retained during filtering. Moreover, lowering the semantic constraint, requiring less than 100% test passing, would keep clones with similar yet distinct behaviors, which is useful for detecting code duplication or refactoring opportunities. Exploring these extensions represents a promising avenue to broaden automated clone generation. Additionally, the embedding models fine-tuned on our dataset should be capable of detecting all

clone types, as they capture code semantics. Since this was not our focus, we did not evaluate these models on Type I–III clones. Exploring this direction is left for future work.

5.2 Threats to Validity

The selection of LLMs (RQ1) may introduce *internal validity* threats [14]. To mitigate this, we selected well-known open-source LLMs recognized for high code-generation performance. The choice of contexts also poses a limitation: we used configurations that ensure broad applicability, such as the `test` context for datasets containing only test cases. Additional contexts (e.g., requirement lists) could easily be integrated by extending the base prompt templates. Another potential threat arises from RQ3 embedding models; to mitigate this, we relied on widely adopted models that have shown robustness in prior code embedding studies [6, 25]. The results in Table 3 confirm that fine-tuning significantly improves precision. For RQ2, CloneCognition reported an accuracy of up to 87.4% [34], indicating that our observed 99.9% Type-IV clone rate may differ, as the tool is not an absolute oracle. To mitigate this, we also performed a manual evaluation on a sample from the dataset. Another threat arises in RQ4 from the selection of similarity thresholds used to partition clones into syntactic-diversity variants, as arbitrary boundaries could bias performance comparisons across tiers. We mitigate this by deriving thresholds empirically from the distribution of pairwise CodeBLEU* scores across all clones, using the 25th and 75th percentiles ($p_{25} = 0.23$, $p_{75} = 0.46$) as natural breakpoints rather than manually chosen values. This ensures that each variant (`Kamino-LD`, `Kamino-MD`, `Kamino-HD`) captures a statistically grounded and roughly equal portion of the similarity distribution, reducing the risk of artificially inflating or deflating performance differences between tiers..

Concerning *external validity*, the dataset selected for RQ1 may not fully represent real-world tasks. We mitigated this by using *BigCodeBench*, a well-known and widely used dataset for LLM code generation tasks. A potential threat is functionality overlap between training and test splits, which could inflate downstream performance. We measured cross-split semantic similarity of *BigCodeBench* task descriptions using sentence-transformer embeddings (`all-MiniLM-L6-v2`) and cosine similarity, obtaining low similarity (mean = 0.16, std = 0.14), suggesting limited overlap. A threat to *construct validity* relates to our reliance on CodeBLEU* and test passing as proxies for syntactic and semantic equivalence. These metrics may not fully capture clone quality across languages or at a larger granularity (e.g., file-level). We mitigated this by using empirically validated thresholds [28] and by introducing a repair step for partially successful clones, reducing the likelihood of false positives. Furthermore, we investigated the retention of only Type-IV clones with RQ2, showing that the pipeline is able to achieve this goal. Another threat arises from the test infrastructure of *BigCodeBench*. The number of test cases per task ranges from 3 to 15 (mean = 5.64, median = 5), indicating moderate and fairly consistent coverage. We rely only on publicly available tests, which may limit corner-case detection and the strictness of functional correctness evaluation. For *conclusion validity*, the limited number of prompt configurations and datasets may influence statistical reliability. We mitigated this by evaluating 224 configurations. Additionally, we controlled LLM stochasticity by fixing a consistent temperature (0.1) [13]. Furthermore, all thresholds used were decided based on the analysis of the results of a pilot study [31] conducted with 50 entries randomly sampled from *BigCodeBench*. We chose thresholds that had a balance between clone quality and variety (number of unique clones per entry).

6 Related Work

In this section, we review prior work relevant to our study.

6.1 Code Generation with LLMs

LLMs have increasingly demonstrated strong capabilities across diverse software engineering tasks, from code synthesis and completion [6] to debugging [9] and translation [19]. Beyond direct code synthesis, other studies focus on improving the reliability and interpretability of LLM-driven generation processes. Xu et al. [49] introduce *Reprompting*, an iterative algorithm that automatically discovers and refines effective CoT reasoning steps for LLMs through a sampling and feedback mechanism. While their work targets reasoning tasks rather than code generation, their iterative refinement process parallels the repairing mechanism in our approach, where failing outputs are re-evaluated and regenerated until functional correctness is achieved (see Section 3.4). Similarly, frameworks like CodableLLM [30] align low-level (binary) and high-level (source) code representations to improve dataset quality for downstream software analysis tasks, highlighting a broader trend toward integrating LLMs in semantically grounded program understanding pipelines.

6.2 Clone Generation and Detection Approaches

Clone detection has evolved from purely syntactic to increasingly semantic representations. Traditional methods relied on token- or tree-based matching, but later approaches began leveraging structural and semantic information extracted from ASTs and Program Dependency Graphs (PDGs). Sheneamer and Kalita [41] combined AST- and PDG-based representations with ML classifiers to detect complex Type-III-IV clones. Their results demonstrated that semantic representations outperform traditional metrics-based detection, emphasizing the need for semantic-focused datasets. More recent frameworks integrate multiple analytical modalities into unified ensembles. Bhaskar and Geetika [7] propose an adaptive, multi-layered clone detection system that combines token-, AST-, PDG-, and ML-based methods, including LLMs. Their ensemble weighting strategy achieves high accuracy, particularly for challenging Type-IV and cross-language clones, and demonstrates that hybrid systems can robustly handle the diversity of real-world codebases. Nonetheless, these partially ML-based approaches remain constrained by the availability of diverse, functionally verified training data, precisely the gap our work aims to address by automatically generating high-quality semantic clone datasets. Alternatively, efforts such as InspectorClone [40] improve reproducibility through semi-automated precision studies combining metric-based filtering with human validation. These methods, however, remain dependent on manual oversight, limiting scalability and consistency. Overall, these works reveal a clear focus on semantically aware and evaluation-driven clone detection, thus highlighting the importance of scalable and automated mechanisms to generate and validate Type-IV clones, such as our approach.

6.3 Type-IV Clone Generation and Datasets

Zhang et al. [51] introduce *CLONEGEN*, a reinforcement learning-based framework that generates semantically preserved code transformations to evaluate clone detector robustness by producing adversarial Type-IV clones that expose model vulnerabilities. While effective for stress-testing models, CLONEGEN focuses on generating adversarial rather than diverse or validated clones, and relies on hand-crafted transformations rather than adaptive LLM-based synthesis. In contrast, Li et al. [28] propose a framework that leverages LLMs to synthesize semantically and syntactically varied code variants across four different categories, including Type-IV clones. Their approach, however, only generates four versions of the code per entry (one per category). This means that only one Type-IV clone is generated per entry, reducing the variety specifically necessary for training ML-based approaches. Furthermore, their approach is not fully automated like ours, requiring human intervention as the first filter for selecting potential candidates.

Alam et al. [3] introduced GPTCloneBench, a dataset of code clones generated using GPT-3 containing Types–I–IV, and cross-language clones. However, while their approach includes filtering for syntactic similarity, the validity of Type-IV clones is assessed only through manual evaluation. This limits the reproducibility and scalability of the approach for large-scale Type-IV clone generation. Furthermore, their dataset comprises a lower number of true Type-IV clone pairs (e.g., at most around 7k Type-IV clone pairs for Java) when compared to ours (56k pairs). Eagal et al. [13] empirically studied the reliability of LLMs (GPT-3.5, GPT-4, and CodeLlama) in generating Type-IV clones across C++, Java, and Python. They found that LLMs can produce behaviorally consistent clones at low temperatures within the same language, but performance drops for harder problems, higher temperatures, and cross-language scenarios. These findings motivated our use of the *BigCodeBench* easy split and a temperature of 0.1. Additionally, their prompts were simpler than ours, which may explain their limited generation. For instance, they used the term "clone" in their prompts, which we observed can lead LLMs to produce overly similar code. For this reason, we used the term "alternative solution" instead. Other complementary efforts [2] adopt a crowdsourced strategy, leveraging Stack Overflow examples to approximate functional similarity between code fragments. Although this approach yields diverse data, it relies on community consensus rather than executable verification, leading to noisy and inconsistently validated clones. Together, these works reveal the growing importance of automated and semantically grounded code clone generation.

Our work differs from prior clone-generation approaches by combining fully automated LLM-based synthesis, deterministic execution-based verification (tests), iterative repairing, and controlled syntactic-diversity filtering (CodeBLEU*) within a single scalable pipeline. In contrast to existing methods that rely on manual validation, hand-crafted transformations, or generate only a limited number of clones per functionality, our approach explicitly targets the large-scale construction of diverse and functionally verified Type-IV clone datasets for downstream ML training and evaluation.

7 Conclusion and Future work

In this paper, we present a hybrid approach for generating Type-IV clones using LLMs in combination with validation and clustering. Through an extensive empirical evaluation, we showed that the *choice of LLM* and *context* are the main factors driving clone generation efficiency. Moreover, the generated Python clones were validated with CloneCognition and manual validation to ensure that the dataset was compose of Type-IV clones. The resulting dataset proved valuable for fine-tuning code embedding models to detect clones in Python, Java, and C#. Furthermore, the syntactic diversity generally improved Type-IV clone detection, particularly for `CodeT5` on *SemanticCloneBench*, though results were less consistent for *GPTCloneBench*.

For future work, we plan to incorporate prompt configurations to increase the clone diversity. Moreover, alternative syntactic and semantic metrics will be considered at a larger granularity or across multiple programming languages. Fine-tuning LLMs on previously successful clones during the repairing step could improve robustness and overall clone generation efficiency, thus addressing edge cases. Finally, we plan to expand our Type-IV clone dataset by generating clones from different input datasets, including the "hard" split from BigCodeBench. These advances will allow us to employ more advanced ML-based methods for clone detection.

Acknowledgements

This work was partially funded by the Natural Sciences and Engineering Research Council of Canada (NSERC), grants number RGPIN-2025-05677 and RGPIN-2026-06532.

Data Availability

A replication package is available at our online repository [31].

References

[1] Qurat Ul Ain, Wasi Haider Butt, Muhammad Waseem Anwar, Farooque Azam, and Bilal Maqbool. 2019. A Systematic Review on Code Clone Detection. *IEEE Access* 7 (2019), 86121–86144. doi:10.1109/ACCESS.2019.2918202

[2] Farouq Al-Omari, Chanchal K. Roy, and Tonghao Chen. 2020. SemanticCloneBench: A Semantic Code Clone Benchmark using Crowd-Source Knowledge. In *2020 IEEE 14th International Workshop on Software Clones (IWSC)*. 57–63. doi:10.1109/IWSC50091.2020.9047643

[3] Ajmain I. Alam, Palash R. Roy, Farouq Al-Omari, Chanchal K. Roy, Banani Roy, and Kevin A. Schneider. 2023. GPTCloneBench: A comprehensive benchmark of semantic clones and cross-language clones using GPT-3 model and SemanticCloneBench. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 1–13. doi:10.1109/ICSME58846.2023.00013

[4] Sebastian Baltes, Florian Angermeir, Chetan Arora, Marvin Muñoz Barón, Chunyang Chen, Lukas Böhme, Fabio Calefato, Neil Ernst, Davide Falessi, Brian Fitzgerald, Davide Fucci, Marcos Kalinowski, Stefano Lambiase, Daniel Russo, Mircea Lungu, Lutz Prechelt, Paul Ralph, Rijnard van Tonder, Christoph Treude, and Stefan Wagner. 2025. Guidelines for Empirical Studies in Software Engineering involving Large Language Models. doi:10.48550/arXiv.2508.15503

[5] Adrien Bardes, Jean Ponce, and Yann LeCun. 2022. VICReg: Variance-Invariance-Covariance Regularization for Self-Supervised Learning. arXiv:2105.04906 [cs.CV] https://arxiv.org/abs/2105.04906

[6] Oussama Ben Sghaier and Houari Sahraoui. 2024. Improving the Learning of Code Review Successive Tasks with Cross-Task Knowledge Distillation. *Proc. ACM Softw. Eng.* 1, FSE, Article 49 (July 2024), 21 pages. doi:10.1145/3643775

[7] P Vijay Bhaskar and Geetika. 2024. A Comprehensive Analysis of Unified Approaches for Revealing Code Clone Detection. In *2024 4th International Conference on Ubiquitous Computing and Intelligent Information Systems (ICUIS)*. 946–953. doi:10.1109/ICUIS64676.2024.10866000

[8] S Carter, RJ Frank, and DSW Tansley. 1993. Clone detection in telecommunications software systems: A neural net approach. In *Proceedings of the International Workshop on Applications of Neural Networks to Telecommunications*. Psychology Press, 273–280.

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, et al. 2021. Evaluating Large Language Models Trained on Code. In *Advances in Neural Information Processing Systems (NeurIPS)*. doi:10.48550/arXiv.2107.03374

[10] Anshuman Chhabra and Prasant Mohapatra. 2022. Fair Algorithms for Hierarchical Agglomerative Clustering. In *2022 21st IEEE International Conference on Machine Learning and Applications (ICMLA)*. 206–211. doi:10.1109/ICMLA55696.2022.00036

[11] Davide Chicco and Giuseppe Jurman. 2020. The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. *BMC genomics* 21, 1 (2020), 6. doi:10.1186/s12864-019-6413-7

[12] Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien N. Nguyen. 2007. Refactoring-aware Configuration Management for Object-Oriented Programs. In *Proceedings of the 29th International Conference on Software Engineering*. IEEE, 271–280. doi:10.1109/ICSE.2007.71

[13] Azeeza Eagal, Kathryn T. Stolee, and John-Paul Ore. 2025. Analyzing the dependability of Large Language Models for code clone generation. *Journal of Systems and Software* 230 (2025), 112548. doi:10.1016/j.jss.2025.112548

[14] Steve Easterbrook. 2007. Empirical Research Methods for Software Engineering. In *Twenty-Second IEEE/ACM International Conference on Automated Software Engineering* (Atlanta, Georgia, USA) *(ASE '07)*. Association for Computing Machinery, New York, NY, USA, 574. doi:10.1145/1321631.1321749

[15] Aryaz Eghbali and Michael Pradel. 2022. CrystalBLEU: precisely and efficiently measuring the similarity of code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12. doi:10.1145/3551349.3556903

[16] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:2002.08155 [cs.CL] https://arxiv.org/abs/2002.08155

[17] Martin Fowler. 2018. *Refactoring: improving the design of existing code.* Addison-Wesley Professional.

[18] Mark Gabel, Lingxiao Jiang, and Zhendong Su. 2008. Scalable detection of semantic clones. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. 321–330. doi:10.1145/1368088.1368132

[19] Aneri Gandhi, Sanjukta De, Marsha Chechik, Vinay Pandit, Max Kiehn, Matthieu Chan Chee, and Yonas Bedasso. 2025. Automated Codebase Reconciliation using Large Language Models. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*. IEEE, 1–11. doi:10.1109/Forge66646.2025.00011

[20] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. doi:10.48550/arXiv.1909.09436

[21] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2026. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology* 35, 2 (2026), 1–72. doi:10.1145/3747588

[22] Yusuf Kartal, E. Kaan Akdeniz, and Kemal Özkan. 2024. Automating modern code review processes with code similarity measurement. *Information and Software Technology* 173 (2024), 107490. doi:10.1016/j.infsof.2024.107490

[23] Mohamad Khajezade, Jie JW Wu, Fatemeh Hendijani Fard, Gema Rodriguez-Perez, and Mohamed Sami Shehata. 2024. Investigating the Efficacy of Large Language Models for Code Clone Detection. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension* (Lisbon, Portugal) *(ICPC '24)*. Association for Computing Machinery, New York, NY, USA, 161–165. doi:10.1145/3643916.3645030

[24] Konstantinos Kitsios, Francesco Sovrano, Earl T. Barr, and Alberto Bacchelli. 2025. Detecting Semantic Clones of Unseen Functionality. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1312–1324. doi:10.1109/ASE63991.2025.00112

[25] Aleksandar Kovačević, Nikola Luburić, Jelena Slivka, Simona Prokić, Katarina-Glorija Grujić, Dragan Vidaković, and Goran Sladić. 2024. Automatic detection of code smells using metrics and CodeT5 embeddings: a case study in C#. *Neural Computing and Applications* 36, 16 (2024), 9203–9220. doi:10.1007/s00521-024-09551-y

[26] Jens Krinke and Chaiyong Ragkhitwetsagul. 2022. BigCloneBench Considered Harmful for Machine Learning. In *2022 IEEE 16th International Workshop on Software Clones (IWSC)*. 1–7. doi:10.1109/IWSC55060.2022.00008

[27] Jens Krinke and Chaiyong Ragkhitwetsagul. 2025. How the Misuse of a Dataset Harmed Semantic Clone Detection. arXiv:2505.04311 [cs.SE] https://arxiv.org/abs/2505.04311

[28] Zhuohao Li, Wenqing Chen, Jianxing Yu, and Zhichao Lu. 2026. Functional consistency of LLM code embeddings: A self-evolving data synthesis framework for benchmarking. *Expert Systems with Applications* 298 (2026), 129523. doi:10.1016/j.eswa.2025.129523

[29] Kaifeng Lyu, Haoyu Zhao, Xinran Gu, Dingli Yu, Anirudh Goyal, and Sanjeev Arora. 2024. Keeping LLMs Aligned After Fine-tuning: The Crucial Role of Prompt Templates. In *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (Eds.), Vol. 37. Curran Associates, Inc., 118603–118631. doi:10.52202/079017-3766

[30] Dylan Manuel and Paul Rad. 2025. CodableLLM: Automating Decompiled and Source Code Mapping for LLM Dataset Generation. *arXiv preprint arXiv:2507.22066* (2025). doi:10.48550/arXiv.2507.22066

[31] Luciano Marchezan, Eugene Syriani, Kevin Delcourt, and Houari Sahraoui. 2026. *Automated Type-IV Clone Generation via LLMs and Deterministic Validation (replication package)*. doi:10.5281/zenodo.18391578

[32] Brian W Matthews. 1975. Comparison of the predicted and observed secondary structure of T4 phage lysozyme. *Biochimica et Biophysica Acta (BBA)-Protein Structure* 405, 2 (1975), 442–451.

[33] T. Mens and T. Tourwe. 2004. A survey of software refactoring. *IEEE Transactions on Software Engineering* 30, 2 (2004), 126–139. doi:10.1109/TSE.2004.1265817

[34] Golam Mostaeen, Banani Roy, Chanchal K. Roy, Kevin Schneider, and Jeffrey Svajlenko. 2020. A machine learning based framework for code clone validation. *Journal of Systems and Software* 169 (2020), 110686. doi:10.1016/j.jss.2020.110686

[35] Premraj Pawade, Mohit Kulkarni, Shreya Naik, Aditya Raut, and K.S. Wagh. 2024. Efficiency Comparison of Dataset Generated by LLMs using Machine Learning Algorithms. In *2024 International Conference on Emerging Smart Computing and Informatics (ESCI)*. 1–6. doi:10.1109/ESCI59607.2024.10497340

[36] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. doi:10.48550/arXiv.2105.12655

[37] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. arXiv:2009.10297 [cs.SE] https://arxiv.org/abs/2009.10297

[38] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society* 74, 2 (1953), 358–366.

[39] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming* 74, 7 (2009), 470–495. doi:10.1016/j.scico.2009.02.007

[40] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Di Yang, Pedro Martins, Hitesh Sajnani, Pierre Baldi, and Cristina V. Lopes. 2019. Towards Automating Precision Studies of Clone Detectors. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 49–59. doi:10.1109/ICSE.2019.00023

[41] Abdullah Sheneamer and Jugal Kalita. 2016. Semantic Clone Detection Using Machine Learning. In *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 1024–1028. doi:10.1109/ICMLA.2016.0185

[42] Yewei Song, Cedric Lothritz, Daniel Tang, Tegawendé Bissyandé, and Jacques Klein. 2024. Revisiting Code Similarity Evaluation with Abstract Syntax Tree Edit Distance. (Aug. 2024), 38–46. doi:10.18653/v1/2024.acl-short.3

[43] Jeffrey Svajlenko and Chanchal K. Roy. 2022. BigCloneBench: A Retrospective and Roadmap. In *2022 IEEE 16th International Workshop on Software Clones (IWSC)*. 8–9. doi:10.1109/IWSC55060.2022.00009

[44] Robert Tairas and Jeff Gray. 2012. Increasing clone maintenance support by unifying clone detection and refactoring activities. *Information and Software Technology* 54, 12 (2012), 1297–1307. Special Section on Software Reliability and Security. doi:10.1016/j.infsof.2012.06.011

[45] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. 8696–8708 pages. doi:10.18653/v1/2021.emnlp-main.685

[46] Yuekun Wang, Yuhang Ye, Yueming Wu, Weiwei Zhang, Yinxing Xue, and Yang Liu. 2023. Comparison and Evaluation of Clone Detection Techniques with Different Code Representations. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 332–344. doi:10.1109/ICSE48619.2023.00039

[47] Martin Weyssow, Houari Sahraoui, and Bang Liu. 2022. Better modeling the programming world with code concept graphs-augmented multi-modal learning. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results* (Pittsburgh, Pennsylvania) *(ICSE-NIER '22)*. Association for Computing Machinery, New York, NY, USA, 21–25. doi:10.1145/3510455.3512771

[48] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. In *Proceedings of the 30th Conference on Pattern Languages of Programs* (Monticello, IL, USA) *(PLoP '23)*. The Hillside Group, USA, Article 5, 31 pages.

[49] Weijia Xu, Andrzej Banburski-Fahey, and Nebojsa Jojic. 2023. Reprompting: Automated chain-of-thought prompt inference through gibbs sampling. *arXiv preprint arXiv:2305.09993* (2023).

[50] Weiwei Xu, Kai Gao, Hao He, and Minghui Zhou. 2025. Licoeval: Evaluating LLMs on License Compliance in Code Generation. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 1665–1677. doi:10.1109/ICSE55347.2025.00052

[51] Weiwei Zhang, Shengjian Guo, Hongyu Zhang, Yulei Sui, Yinxing Xue, and Yun Xu. 2023. Challenging Machine Learning-Based Clone Detectors via Semantic-Preserving Code Transformations. *IEEE Transactions on Software Engineering* 49, 5 (2023), 3052–3070. doi:10.1109/TSE.2023.3240118

[52] Zixian Zhang and Takfarinas Saber. 2024. Assessing the Code Clone Detection Capability of Large Language Models. In *2024 4th International Conference on Code Quality (ICCQ)*. 75–83. doi:10.1109/ICCQ60895.2024.10576803

[53] Zixian Zhang and Takfarinas Saber. 2025. Machine Learning Approaches to Code Similarity Measurement: A Systematic Review. *IEEE Access* 13 (2025), 51729–51764. doi:10.1109/ACCESS.2025.3553392

[54] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. 2025. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *International Conference on Learning Representations* 2025 (2025), 66602–66656.

Received 2026-01-29; accepted 2026-06-25