The Journal of Systems and Software 230 (2025) 112548

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Analyzing the dependability of Large Language Models for code clone generation[☆]

Azeeza Eagal[*], Kathryn T. Stolee, John-Paul Ore

Department of Computer Science, North Carolina State University, 890 Oval Drive, Engineering Building II, Raleigh, 27606, NC, USA

ARTICLE INFO

Keywords:
Machine learning for software engineering
Software quality assurance
Software measurement and analytics

ABSTRACT

The ability to generate multiple equivalent versions of the same code segment across different programming languages and within the same language is valuable for code translation, language migration, and code comprehension in education. However, current avenues for generating code clones – through manual creation or specialized software tools – often fail to consistently generate a variety of behaviorally equivalent code clones. Large Language Models (LLMs) offer a promising solution by leveraging their extensive training on diverse codebases to automatically generate code. Unlike traditional methods, LLMs can produce code across a wide variety of programming languages with minimal user effort. Using LLMs for code clone generation could significantly reduce the time and resources needed to create code clones while enhancing their syntactic diversity.

In this quantitative empirical study, we investigate the dependability of LLMs as potential generators of code clones. We gathered equivalent code solutions (i.e., behavioral clones) in C++, Java, and Python from thirty-six programming problems from the well-known technical interview practice platform, LeetCode. We query OpenAI's GPT-3.5, GPT-4, and CodeLlama to generate code clones of the LeetCode solutions. We measure the behavioral equivalence of the LLM-generated clones using a behavioral similarity clustering technique inspired by the code clone detection tool, Simion-based Language Agnostic Code Clones (SLACC). This study reveals that, despite LLMs demonstrating the potential for code generation, their capacity to consistently generate syntactically diverse but behaviorally equivalent code clones is limited. At lower temperature settings, LLMs are more successful in producing behaviorally consistent, syntactically similar code clones within the same language. However, for cross-language cloning tasks and at higher temperature settings and programming difficulties, LLMs introduce greater syntactic diversity and lead to higher rates of compilation and runtime errors, resulting in a decline in behavioral consistency. These findings indicate a need for further quality assurance measures for the use of LLMs for code clone generation. All the data and scripts associated with this paper can be found https://zenodo.org/records/14968618.

1. Introduction

The capability to generate accurate code clones holds significant value in software engineering, impacting several areas including the development of educational material for software engineering students (Patitsas et al., 2013), the creation of robust datasets for semantic code clone detection (Zakeri-Nasrabadi et al., 2023), language migration (Mathew et al., 2020), and software maintenance activities (Aversano et al., 2007; Thummalapenta et al., 2010). Current methods for generating behaviorally equivalent code clones typically involve manual efforts or the use of specialized software tools (Avetisyan et al., 2015; Wei and Li, 2017; Saini et al., 2018). Although manually generating code clones can be beneficial, this approach often demands significant investment in time and resources.

Large Language Models (LLMs) have shown promising code generation capabilities (Xu et al., 2022; Fried et al., 2023; Nijkamp et al., 2023). However, while previous research has predominantly focused on code generation, the area of code clone generation has remained largely unexplored. Unlike code generation, code clone generation requires the input of a prompt that includes an existing code snippet that the LLM is expected to clone, rather than a prompt that is a broad textual description outlining programming tasks. Also, unlike code generation, code clone generation requires that the resulting code segment is behaviorally equivalent to the code segment used in the prompt. This

☆ Editor: Raffaela Mirandola.
* Corresponding author.
E-mail addresses: aeagal@ncsu.edu (A. Eagal), ktstolee@ncsu.edu (K.T. Stolee), jwore@ncsu.edu (J.-P. Ore).

https://doi.org/10.1016/j.jss.2025.112548
Received 20 June 2024; Received in revised form 18 April 2025; Accepted 22 June 2025
Available online 4 July 2025

paper investigates the reliability and effectiveness of LLMs' to produce code clones. If successful, LLMs could amplify current research and software engineering tasks that depends on clone generation. However, if a user asks an LLM for a clone and unexpectedly receives non-clones, this could increase their workload, necessitating additional time for verifying and refining the LLM-generated code. This paper explores the reliability and effectiveness of LLMs in code clone generation through the following two research questions:

RQ1 **Within-language Code Clone Generation:** Given a brief natural language prompt and a code snippet, can a Large Language Model produce consistent code clones of the given code snippet based on runtime behavior *within the same programming language?*

RQ2 **Cross-Language Code Clone Generation:** Given a brief natural language prompt and a code snippet, can a Large Language Model produce consistent code clones of the given code snippet based on runtime behavior *across multiple programming languages?*

One approach to evaluating the code clone generation capabilities of an LLM involves the use of code clone detection tools. While recent advancements have introduced machine-learning techniques for detecting cross-language code clones, they require reliable datasets that are scarce in this field (Zakeri-Nasrabadi et al., 2023). Simion-based Language Agnostic Code Clones (SLACC) is an exception; it does not rely on training datasets to discover code clones. Instead, SLACC employs a dynamic analysis code clone detection framework based on runtime behavior (Mathew et al., 2020). This approach leverages the theory-predicted promises of dynamic program analysis, but inherits some practical limitations caused by the implementation of program analysis tools. SLACC analyzes the functional equivalence of snippets by using similar input and output relationships *"simions"* (Deissenboeck et al., 2012) to identify clones, thereby addressing the challenge of differing underlying representations in various programming languages. It has shown efficacy in detecting statically typed Java code clones with 87.3% precision and dynamically typed Python clones with 94.1% precision (Mathew et al., 2020). The strength of SLACC lies in its transparent process of directly analyzing runtime behavior to detect code clones, but the trade-off is the potential for error inherent in dynamic analysis, compared to the more precise but less versatile static analysis tools. In this study, a SLACC-inspired approach is used to evaluate the code clone generation abilities of LLMs.

Our contributions are as follows:

- **Evaluation of LLMs for Code Clone Generation:** The study provides a comprehensive evaluation of the ability of LLMs, specifically GPT-3.5, GPT-4, and CodeLlama to generate behaviorally equivalent code clones within and across multiple programming languages. This assessment highlights their limitations in consistently producing syntactically diverse yet behaviorally consistent clones.
- **Insights into Temperature Settings and Behavioral Equivalence:** This study explores how different temperature settings affect the diversity and accuracy of generated code clones. It provides empirical evidence that reveals lower temperature settings result in more behaviorally equivalent clones, whereas higher temperatures increase the diversity of generated code clones at the cost of accuracy.

The rest of the paper is organized as follows: Section 2 motivates our study, highlighting the practical importance and challenges in LLM-based code clone generation. Section 3 reviews related work, situating our research within the broader context of code clone detection and LLM capabilities. Section 4 describes our methodology, the specifics of the LLMs used, and our approach to code clone detection with our SLACC-inspired approach. Section 5 details the results of our study, examining the reliability and effectiveness of LLMs to generate code clones within and across different programming languages and temperature settings. Section 6 provides a comprehensive discussion of these results, performing additional analyses to further understand the types of code clones the models generated. Section 7 examines the threats to the validity of our study, both internal and external. Finally, Section 8 outlines future work directions, and the paper concludes with Section 9.

2. Motivating example

Consider Oliver, a computer science professor, who was determined to enhance their introductory programming course. After discovering research studies suggesting that teaching programming through comparing multiple solutions is more effective than the traditional approach of showing one solution at a time (Patitsas et al., 2013; Rittle-Johnson et al., 2020; Margulieux et al., 2021), Oliver decided to use LLMs to help integrate this finding into their course. Oliver hoped that by leveraging LLMs, they could provide students with a diverse range of solutions for their recently completed homework assignment.

However, Oliver faced an unexpected hurdle. Some of the solutions generated by the LLM did not pass the homework test suite. This revealed a critical limitation in the current available LLMs: their inability to consistently produce code clones that are both syntactically diverse and behaviorally consistent. The outcome was that Oliver had to spend additional time and effort tweaking the solutions until they passed the test cases.

Our research aims to overcome Oliver's challenge by first quantifying the impact of the issue – that LLMs fail to produce syntactically diverse and behaviorally consistent clones – with the future goal of enhancing an LLM's capacity to generate accurate and dependable code clones.

This advancement would not only benefit innovative teaching methods like Oliver's but also have broader implications in the field of software engineering. By improving the generation of code clones, LLMs have the potential to create large amounts of reliable code clones, leading to the development of code clone detection datasets that are free from the human biases commonly found in existing datasets (Roy and Cordy, 2018). Furthermore, in a cross-language context, being able to generate code clones in other languages would assist with code translation, language migration, and transpilation (Mayer et al., 2017).

3. Related work

Definition of code clone. A code clone refers to a group of code snippets that "exhibit similarity according to a certain measure of similarity" (Zakeri-Nasrabadi et al., 2023; Baxter et al., 1998). Researchers have proposed two taxonomies to categorize how code clones exhibit similarity: *type similarity* and *threshold similarity* (Baxter et al., 1998).

Type similarity. The type similarity taxonomy identifies four types of code clones. Types I, II, and III usually refer to clones at the method level. Type I code clones are identical except for whitespace and comments. Type II code clones are syntactically equivalent but may differ in trivial components such as identifiers. Type III code clones can differ in control flow statements and may involve added, modified, or removed statements while maintaining functional equivalence. Type IV code clones, unique from the others, involve significantly syntactically dissimilar code pairs with functional equivalence across varying granularity scopes (Roy and Cordy, 2007).

This work focuses on the method level Type IV clones. This is because code clones that are significantly syntactically diverse but behaviorally equivalent are more useful for the targeted domains of code translation and clone generation for education.

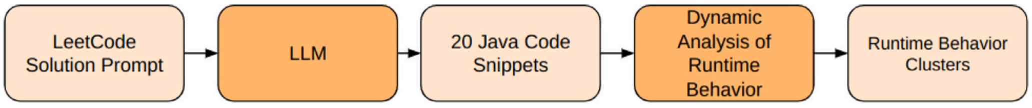

LeetCode Solution Prompt
LLM
20 Java Code Snippets
Dynamic Analysis of Runtime Behavior
Runtime Behavior Clusters

Fig. 1. High-level methodology.

***Threshold similarity*.** similarity describes the extent of similarity between code snippets: a maximum threshold value would indicate an exact match and a minimum threshold value would indicate little to no similarity (Zakeri-Nasrabadi et al., 2023). M. Zakeri-Nasrabadi, et al. identify nine categories of techniques that employ threshold similarity for source code: text, token, tree, graph, metric, test, learning, image, and hybrid techniques. These similarity measures are used in a variety of applications, including plagiarism detection, code smell detection, and program repair. This work uses *test* similarity to identify Type IV code clones (Mathew et al., 2020).

***Code clone detection techniques*.** Text, token, image, and tree code clone detection techniques generally have been shown to detect syntactic code clones (Types I, II, III) – but not semantic code clones (Type IV) – because the techniques exclusively analyze the textual and syntactic features of code snippets (Zakeri-Nasrabadi et al., 2023).

Graph, metric, learning, and hybrid code clone detection techniques have proven more effective at detecting semantic code clones (Type IV) (Zakeri-Nasrabadi et al., 2023). Graph code clone detection techniques commonly utilize program dependency graphs (PDGs) to identify code clones (Zakeri-Nasrabadi et al., 2023). However, determining whether parts of PDGs are equivalent requires finding a subgraph isomorphism, a problem known to be NP-Complete (Ullmann, 1976; Zakeri-Nasrabadi et al., 2023).

Test-based code clone detection techniques analyze the runtime behavior of snippets in order to assess similarity (Zakeri-Nasrabadi et al., 2023). Simion-based Language-Agnostic Code Clone detection technique (SLACC), the technique that inspired the methodology of this study, is a test-based technique. Utilizing the runtime behavior of code snippets, SLACC addresses the challenges associated with identifying code clones across multiple programming languages that lack a common underlying representation. The uniqueness of SLACC lies in its clustering process. SLACC first segments the code repository into smaller executable functions. These functions are then executed using a custom input generator inspired by grey-box testing and multimodal distribution techniques. Once the functions are executed, the resulting input and output relationships are utilized for clustering code snippets (Mathew et al., 2020).

HitoshiIO is another test-based clone detection technique that distinguishes functionally similar source code methods by analyzing their runtime behavior in the Java Virtual Machine (Su et al., 2016a). HitoshiIO uses existing workloads via codebase test cases to identify inputs and their corresponding outputs and measure the functions' input and output similarity (Su et al., 2016a). SLACC employs a similar input and output *"simions"* process to determine code clones (Mathew et al., 2020). In comparison to HitoshiIO, SLACC does not rely upon the existing inputs or workloads, but employs an input generation process to probe the input state space inspired by EQMiner, a separate test-based clone detection technique (Jiang and Su, 2009).

Detecting clones with data-driven, machine learning techniques has been used to detect Type IV code clones (Zakeri-Nasrabadi et al., 2023). However, machine learning approaches are data-hungry and require substantial amounts of reliable training data (Roy and Cordy, 2018). M. Zakeri-Nasrabadi, et al. identified three types of datasets used to train and assess code clone detection techniques: datasets with human oracles, datasets with machine oracles, and hybrid datasets (Zakeri-Nasrabadi et al., 2023). Manually validating these datasets is time-consuming and costly (Roy and Cordy, 2018). Moreover, the reliability of clone datasets is dubious due to the introduction of human bias during either the creation of the clones or validation of the clones within these datasets (Roy and Cordy, 2018). In comparison, SLACC does not require a dataset to train on and still achieves high recall, precision, and F1 scores.

Learning techniques that LLMs have demonstrated high precision and recall in detecting code clones (Zakeri-Nasrabadi et al., 2023). While LLMs might perform well in identifying code clones, their decision-making processes are often opaque, making their results less trustworthy (Zhao et al., 2023). In contrast, test-based techniques offer a clear and transparent methodology that uses runtime behavior of code snippets to assess similarity, providing a level of reliability and transparency not available with LLMs (Mathew et al., 2020).

Hybrid code clone detection techniques are techniques that combine one or more of the previously mentioned techniques to detect code similarity (Zakeri-Nasrabadi et al., 2023). Often, token and text code clone detection techniques are efficient, and graph and tree techniques are effective (Zakeri-Nasrabadi et al., 2023). Combining these techniques intelligently produces robust code clone detection techniques (Zakeri-Nasrabadi et al., 2023). One example of a tool that employs a hybrid code clone detection technique is the tool COSAL (Mathew and Stolee, 2021). COSAL uses SLACC as well as token and tree similarity techniques to perform code search (Mathew and Stolee, 2021). COSAL, given a code query, gathers the top-N search results for each similarity measure — token, tree, and test (Mathew and Stolee, 2021). Then, the results are sorted and ranked by a genetic algorithm NGSA-II (Non-dominated Sorting Genetic Algorithm II) (Mathew and Stolee, 2021; Deb et al., 2002). Researchers found that the non-dominated ranking of these similarity measurements was more effective than a single measurement or other weighted measures at returning relevant snippets (Mathew and Stolee, 2021). COSAL was not used because the overall goal of this paper is to characterize the code clone generation capabilities of the LLM. COSAL did not cluster code clones based on similarity measurements, which is vital to analyzing LLMs code clone generation capabilities.

***LLMs for code generation*.** The potential of machine learning models, initially developed for natural language processing, to address software engineering tasks was recognized in 2016, when researchers identified striking similarities between source code and human language (Hindle et al., 2016). This insight has since catalyzed the development and widespread adoption of LLMs for code generation. Notable advances in this area include the introduction of specialized architectures and enhanced training methodologies, shown by CodeBERT (Feng et al., 2020) and other seminal works (Rozière et al., 2023; Ahmad et al., 2021).

However, the deployment of LLMs in code generation is not without challenges. Key issues remain, including the security of generated code (Yao et al., 2024) and the production of functionally incorrect code snippets (Jesse et al., 2023). These obstacles highlight the need for ongoing research to refine the capabilities and reliability of LLMs in practical software engineering applications (Lo, 2023).

4. Methodology

This work aims to address the research questions in Section 1. Our high-level methodology is shown in Fig. 1. Using a brief natural language task command and LeetCode programming task solution written in Java (*LeetCode Solution Prompt*), we prompted an *LLM* to return semantically equivalent Java code. The same prompt was repeated in the same session to gather twenty snippets in the Java language (*20 Java Code Snippets*). Afterward, a *Dynamic Analysis of Runtime Behavior*

Table 1
Easy LeetCode problem descriptions. The LOC counts refer to the reference examples.

Problem_ID	Problem description	(Input type): Output type	Lines of Code (LOC)		
			C++	Java	Python
adj_inc_subarr	Determine if there exist two adjacent subarrays of length k in an array that are strictly increasing.	(int[], int): boolean	35	47	36
button_longest_push	Return the index of the button with the longest press time.	(int[][]): int	53	31	26
count_subarr_3_cond	Return the number of subarrays of length 3 where the sum of the first and third numbers equals half of the middle number.	(int[]): int	29	26	25
date_to_binary	Convert a given Gregorian calendar date into its binary representation.	(String): String	36	17	18
digitville	Find the two numbers that appear twice in an array of integers from 0 to $n-1$, where all other numbers appear exactly once.	(int[]): int[]	38	31	26
min_ele_after_replacement_w_digit_sum	Replace each number in an array with the sum of its digits and return the minimum value after all replacements.	(int[]): int	32	29	19
min_ops_make_arr_K	Minimize operations to make all elements in an array equal to k by reducing larger values step by step. Return the count or -1 if impossible.	(int[], int): int	28	31	19
min_pos_sum_subarr	Find the minimum sum of a subarray with length between l and r and a sum greater than 0. Return -1 if no such subarray exists.	(int[], int, int): int	39	29	31
original_typed_str_one	Count the possible original strings Alice intended, given that one character may have been repeated at most once.	(String): int	25	21	18
smallest_divisible_digit_prod_one	Find the smallest number ≥ n whose digit product is divisible by t.	(int, int): int	27	33	32
smallest_num_with_all_set_bits	Find the smallest number ≥ n whose binary representation consists only of set bits.	(int): int	14	20	17
stone_removal_game	Alice starts by removing 10 stones, with each turn decreasing by 1. The player who cannot move loses. Determine if Alice wins.	(int): bool	21	23	26

was conducted of the 20 code snippets and the original code snippet to identify *Runtime Behavioral Clusters*. If one behavioral cluster is found among the 20 code snippets, that indicates all the code snippets generated by the LLM are behavioral clones of the reference code. If multiple clusters are found, this suggests that the LLM has produced code from the same prompt that exhibits varying behaviors, signifying that LLM-generated code snippets are not all code clones of the reference code.

To address RQ1, the LeetCode Solution Prompt contained Java code reference examples and asked the LLM to generate Java code snippets. To address RQ2, the *LeetCode Solution Prompt* contained C++ or Python reference examples and asked the LLM to generate Java code snippets.

OpenAI's GPT-3.5 (gpt-3.5-turbo) is studied due to its popularity within both the developer community and the broader public (Yepis, 2023; Milmo and Agenc, 2023). GPT-3.5 is compared against the latest iteration, GPT-4, which demonstrated significantly better performance across several academic benchmarks, including the HumanEval Python coding tasks dataset (OpenAI, 2023). The comparison should provide insight into the reliability LLMs for code clone generation as these models continue to evolve. Additionally, we evaluated CodeLlama (Rozière et al., 2023), focusing on the codeLlama-7b-Instruct-hf variant, an open-source model specifically optimized for tasks involving code generation and understanding. This range of models ensures a comprehensive evaluation of established and emerging approaches.

4.1. Task description

To assess the LLMs' capacity to generate code clones, a list of thirty-six programming problems was compiled from the well-known platform LeetCode (2025). LeetCode offers programming problems intended to aid developers in their preparation for technical interviews. Existing research practices support the utilization of LeetCode programming problems to evaluate the code generation capabilities of an LLM (OpenAI, 2023; Chen et al., 2023; Tian et al., 2023; Döderlein et al., 2023; Nguyen and Nadi, 2022).

We selected LeetCode problems across easy, medium, and hard difficulties to provide a well-rounded evaluation. This inclusion of varying complexities ensures the models are assessed on a diverse set of tasks.

To ensure that the LLMs had not been trained using the information from the LeetCode programming solutions, we collected programming solutions where the first public solution was posted between 09-08-2024 and 01-09-2025. We chose this approach to prevent data contamination, which could inflate the LLMs' performance by letting them generate solutions from seen data instead of truly generalizing (Coignion et al., 2024). From each problem, the first public solution submission in Python, C++, and Java was collected as the *reference code* to use in the prompting phase of the methodology. Each reference code was checked against its associated LeetCode test suite to ensure that the solutions were correct in that all the LeetCode tests passed. Information about each LeetCode problem and their reference codes can be seen in Tables 1, 2, and 3. These table includes a unique problem ID, a brief description of the problem, and the corresponding input and output types. The lines of code (LOC) for each reference implementation in C++, Java, and Python were determined using *CLOC* (Danial, 2024), a command-line tool designed to analyze the size of software projects by counting the lines in source code files. More information about each LeetCode problem can be found in the code artifact associated with this paper: https://zenodo.org/records/14968618.

Table 2
Medium LeetCode problem descriptions. The LOC counts refer to the reference examples.

Problem_ID	Problem description	(Input type): Output type	Lines of Code (LOC)		
			C++	Java	Python
adj_inc_subarr_detect	Find the maximum k for which two adjacent subarrays of length k are strictly increasing.	(int[]): int	46	47	35
beautiful_splits	Count the number of ways to split an array into three contiguous subarrays where the first is a prefix of the second or the second is a prefix of the third.	(int[]): int	51	76	40
grid_sections	Check if two horizontal or vertical cuts can divide an n × n grid into three sections, each containing at least one rectangle without splitting any.	(int, int[][]): bool	73	41	43
max_area_point_constraints	Find the largest axis-aligned rectangle using four points as corners, ensuring no other point lies inside or on the border. Return the area or −1 if none exist.	(int[][]): int	72	60	44
max_coins	Find the maximum coins from k consecutive bags, given non-overlapping segments with fixed coin amounts.	(int[][], int): int	96	70	60
max_num_dist_aft_op	Maximize the number of distinct elements in an array by adding an integer in {−k, k} to each element at most once.	(int[], int): int	39	33	29
max_tar_nodes	For each node in the first tree, find the maximum nodes reachable within k edges when connected to any node in the second tree.	(int[][], int[][], int): int[]	112	85	39
mirror_score	Find the total score by pairing each character in a string with its closest unmarked mirror, adding their index difference to the score.	(String): int	25	32	30
string_shift_dis	Find the minimum total cost to transform string s into t by shifting each character forward or backward in the alphabet with given costs.	(String, String, int[], int[]): int	62	59	51
two_days	Maximize initialCurrency by performing conversions using given exchange rates over two days.	(String, String[][], double[], String[][], double[]): double	92	87	55
XOR_paths	Count paths in a grid, moving right or down, where the XOR of numbers equals k. Return modulo 10^9 +7.	(int[][], int): int	53	48	32
zero_array_transformation	Find the maximum number of queries that can be removed while still converting the array to all zeros using the remaining queries. Return −1 if impossible.	(int[], int[][]): int	64	63	25

4.2. Prompting

The following structured prompt was used to facilitate the generation of code clones:

“Generate a Java semantic code clone of the code below:

{reference code}”

The reference code variable was replaced by various Leetcode programming solution submissions in Java, C++, or Python. When the code was in Java, the task focused within-language cloning for RQ1. For code written in C++, the task involved cross-language code cloning from C++ to Java for RQ2. Similarly, for Python submissions, the task was to clone the code from Python to Java for RQ2.

To further explore RQ1 and assess the model’s ability to create syntactically diverse semantic within-language (Type IV) clones, the following structured prompt was used:

“Generate a syntactically different semantic Java code clone of

the code below: {reference code}”

This task is referred to as “Java to SD Java” throughout the rest of the paper.

OpenAI’s API (OpenAI, 2025) and HuggingFace Inference Endpoint’s API was utilized to query the models. The only parameter that was adjusted throughout the study was the temperature parameter, which controls the randomness of the model’s output. Higher temperature settings result in more varied and unpredictable responses. To explore how this parameter influences code diversity, we tested temperature settings of 0.01, 0.5, and 1. We avoided temperature settings higher than 1 because we observed a noticeable decline in output quality at 1.5 and 2. We adjusted this parameter to understand the impact of temperature adjustments on generating diverse code clones across multiple queries with the same prompt.

4.3. Data

Each of the thirty-six LeetCode problems was promoted using four different prompts (two within-language and two cross-language), across three different temperatures (0.01, 0.5, 1), and repeated 20 times to create 20 code snippets per combination of LeetCode problem, input language (prompt), and temperature. In total, our study analyzes 25,920 LLM-generated code clone candidates — 8640 samples per LeetCode difficulty level, per temperature, and per model, and 6480 samples per task.

After collecting the LLM responses to the prompts, we used regular expressions and the `javalang` Python library (Thunes, 2020) – a tool that provides a lexer and parser for Java source code – to extract the Java code. If a main method was absent, we added one to make the code runnable. Otherwise, no syntactic corrections were applied to the LLM output.

4.4. Metrics

For each combination of LeetCode problem, temperature, and prompt, the 20 code snippets and the reference code were clustered according to their behavior. We measure quality in two ways: the *accuracy* of an individual generated snippet, and the presence *one cluster* when the LLM generates a set of 20 code clones with no false positives.

Accuracy: If a code snippet generated by an LLM is *accurate*, this means it is clustered with the reference code used in the prompt. That

Table 3
Hard LeetCode problem descriptions. The LOC counts refer to the reference examples.

Problem_ID	Problem description	(Input type): Output type	Lines of Code (LOC)		
			C++	Java	Python
count_LCM	Count connected components where nodes connect if LCM ≤ threshold.	(int[], int): int	75	7 5	65
count_num_k_match_adj	Count the number of arrays of size n with values in {1, m} where exactly k adjacent pairs are equal. Return the result modulo 10^9+7.	(int, int, int): int	47	50	36
if_palindrome	For each node in a tree, perform a DFS traversal and check if the resulting string is a palindrome.	(int[], String): boolean[]	94	87	47
max_area_rect_point	Find the largest axis-aligned rectangle with four given points as corners, ensuring no other point is inside or on its border.	(int[], int[]): int	128	94	54
max_freq_ele_performing_ops_two	Maximize the frequency of any element in an array by modifying up to numOperations distinct elements within the range {−k, k}.	(int[], int, int): int	50	40	30
max_fruit	Maximize fruits collected as three children move from different corners to the bottom-right of an n × n grid, emptying rooms they visit.	(int[][]): int	81	57	55
max_score_intervals	Select up to 4 non-overlapping intervals with the maximum total weight. If multiple choices exist, return the lexicographically smallest set of indices.	(int[][]): int[]	93	140	31
max_subarr_rm	Remove all occurrences of one integer from the array at most once and return the maximum possible subarray sum.	(int[]): int	87	87	76
max_sum_weights_after_edge_removal	Remove edges from a tree to ensure no node connects to more than k nodes while maximizing the sum of remaining edge weights.	(int[][], int): int	115	73	50
min_diff_adj_ele_diff	Replace missing values with two chosen integers to minimize the maximum adjacent difference.	(int[]): int	93	72	97
small_id_substring	Flip up to a given number of bits in a binary string to minimize the longest substring of identical characters.	(String, int): int	59	39	51
subsequence_unique_middle_mode	Count subsequences of length 5 with a unique middle mode, returning the result modulo 10^9+7.	(int[]): int	148	225	64

is, for a set of 200 snippets, if 120 are clustered with their specific reference code and 80 are in other clusters, this puts the accuracy at 60%. In a cross-language context, an LLM-generated Java code snippet is a clone of a Python reference code if they are in the same behavioral cluster. That is to say that given the same input to both pieces of code, they produce the same output at least 99% of the time (see Section 4.5.3 for details on the clustering technique).

A special case to consider is the presence of exceptions. During the input generation process, any inputs that cause runtime exceptions in any of the reference code snippets are excluded from the final input corpus. This ensures that all the generated inputs produce valid output for the reference codes to facilitate comparison. Therefore, when an LLM-generated code snippet produces an exception, it indicates a different behavior without having to compare the specifics of the exceptions between the LLM-generated snippet and the reference code.

One Cluster: In an ideal scenario, all 20 code snippets generated per LeetCode problem, model, prompt, temperature, and task would exhibit identical behavior, clustering into one behavioral cluster with the reference code. To determine how frequently the ideal scenario occurs, the success and failures change. If all 20 snippets cluster into one behavior cluster with the reference code, that is a success. Otherwise, if even one LLM-generated snippet is not in the same cluster as the reference code, that is seen as a failure. This tries to illuminate when the LLM is completely dependable for code clone generation.

4.5. Dynamic analysis

Dynamic analysis requires three phases: generating inputs, executing the code on the inputs, and clustering. From there, the accuracy metrics can be computed.

4.5.1. Input generation

Dynamic analysis relies on a set of inputs against which each generated code snippet will be run. To create an input corpus for all code snippets, we used the `Hypothesis` library (MacIver, 2013) — a Python tool designed for generating edge case inputs for unit tests. The method signatures of the Java LeetCode solutions were used by `Hypothesis` to generate the input corpus. Initially, between 512 and 3000 inputs were produced for each LeetCode problem, covering Java, Python, and C++ solutions. The same input corpus was then applied across all solutions.

Next, each original LeetCode solution was executed against its corresponding input corpus, and any inputs that triggered errors or exceptions were excluded. This step was crucial to avoid comparing error messages between the original and generated snippets.

The refined input corpus included 256 inputs per LeetCode problem, ensuring that each input could generate valid outputs across all the Java, Python, and C++ reference codes.

4.5.2. Behavioral data collection

Following the creation of the input corpus for each generated LLM snippet, the Python `subprocess` module (Python, 2025) was utilized to execute the snippets using the input corpus to gather the input/output relationships of each snippet. The Java snippets were executed using OpenJDK version 17.0.13, Python snippets with Python version 3.10.12, and C++ snippets were compiled and run using C++20.

This study finds behavioral clusters for each set of 20 functions generated from the same prompt, model, and task, under the same temperature setting for each LeetCode problem. The clustering is based on the behavioral similarity of the input corpus that excluded inputs causing errors and exceptions. The reference code's behavior was also

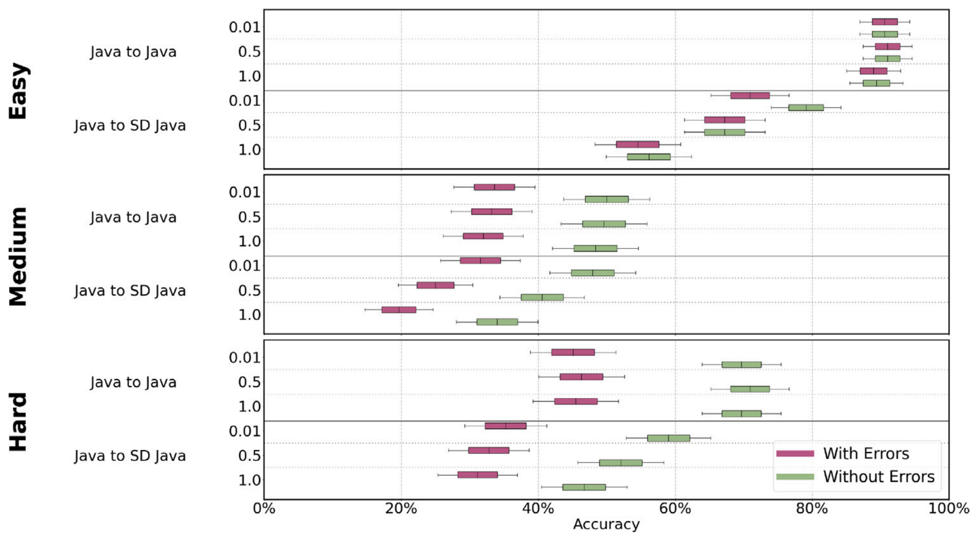


Fig. 2. Within-language semantic correctness across levels, temperatures, and tasks for GPT-3.5.

included to serve as a benchmark for correctness, with the premise that the cluster containing the original solution represents the correct clustering.

4.5.3. Clustering

Consistent with the methodology of SLACC, the similarity between two functions was determined based on the proportion of identical input/output pairs, akin to the Jaccard Index (Mathew et al., 2020).

We defined a similarity threshold of 99% behavioral similarity across 256 inputs to signify behavioral equivalence. Previous work has shown that false positives plateau after 64 inputs (Mathew et al., 2020) and support the use of 256 inputs for cross-language code clone detection. While this stringent criteria drastically reduces the risk of incorrectly classifying behaviorally different functions as identical, we avoided setting the threshold at 100% because our 256 inputs were edge case inputs. Previous work has demonstrated that functions producing the same output for trivial input cases often differ significantly for more complex inputs, underscoring the necessity of a high but not absolute threshold (Deissenboeck et al., 2012).

A representative-based partitioning strategy, as described in prior research (Roy et al., 2009; Su et al., 2016b; Mathew et al., 2020), was used for clustering. The process began with the initialization of an empty Union-Find data structure, with each function initially acting as its own representative. Functions then were compared pairwise; if the behavioral similarity between two functions exceeds a predefined similarity threshold, they are unified under the same representative. Functions that did not meet this threshold remained as independent representatives.

As the process concluded, clusters were established based on their root representatives. To ensure the robustness of the clustering, every function within a cluster was cross-validated against every other function in the same cluster to confirm that all met the similarity threshold.

5. Results

5.1. RQ1: Within-language clone generation

Figs. 2, 3, and 4 present Agresti-Coull confidence interval results of clustering the clone candidates with a 99% runtime behavior similarity threshold for each model. A snippet is deemed semantically correct if it clusters with the reference code; otherwise, it is considered a failure.

The results indicate that all models performed best when generating Java to Java clones of Easy reference codes at lower temperatures. At temperatures of 0.01 and 0.5, each model achieved a median expected success rate of approximately 90%. However, while the GPT models maintained a high success rate even at the highest temperature, CodeLlama's expected success rate dropped by 17% at temperature 1. Notably, these trends remained consistent even when excluding compilation and runtime errors, suggesting that errors were not the primary factor influencing semantic dissimilarity.

A significant decline in semantic correctness is observed for Medium Java to Java clone candidates across all models compared to their Easy counterparts. GPT-3.5's median expected success rate plummeted from 90% to 34% at temperature 0, 33% at 0.5, and 32% at 1. GPT-4 followed a similar trend, maintaining 34% at both 0 and 0.5 before slightly decreasing to 31% at 1. CodeLlama exhibited a more pronounced drop, starting at 34% at temperature 0, decreasing to 32% at 0.5, and falling sharply to 24% at 1.

When exceptions and errors were excluded, the GPT models demonstrated notable improvements in semantic correctness. For Java to Java cloning of Medium reference codes, GPT-3.5 and GPT-4 both improved to a median expected success rate of approximately 50% across all temperatures. In contrast, as the temperature increased CodeLlama showed less variation in performance. Overall, these findings suggest that a non-trivial portion of the semantic dissimilarity between Medium reference codes and their clone candidates stem from the GPT models producing buggy code.

The models exhibited higher semantic correctness when generating clones of Hard reference codes compared to Medium ones —— a anomalous result, particularly when compared to prior studies leveraging LeetCode as a benchmark, which consistently report greater model success on Medium-level tasks relative to Hard-level tasks (OpenAI, 2023; Yeo et al., 2024; Coignion et al., 2024). GPT-3.5's expected semantic correctness was higher by approximately 12% for Hard reference codes, while GPT-4 and CodeLlama showed a similar improvement of around 8%. As with the Medium clone candidates, excluding exceptions and errors led to a notable performance boost. GPT-3.5's expected success rate increased by approximately 25% across all temperatures, with GPT-4 and CodeLlama demonstrating similar gains. This further suggests that a substantial portion of semantic inaccuracies in Hard clone candidates can be attributed to errors in the generated code.

Part of this study was to investigate the ability of LLMs to generate syntactically diverse (Type IV) code clones. When prompted to produce these types of clones, both GPT-4 and CodeLlama were effective in producing at least semantically correct clones. GPT-4's expected success rate is approximately 91% across all temperatures. CodeLlama performed similarly, reaching 91% at temperature 0, 88% at 0.5, but

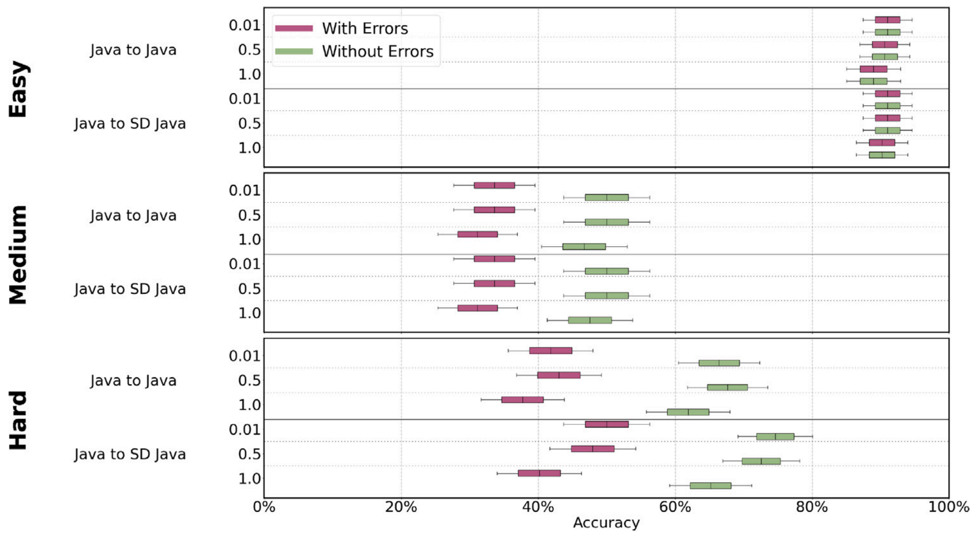


Fig. 3. Within-language semantic correctness across levels, temperatures, and tasks for GPT-4.

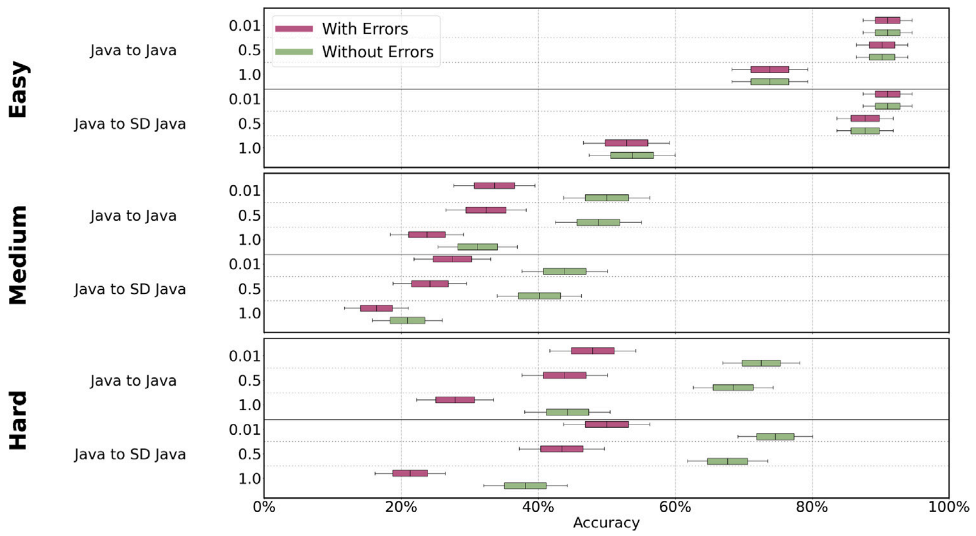


Fig. 4. Within-language semantic correctness across levels, temperatures, and tasks for CodeLlama.

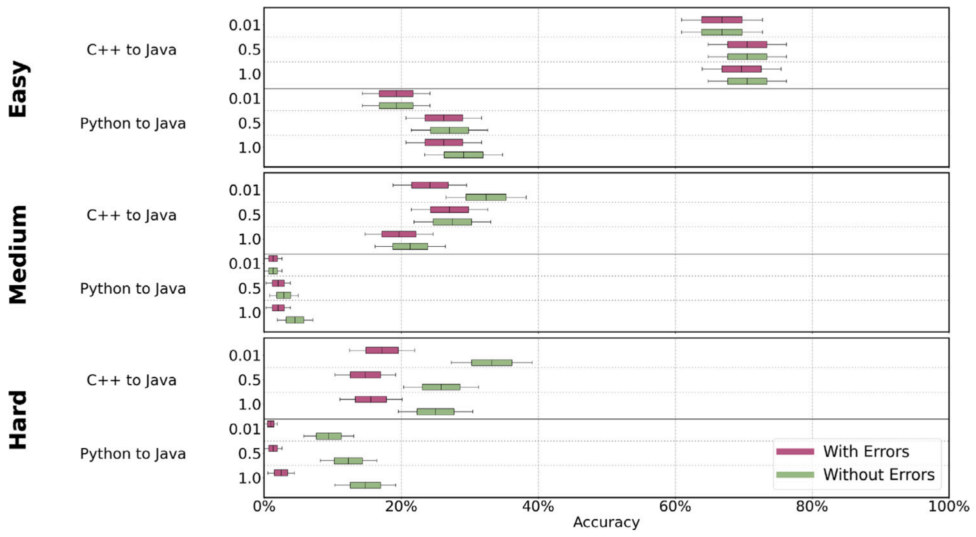


Fig. 5. Cross-language semantic correctness across levels, temperatures, and tasks for GPT-3.5.

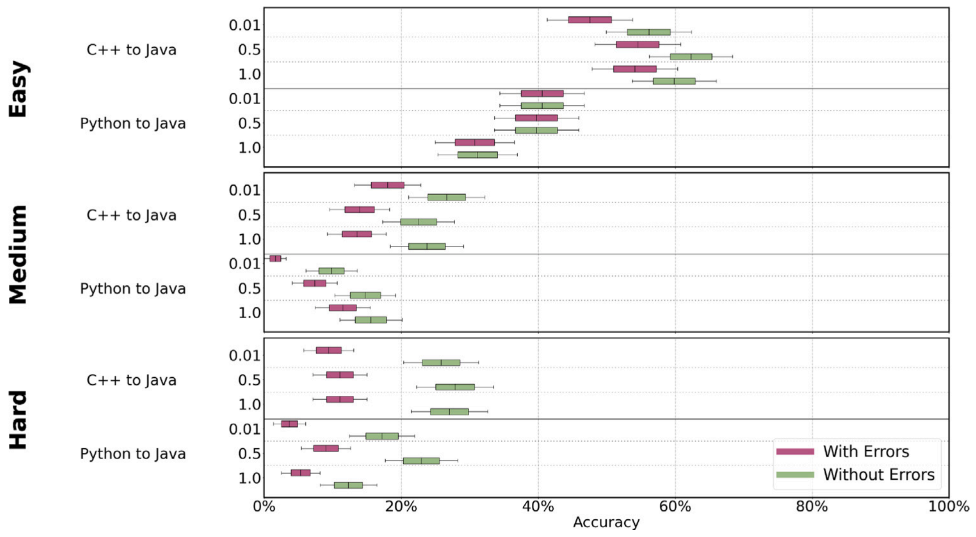


Fig. 6. Cross-language semantic correctness across levels, temperatures, and tasks for GPT-4.

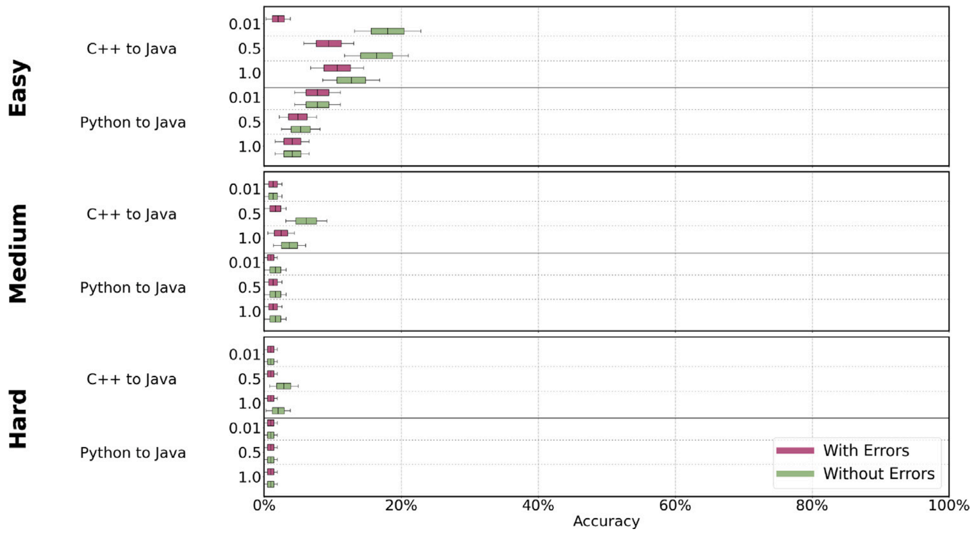


Fig. 7. Within-language semantic correctness across levels, temperatures, and tasks for CodeLlama.

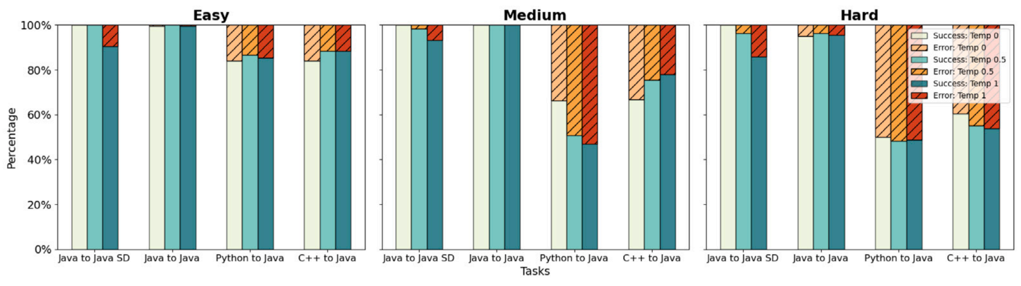
declining significantly to 53% at temperature 1. GPT-3.5 performed the worst with a decreasing success rate as temperature increased, starting at 71% at temperature 0, decreasing to 67% at 0.5, and further dropping to 55% at temperature 1. Just as before, exceptions were not a large cause of semantic dissimilarity for cloning Easy LeetCode reference codes — except for GPT-3.5 at the lowest temperature which saw a 8% improvement in correctness when exceptions were filtered out.

Similar to within-language cloning for unspecified types, the expected success rate for semantic correctness dropped significantly when generating Type IV clones for Medium and Hard reference codes. The trend of the LLMs performing better at cloning Hard reference codes than Medium reference codes also persisted. Similarly, the substantial improvement in semantic correctness at higher temperatures, when excluding compilation and runtime errors, also continued when cloning Hard reference codes.

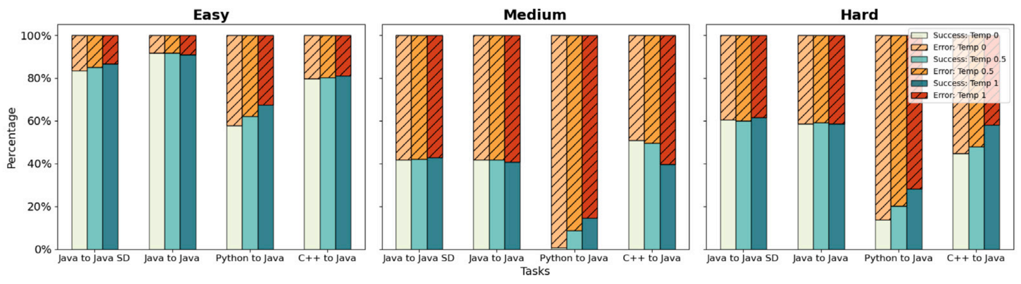
To get a fuller understanding of the dependability of LLMs for code clone generation, we need to assess how frequently the ideal scenario occurs. Specifically, we need to determine how often the LLMs produce behaviorally equivalent code clones for **all** 20 code snippets generated using the same reference code, prompt, and temperature. Under this scenario, significant variations in performance across different difficulty levels and temperature settings.

In general, the ideal scenario typically occurs when the models generate Java to Java clones of Easy reference codes. At lower temperature settings, particularly at 0.01, the GPT models are able to produce single behavioral clusters, but this occurrence decreases as the temperature increases. This suggests that at low temperatures, LLMs exhibit strong determinism, producing nearly identical outputs across multiple generations. However, as the temperature increases to 0.5 and 1.0, the code snippet at least has some changes increases and the reliability of the generated clones decreases. This can especially be seen in CodeLlama, where it never produces a perfect cluster for within-language clones at temperature 1.

For Type IV clones, the ability to balance syntactic diversity with behavioral consistency varies across models. GPT-4 maintains a higher accuracy across temperature settings, though behavioral inconsistencies emerge at higher programming difficulties. In contrast, CodeLlama exhibits substantial degradation at higher temperatures, exclusively generating outputs that do not all cluster with the reference code. This decline in semantic correctness suggests that while increased temperature encourages more diverse types of clones, it also introduces

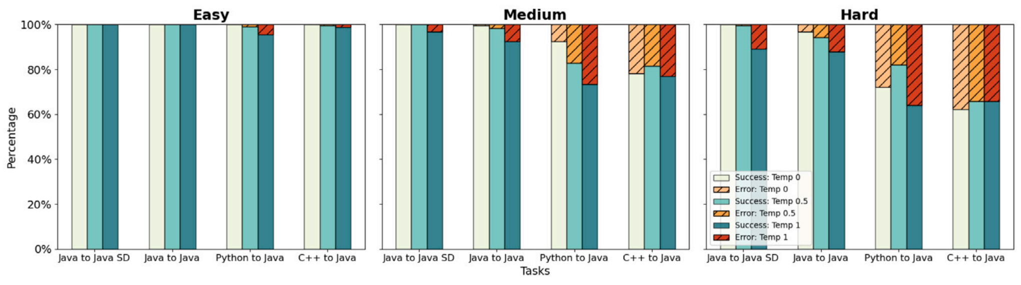


(a) Compilation Errors

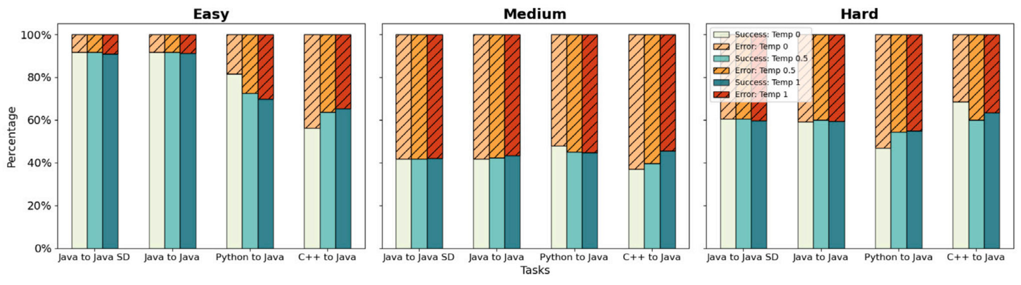


(b) Runtime Errors

Fig. 8. Compilation and runtime errors of GPT-3.5.



(a) Compilation Errors



(b) Runtime Errors

Fig. 9. Compilation and runtime errors of GPT-4.

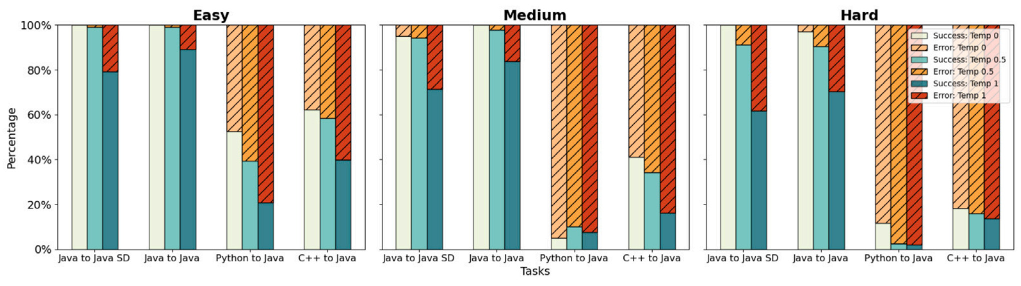
modifications that compromise functional correctness. The challenge becomes more pronounced with Medium and Hard clone candidates, where the likelihood of all 20 generated clone candidates exhibiting identical behavior diminishes significantly.

These findings reveal a fundamental trade-off in LLM-based clone generation: lower temperatures promote deterministic, behaviorally consistent outputs but may limit syntactic diversity, whereas higher temperatures introduce variation at the cost of reliability. Among the evaluated models, GPT-4 demonstrates the highest stability for within-language cloning of unspecified types, while CodeLlama exhibits the most variability.

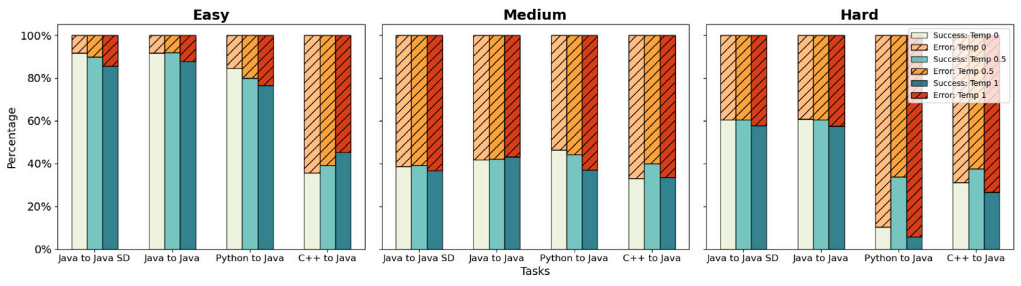
Additionally, when investigating the dependability of LLMs for code clone generation, we must consider the types of errors that occur during the process — both compilation and runtime errors. This information for each model can be seen in Figs. 8, 9, and 10. Interestingly, the models tend to have a low amount of compilation errors when producing within-language clones, meaning that the code outputted is well-formed and syntactically correct, but produces a significant amount of runtime errors when cloning Medium and Hard reference codes.

5.2. RQ2: Cross-language clone generation

Figs. 5, 6, and 7 show the Agresti-Coull confidence interval results for clustering the cross-language code clone candidates based on runtime behavior. Across all models, performance in cross-language cloning is consistently lower than within-language cloning, with a



(a) Compilation Errors



(b) Runtime Errors

Fig. 10. Compilation and runtime errors of CodeLlama.

steep decline in expected correctness at higher problem difficulties and higher temperatures.

For C++ to Java cloning, GPT-3.5 outperforms GPT-4, particularly when cloning Easy reference codes. With these reference codes, GPT-3.5's median expected success rate hovers around 70% across temperatures. In contrast, GPT-4 struggles with median expected success rates below 55% on similar tasks. However, for both models, performance declines sharply at higher programming difficulty levels. Among the models tested, CodeLlama performs the worst, consistently showing low success rates across all difficulty levels and temperature settings.

The models performed the worst when generating clones from Python to Java. GPT-4 achieves the highest expected success rate when cloning Easy reference codes at lower temperatures. However, this rate declines sharply for Medium and Hard reference codes. GPT-3.5 follows a similar trend, with a steep performance drop at higher temperatures. CodeLlama performs the worst among the models; it never achieves an expected semantic correctness rate above 8% across all temperatures and difficulty levels. We hypothesize that the performance gap is due to fundamental syntactic differences between Python and Java. Unlike C++ and Java, which are both statically typed and share a C-style syntax, Python is dynamically typed and relies on indentation rather than brackets to define code blocks. This contrast in typing and block structure likely contributes to the decreased accuracy of LLMs when translating between Python and Java.

The ideal scenario – where all LLM-generated clone candidates gathered into the same behavior cluster as the reference code – was rare in cross-language clone generation. As shown in Figs. 8, 9, and 10, medium and high reference code difficulty led to more compilation errors, a trend also observed in within-language cloning. Even when clones were compiled successfully, runtime errors remained prevalent, particularly for Medium reference codes.

6. Discussion

6.1. Levenshtein edit distance & accuracy

In the previous section, it was established that GPT-3.5, GPT-4, and CodeLlama frequently generated within-language code clones with high behavioral similarity. Although the primary research questions of this study have been addressed, further exploration is needed to characterize the specific **types** of within-language LLM-generated code clones. We employed the Levenshtein Edit Distance algorithm via the Python library `Levenshtein` (Bachl, 2024) to discern the amount of Type I within-language clones. Given adjustments in the preprocessing phase of our study, we defined Type I code clones as those having a Levenshtein distance of 10 or less relative to the reference code.

Table 4 shows the percent of Type I clones across within-language tasks, temperatures, and models. This table shows that GPT-4 produced no Type I clones, and that GPT-3.5 produced a limited number, mostly at lower temperatures. Codellama produced the highest number of Type I clones, again at lower temperatures. While this indicates strong replication capability, it is not ideal for within-language cloning where Type IV clones – structurally diverse but functionally equivalent clones – are more valuable. The prevalence of Type I suggests CodeLlama struggles with either Type IV cloning or prompt guidance. Section 5.1 showed that all the models struggled with within-language cloning of higher difficulty reference codes. Given this and the varying rates of Type I clone generation between the models, examining the correlation between a clone's Levenshtein Edit Distance and its semantic correctness could reveal how well LLMs produce **accurate** Type IV clones.

To examine the correlation between Levenshtein Edit Distance and the semantic accuracy of LLM-generated clones, we conducted a logistic regression analysis. The analysis revealed a statistically significant negative correlation between the Levenshtein Edit Distance and the accuracy of the LLM-generated solutions (p-value < 0.0001). Specifically, the regression coefficient for Levenshtein Edit Distance was -0.0009, suggesting that an increase in edit distance slightly decreases the likelihood of generating a correct solution.

6.2. AST edit distance & accuracy

While Levenshtein Edit Distance is used to identify Type I code clones by measuring textual similarity between code snippets and their reference code, Abstract Syntax Tree (AST) Edit Distance identifies Type II clones by assessing structural similarity in their underlying syntax trees. We used the Python library `javalang` (Thunes, 2020) and `zss` (Henderson, 2021) to calculate the AST Edit Distances between the LLM-generated code snippets and their reference code.

Table 4
Percent of Type I clones across within-language tasks, temperatures, and models.

Model	Task	Temperature	Easy (%)	Medium (%)	Hard (%)
GPT-3.5	Java to Java	0.01	13	18	30
		0.5	15	13	17
		1	9	4	7
	Java to SD Java	0.01	0	0	17
		0.5	0	1	2
		1	0	0	0
GPT-4	Java to Java	0.01	0	0	0
		0.5	0	0	0
		1	0	0	0
	Java to SD Java	0.01	0	0	0
		0.5	0	0	0
		1	0	0	0
CodeLlama	Java to Java	0.01	58	80	65
		0.5	57	62	44
		1	32	27	20
	Java to SD Java	0.01	70	82	66
		0.5	59	67	48
		1	18	23	15

Table 5
Percent of Type II clones across within-language tasks and temperatures.

Model	Task	Temperature	Easy (%)	Medium (%)	Hard (%)
GPT-3.5	Java to Java	0.01	100	96	98
		0.5	98	91	96
		1	84	84	90
	Java to SD Java	0.01	90	100	99
		0.5	86	96	88
		1	83	81	77
GPT-4	Java to Java	0.01	92	100	85
		0.5	95	99	87
		1	92	79	79
	Java to SD Java	0.01	98	99	88
		0.5	93	96	88
		1	81	78	74
CodeLlama	Java to Java	0.01	91	97	97
		0.5	95	87	85
		1	82	68	59
	Java to SD Java	0.01	100	100	100
		0.5	98	96	89
		1	68	62	59

Table 5 shows the percent of Type I clones across within-language tasks, temperatures, and models. This table shows that most clone candidates were Type II clones at lower temperatures, with the lowest percentages consistently at Temperature 1. GPT-3.5 and GPT-4 maintained at least 90% Type II clones at temperatures 0.01 and 0.5 but saw a decline at Temperature 1. CodeLlama followed a similar trend, with a significant drop at Temperature 1, particularly for Java to SD Java. The decline from 0.5 to 1 across all models highlights the impact of temperature on syntactically diverse code clone generation.

As with the Levenshtein distance, it is important to investigate the relationship between AST distance and accuracy. Our logistic regression analysis reveals a significant negative correlation between AST distance and accuracy ($\beta = -0.6609$, $p < 0.001$), indicating that a greater AST distance reduces accuracy. Despite a low pseudo R^2 (0.01938), the model significantly improves over the null ($p = 3.044 \times 10^{-74}$), confirming AST distance as a key factor. This aligns with the observed accuracy drop at higher temperatures, reinforcing that increased syntactic variation challenges LLMs. Future work should explore ways to balance diversity and accuracy in code clone generation (see Table 5).

6.3. Alternatives to LLMs for code clone generation

For within-language code clone generation, a direct copy-and-paste approach is the most accessible method for producing Type I clones. However, Type II and III clones require specialized tools. For example, Recaf (GitHub, 2025a) and Java Deobfuscator (GitHub, 2023) can modify Java code, but both have limitations: Recaf (GitHub, 2025a) requires manual intervention, making it impractical for large-scale automation, while Java Deobfuscator (GitHub, 2023) is limited to reversing obfuscated code and lacks a general-purpose refactoring approach.

For Java Type IV clones, ReFaster (Wasserman, 2013) enables structured refactoring through predefined transformation rules while preserving functional equivalence. While effective for targeted modifications, its major drawback is the need for explicit rule definitions, limiting its adaptability compared to LLMs, which can generate diverse syntactic variations dynamically.

Generating functionally equivalent clones between Python and Java or C++ and Java requires tools that can either transpile, translate, or infer equivalences between the two languages. Several transpilers have been developed to create these clones. P2J (Chris Humphreys, 2013) translates Python to Java but lacks support for advanced constructs like lambdas, metaclasses, and dynamic typing. Jython (GitHub, 2024) allows Python to run on the JVM but does not generate independent Java source code and requires manual updates when languages change. For C++ to Java cloning, Tangible Software's C++ to Java Converter (Inc., 2025) preserves object-oriented structure but struggles with STL templates and pointer manipulation.

While these transpilers provide rule-based mappings between languages, their effectiveness is limited. Unlike LLMs, which leverage probabilistic reasoning to generate functionally equivalent clones, transpilers require explicit rules and lack adaptability to novel code structures. Additionally, transpilers typically produce deterministic output, whereas LLMs can generate multiple structurally diverse implementations of functionally equivalent code. However, as shown in Section 5.2, LLMs seem to be prone to errors and inconsistencies for cross-language clone generation, necessitating post-processing and verification to ensure correctness.

6.4. Practical implications

The practical implications of this study include software engineering tasks that make use of code clones, revealing both the potential and challenges of using LLMs to create code clones.

LLMs offer the ability to create multiple versions of a code segment, providing students with diverse solutions to the same problem. For instance, educators in software engineering could make use of LLMs to create comparative comprehension educational materials, enhancing student learning through the showcasing of an array of solutions to one problem (Patitsas et al., 2013). Our study reveals that using LLMs at lower temperatures can result in the generation of more precise within-language code clones. However, this is associated with a decrease in syntactic diversity, an aspect highly desirable in both educational settings and the formation of robust code clone detection datasets (Zakeri-Nasrabadi et al., 2023). In contrast, LLM-generated clones at higher temperatures show increased syntactic diversity but are considerably more error-prone. The findings suggest a potential solution: the application of program repair techniques on LLM-generated clones could be used to reduce the errors while preserving their syntactic diversity. This type of solution along with others could make LLMs useful for wide range of software engineering tasks that involve code clone generation.

However, while our study primarily focuses on the behavioral similarity of code clones generated by LLMs, there are practical challenges that come from using LLMs for code clone generation. The automated generation of code clones raises concerns about the propagation of security vulnerabilities. If an LLM generates code clones with subtle security flaws, developers may inadvertently introduce vulnerabilities into their codebases. In fact, a study from 2021 found that when GitHub Copilot (GitHub, 2025) was used to infill code for 1869 programs in 89 scenarios, 40% of the code chunks generated were vulnerable (Pearce et al., 2021). Later in 2023, the same research institution conducted a study comparing code from 58 students – some with LLM access and some without – who were tasked with implementing 12 functions for basic operations on a linked list representing a ‘shopping list’ in C (Sandoval et al., 2023). Surprisingly, the study found that the group using GitHub Copilot was 6%–10% more productive than the control group, with no more than a 10% increase in bug rates when compared to the control group. While these findings suggest that security concerns with LLM assistants may be less severe than initially thought, the authors emphasized the need for larger sample sizes and more diverse user groups. In 2024, an in-lab study assigned 30 experienced software developers to three groups: one using a poisoned code completion tool, another using a poisoned code generation tool, and a control group with no tool. Participants completed three programming tasks followed by an exit interview. The study found that developers using code generation tools, similar to those examined in this study, were more likely to introduce insecure code compared to the other groups (Oh et al., 2024). These findings highlight the need for further research into LLM-generated insecure clones and the complexities associated with this form of code generation.

Furthermore, ethical considerations arise when LLMs are used for automated code clone generation. In the context of computer science education, a student could use an LLM to transform a prior assignment solution into a Type IV clone. Because these clones differ syntactically but remain functionally equivalent, traditional plagiarism detectors may fail to flag them. This raises academic integrity concerns, as students could exploit LLMs to evade detection. While there is prior work on detecting ChatGPT-generated code submissions in CS1 courses (Hoq et al., 2024), this specific form of plagiarism remains unexplored.

Overall, LLM-facilitated code clone generation has practical applications in areas such as language migration (Mathew et al., 2020) and software maintenance tasks (Aversano et al., 2007; Thummalapenta et al., 2010), where cross-language code generation may be particularly beneficial. However, our study highlights significant challenges, including inconsistencies in behavioral similarity and high rates of runtime and compilation errors in LLM-generated clones, signaling the need for solutions. Additionally, LLM-generated code clones raise security and ethical concerns, such as the potential propagation of vulnerabilities and risks of misuse, including plagiarism. Addressing these issues is essential to ensuring the reliability, security, and ethical use of LLM-generated code clones in software development.

7. Threats to validity

7.1. Threats to internal validity

The study has several threats to its internal validity. First, the study may be influenced by selection bias due to the specific choice of LeetCode problems, code clone generation tasks, and languages used. Different LeetCode problems, programming languages, and code clone generation tasks might lead to variations in the results. We chose to include the full range of LeetCode difficulty levels, our code cloning tasks, and our programming languages due to their broad representation of distinct programming paradigms and widespread use among developers. While these choices introduce some limitations, our technical processes could be adapted to study additional datasets, code cloning tasks, and programming languages in future work to further generalize our findings. Second, the preprocessing portion of the methodology implemented in the study to gather the input/output relationships efficiently, such as removing superfluous English explanations from the LLMs’ output and creating Java classes from the LLMs’ output could also introduce biases affecting the experiment’s internal validity. Third, across the LLMs at varying degrees of temperature, code clone generation tasks, and Leetcode problem, a fixed sample size of twenty was used. While this was deemed practical for preliminary analysis, an LLM can potentially generate a broader array of code snippets, which might not be accounted for in this limited sample size. Lastly, this study employed minimal prompt engineering. This is due to two reasons: the study aimed to probe the baseline code clone generation abilities of the models and to minimize the effects that could arise from using different wording with prompts across models to generate the code clone data. In addition, the prompts for generating semantically equivalent but syntactically diverse within-language clones were minimally modified from those used in other code cloning tasks to evaluate the model’s ability to generate Type-IV clones. Employing this consistent minimal prompting approach across all models may underestimate their full potential; however, it provides a foundational analysis of their code clone generation capabilities.

7.2. Threats to external validity

The external validity of this study is subject to several threats that could limit the generalizability of the findings. First, the controlled experiments with LeetCode solutions in this study might not reflect the diverse and complex scenarios encountered in the broad spectrum of real-world software projects. We used LeetCode problems to study the code clone generation abilities of LLMs because these types of problems have served as a benchmark for evaluating the code generation previously (OpenAI, 2023; Bubeck et al., 2023; Huang et al., 2024), they have well-defined correctness criteria, and they generally reflect algorithmic and data structure concepts encountered in software development. We have structured our technical processes such that this experiment could extend to other datasets in future work. Second, the behaviors observed in this study might not scale to larger or more complex software projects. While we did not include such examples in this study. Our technical process with modification could be extended for these types of datasets. Third, the study focuses on the GPT-3.5, GPT-4, and CodeLlama models. Hence, the observed limitations and inconsistencies in code clone generation may not be representative of other LLMs or future iterations of these models.

8. Future work

Overall, this study suggests several directions for future work, some of which are motivated by the limitations and validity threats identified in the evaluation. A particularly promising direction is the evaluation of code clone generation using SLACC (or similar tools) across a broader range of LLMs, with an emphasis on recently-developed models that exhibit strong code generation capabilities. This includes specialized models such as StarCoder (Li et al., 2023), GraphCodeBERT (Guo et al., 2021), and Qwen2-Coder-Instruct (Yang et al., 2024), as well as general-purpose models like the Claude 3 family (Anthropic, 2025), which has demonstrated superior performance to GPT-3.5 and GPT-4 on code generation benchmarks such as MBPP and HumanEval (Anthropic, 2025). Evaluating these models may offer a more comprehensive understanding of the capabilities and limitations of LLMs, while also offering additional context for the code clone generation results presented in this paper.

Additionally, as mentioned in Section 7, our technical process can be adapted to study LLMs' ability to perform code clone generation of code clone snippets for larger, more complex software projects, other languages, and other datasets. Our results also indicate the LLMs particularly struggle with cross-language tasks. Future work could explore prompt engineering techniques and fine-tuning on curated datasets to improve performance.

Lastly, in light of the issues identified with the LLMs investigated in this study, it may be worth implementing the potential solutions to improve the code clone generation abilities of large language models such as incorporating runtime behavior and syntactic error detection tools into their training and validation processes.

9. Conclusion

This study examined the dependability of Large Language Models in generating behaviorally equivalent code clones within and across programming languages. Specifically, we assessed the clone-generation capabilities of GPT-3.5, GPT-4, and CodeLlama across different temperature settings by testing them on a diverse set of LeetCode problems. Our findings indicate that while LLMs can successfully generate within-language clones at lower temperatures, their reliability diminishes at higher temperatures and programming difficulties.

Furthermore, cross-language clone generation presents a greater challenge, with significantly higher rates of compilation and runtime errors, particularly when translating from Python to Java. Even when successful, the generated clones exhibit lower semantic correctness compared to their within-language counterparts. Our results suggest that while LLMs can produce syntactically diverse clones, they struggle with preserving behavioral consistency across languages.

These results highlight both the promise and limitations of LLMs for automated code clone generation. While LLMs can facilitate rapid generation of functionally equivalent code snippets, their inconsistency in maintaining correctness – especially for complex and cross-language tasks – suggests that they are not yet a fully reliable solution for automated clone generation without human oversight or supplementary verification techniques. Future work should explore methods for improving clone accuracy, such as integrating program repair tools, refining prompt engineering techniques, or leveraging fine-tuned models specifically trained for clone generation tasks. Future work should also consider studying a broader range of LLMs, especially newer and specialized LLMs with strong code generation capabilities, to further generalize our findings.

CRediT authorship contribution statement

Azeeza Eagal: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Kathryn T. Stolee:** Writing – review & editing, Writing – original draft, Supervision, Methodology, Funding acquisition. **John-Paul Ore:** Writing – review & editing, Writing – original draft, Supervision, Investigation, Funding acquisition, Conceptualization.

Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this work the author(s) used OpenAI's ChatGPT to improve the readability and language of the paper. After using tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the publication.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Azeeza Eagal reports financial support was provided by National Science Foundation. John-Paul Ore reports financial support was provided by National Science Foundation. Kathryn Stolee reports financial support was provided by National Science Foundation. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research was supported in part by National Science Foundation SHF #2006947. Generative AI tools significantly contributed to both the software development process and the revision of this paper. Specifically, we used GitHub Copilot for some aspects of this paper's software development component. During the preparation of this work the author(s) used OpenAI's ChatGPT to improve the readability and language of the paper. After using tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the publication. We acknowledge these tools for their role in facilitating our research and thank the respective teams and organizations for their development and maintenance.

Data availability

I have shared my data and code on Zenodo.

References

Ahmad, W.U., Chakraborty, S., Ray, B., Chang, K., 2021. Unified pre-training for program understanding and generation. CoRR https://arxiv.org/abs/2103.06333.

Anthropic, 2025. The claude 3 model family: Opus, sonnet, haiku. https://assets.anthropic.com/m/61e7d27f8c8f5919/original/Claude-3-Model-Card.pdf. (Accessed 31 March 2025).

Aversano, L., Cerulo, L., Di Penta, M., 2007. How clones are maintained: An empirical study. In: 11th European Conference on Software Maintenance and Reengineering. CSMR'07, pp. 81–90. http://dx.doi.org/10.1109/CSMR.2007.26.

Avetisyan, A., Kurmangaleev, S., Sargsyan, S., Arutunian, M., Belevantsev, A., 2015. LLVM-based code clone detection framework. In: 2015 Computer Science and Information Technologies. CSIT, pp. 100–104. http://dx.doi.org/10.1109/CSITechnol.2015.7358259.

Bachl, M., 2024. Levenshtein: A python c extension module for fast computation of Levenshtein distance and string similarity. URL https://pypi.org/project/python-Levenshtein/, Version 0.25.1.

Baxter, I., Yahin, A., Moura, L., Sant'Anna, M., Bier, L., 1998. Clone detection using abstract syntax trees. In: Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272). pp. 368–377. http://dx.doi.org/10.1109/ICSM.1998.738528.

Bubeck, S., Chandrasekaran, V., Eldan, R., Gehrke, J., Horvitz, E., Kamar, E., Lee, P., Lee, Y.T., Li, Y., Lundberg, S., Nori, H., Palangi, H., Ribeiro, M.T., Zhang, Y., 2023. Sparks of artificial general intelligence: Early experiments with GPT-4. arXiv:2303.12712. URL https://arxiv.org/abs/2303.12712.

Chen, L., Zaharia, M., Zou, J., 2023. How is ChatGPT's behavior changing over time?. arXiv:2307.09009.

Chris Humphreys, J.W., 2013. GitHub - chrishumphreys/p2j: Python to java translator — github.com. https://github.com/chrishumphreys/p2j. (Accessed 27 February 2025).

Coignion, T., Quinton, C., Rouvoy, R., 2024. A performance study of LLM-generated code on leetcode. In: Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering. In: EASE 2024, ACM, pp. 79–89. http://dx.doi.org/10.1145/3661167.3661221.

Danial, A., 2024. Cloc: 2.00. http://dx.doi.org/10.5281/zenodo.5760077.

Deb, K., Pratap, A., Agarwal, S., Meyarivan, T., 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comput. 6 (2), 182–197. http://dx.doi.org/10.1109/4235.996017.

Deissenboeck, F., Heinemann, L., Hummel, B., Wagner, S., 2012. Challenges of the dynamic detection of functionally similar code fragments. In: Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering. CSMR '12, IEEE Computer Society, USA, pp. 299–308. http://dx.doi.org/10.1109/CSMR.2012.38.

Döderlein, J.-B., Acher, M., Khelladi, D.E., Combemale, B., 2023. Piloting copilot and codex: Hot temperature, cold prompts, or black magic?. arXiv:2210.14699. URL https://arxiv.org/abs/2210.14699.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M., 2020. CodeBERT: A pre-trained model for programming and natural languages. arXiv:2002.08155.

Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., tau Yih, W., Zettlemoyer, L., Lewis, M., 2023. InCoder: A generative model for code infilling and synthesis. arXiv:2204.05999.

2023. GitHub - java-deobfuscator/deobfuscator: The real deal — github.com. https://github.com/java-deobfuscator/deobfuscator/graphs/contributors, https://github.com/java-deobfuscator/deobfuscator. (Accessed 27 February 2025).

2024. GitHub - jython/jython: Python for the java platform — github.com. https://github.com/jython/jython/graphs/contributors, https://github.com/jython/jython. (Accessed 27 February 2025).

2025. GitHub copilot · your AI pair programmer — github.com. https://github.com/features/copilot. (Accessed 05 February 2025).

2025a. GitHub - col-e/recaf: The modern java bytecode editor — github.com. https://github.com/Col-E/Recaf/graphs/contributors, https://github.com/Col-E/Recaf. (Accessed 27 February 2025).

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S.K., Clement, C., Drain, D., Sundaresan, N., Yin, J., Jiang, D., Zhou, M., 2021. GraphCodeBERT: Pre-training code representations with data flow. arXiv:2009.08366.

Henderson, T., 2021. Zss: A python library for comparing and matching hierarchical structures. URL https://pypi.org/project/zss/1.1.4/, Version 1.1.4.

Hindle, A., Barr, E.T., Gabel, M., Su, Z., Devanbu, P., 2016. On the naturalness of software. Commun. ACM 59 (5), 122–131. http://dx.doi.org/10.1145/2902362.

Hoq, M., Shi, Y., Leinonen, J., Babalola, D., Lynch, C., Price, T., Akram, B., 2024. Detecting ChatGPT-generated code submissions in a CS1 course using machine learning models. In: Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1. In: SIGCSE 2024, Association for Computing Machinery, New York, NY, USA, pp. 526–532. http://dx.doi.org/10.1145/3626252.3630826.

Huang, D., Qing, Y., Shang, W., Cui, H., Zhang, J.M., 2024. EffiBench: Benchmarking the efficiency of automatically generated code. arXiv:2402.02037. URL https://arxiv.org/abs/2402.02037.

Inc., T.S.S., 2025. Source code converters — tangiblesoftwaresolutions.com. https://www.tangiblesoftwaresolutions.com/. (Accessed 27 February 2025).

Jesse, K., Ahmed, T., Devanbu, P.T., Morgan, E., 2023. Large language models and simple, stupid bugs. In: 2023 IEEE/ACM 20th International Conference on Mining Software Repositories. MSR, pp. 563–575. http://dx.doi.org/10.1109/MSR59073.2023.00082.

Jiang, L., Su, Z., 2009. Automatic mining of functionally equivalent code fragments via random testing. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis. ISSTA '09, Association for Computing Machinery, New York, NY, USA, pp. 81–92. http://dx.doi.org/10.1145/1572272.1572283.

LeetCode, 2025. Problems, URL https://leetcode.com/problemset/all/.

Li, R., Allal, L.B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., Liu, Q., Zheltonozhskii, E., Zhuo, T.Y., Wang, T., Dehaene, O., Davaadorj, M., Lamy-Poirier, J., Monteiro, J., Shliazhko, O., Gontier, N., Meade, N., Zebaze, A., Yee, M.-H., Umapathi, L.K., Zhu, J., Lipkin, B., Oblokulov, M., Wang, Z., Murthy, R., Stillerman, J., Patel, S.S., Abulkhanov, D., Zocca, M., Dey, M., Zhang, Z., Fahmy, N., Bhattacharyya, U., Yu, W., Singh, S., Luccioni, S., Villegas, P., Kunakov, M., Zhdanov, F., Romero, M., Lee, T., Timor, N., Ding, J., Schlesinger, C., Schoelkopf, H., Ebert, J., Dao, T., Mishra, M., Gu, A., Robinson, J., Anderson, C.J., Dolan-Gavitt, B., Contractor, D., Reddy, S., Fried, D., Bahdanau, D., Jernite, Y., Ferrandis, C.M., Hughes, S., Wolf, T., Guha, A., von Werra, L., de Vries, H., 2023. StarCoder: may the source be with you!. arXiv:2305.06161.

Lo, D., 2023. Trustworthy and synergistic artificial intelligence for software engineering: Vision and roadmaps. arXiv:2309.04142.

MacIver, D.R., 2013. Welcome to hypothesis!00b6. URL https://hypothesis.readthedocs.io/en/latest/index.html#.

Margulieux, L., Denny, P., Cunningham, K., Deutsch, M., Shapiro, B.R., 2021. When wrong is right: The instructional power of multiple conceptions. In: Proceedings of the 17th ACM Conference on International Computing Education Research. In: ICER 2021, Association for Computing Machinery, New York, NY, USA, pp. 184–197. http://dx.doi.org/10.1145/3446871.3469750.

Mathew, G., Parnin, C., Stolee, K.T., 2020. SLACC. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. ACM, http://dx.doi.org/10.1145/3377811.3380407.

Mathew, G., Stolee, K.T., 2021. Cross-language code search using static and dynamic analyses. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. In: ESEC/FSE 2021, Association for Computing Machinery, New York, NY, USA, pp. 205–217. http://dx.doi.org/10.1145/3468264.3468538.

Mayer, P., Kirsch, M., Le, M.A., 2017. On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers. J. Softw. Eng. Res. Dev. 5, 1–33.

Milmo, D., Agenc, 2023. CHATGPT reaches 100 million users two months after launch. URL https://www.theguardian.com/technology/2023/feb/02/chatgpt-100-million-users-open-ai-fastest-growing-app.

Nguyen, N., Nadi, S., 2022. An empirical evaluation of GitHub copilot's code suggestions. In: Proceedings of the 19th International Conference on Mining Software Repositories. MSR '22, Association for Computing Machinery, New York, NY, USA, pp. 1–5. http://dx.doi.org/10.1145/3524842.3528470.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., Xiong, C., 2023. CodeGen: An open large language model for code with multi-turn program synthesis. arXiv:2203.13474.

Oh, S., Lee, K., Park, S., Kim, D., Kim, H., 2024. Poisoned ChatGPT finds work for idle hands: Exploring developers' coding practices with insecure suggestions from poisoned AI models. In: 2024 IEEE Symposium on Security and Privacy. SP, pp. 1141–1159. http://dx.doi.org/10.1109/SP54263.2024.00046.

OpenAI, 2023. GPT-4 technical report. arXiv:2303.08774.

2025. [Link]. URL https://platform.openai.com/docs/api-reference.

Patitsas, E., Craig, M., Easterbrook, S., 2013. Comparing and contrasting different algorithms leads to increased student learning. In: Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research. pp. 145–152.

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., Karri, R., 2021. Asleep at the keyboard? Assessing the security of GitHub copilot's code contributions. arXiv:2108.09293. URL https://arxiv.org/abs/2108.09293.

2025. ubprocess - Subprocess management, URL https://docs.python.org/3/library/subprocess.html.

Rittle-Johnson, B., Star, J.R., Durkin, K., 2020. How can cognitive-science research help improve education? The case of comparing multiple strategies to improve mathematics learning and teaching. Curr. Dir. Psychol. Sci. 29 (6), 599–609. http://dx.doi.org/10.1177/0963721420969365.

Roy, C.K., Cordy, J.R., 2007. A survey on software clone detection research. Queen' s Sch. Comput. TR 541 (115), 64–68.

Roy, C.K., Cordy, J.R., 2018. Benchmarks for software clone detection: A ten-year retrospective. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering. SANER, pp. 26–37. http://dx.doi.org/10.1109/SANER.2018.8330194.

Roy, C.K., Cordy, J.R., Koschke, R., 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Sci. Comput. Program. 74 (7), 470–495. http://dx.doi.org/10.1016/j.scico.2009.02.007, URL https://www.sciencedirect.com/science/article/pii/S0167642309000367.

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X.E., Adi, Y., Liu, J., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C.C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., Synnaeve, G., 2023. Code llama: Open foundation models for code. arXiv:2308.12950.

Saini, V., Farmahinifarahani, F., Lu, Y., Baldi, P., Lopes, C.V., 2018. Oreo: Detection of clones in the twilight zone. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. In: ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA, pp. 354–365. http://dx.doi.org/10.1145/3236024.3236026.

Sandoval, G., Pearce, H., Nys, T., Karri, R., Garg, S., Dolan-Gavitt, B., 2023. Lost at c: A user study on the security implications of large language model code assistants. arXiv:2208.09727. URL https://arxiv.org/abs/2208.09727.

Su, F.-H., Bell, J., Kaiser, G., Sethumadhavan, S., 2016a. Identifying functionally similar code in complex codebases. In: 2016 IEEE 24th International Conference on Program Comprehension. ICPC, pp. 1–10. http://dx.doi.org/10.1109/ICPC.2016.7503720.

Su, F., Bell, J., Kaiser, G.E., Sethumadhavan, S., 2016b. Identifying functionally similar code in complex codebases. In: 24th IEEE International Conference on Program Comprehension, ICPC 2016, Austin, TX, USA, May 16-17, 2016. IEEE Computer Society, pp. 1–10. http://dx.doi.org/10.1109/ICPC.2016.7503720.

Thummalapenta, S., Cerulo, L., Aversano, L., Di Penta, M., 2010. An empirical study on the maintenance of source code clones. Empir. Softw. Eng. 15, 1–34, URL https://api.semanticscholar.org/CorpusID:2279999.

Thunes, C., 2020. Javalang. URL https://pypi.org/project/javalang/.

Tian, H., Lu, W., Li, T.O., Tang, X., Cheung, S.-C., Klein, J., Bissyandé, T.F., 2023. Is ChatGPT the ultimate programming assistant – how far is it?. arXiv:2304.11938.

Ullmann, J.R., 1976. An algorithm for subgraph isomorphism. J. ACM 23 (1), 31–42.

Wasserman, L., 2013. Scalable, example-based refactorings with refaster. In: Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools. WRT '13, Association for Computing Machinery, New York, NY, USA, pp. 25–28. http://dx.doi.org/10.1145/2541348.2541355.

Wei, H., Li, M., 2017. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence. IJCAI-17, pp. 3034–3040. http://dx.doi.org/10.24963/ijcai.2017/423.

Xu, F.F., Alon, U., Neubig, G., Hellendoorn, V.J., 2022. A systematic evaluation of large language models of code. arXiv:2202.13169.

Yang, A., Yang, B., Hui, B., Zheng, B., Yu, B., Zhou, C., Li, C., Li, C., Liu, D., Huang, F., Dong, G., Wei, H., Lin, H., Tang, J., Wang, J., Yang, J., Tu, J., Zhang, J., Ma, J., Xu, J., Zhou, J., Bai, J., He, J., Lin, J., Dang, K., Lu, K., Chen, K., Yang, K., Li, M., Xue, M., Ni, N., Zhang, P., Wang, P., Peng, R., Men, R., Gao, R., Lin, R., Wang, S., Bai, S., Tan, S., Zhu, T., Li, T., Liu, T., Ge, W., Deng, X., Zhou, X., Ren, X., Zhang, X., Wei, X., Ren, X., Fan, Y., Yao, Y., Zhang, Y., Wan, Y., Chu, Y., Liu, Y., Cui, Z., Zhang, Z., Fan, Z., 2024. Qwen2 technical report. arXiv preprint arXiv:2407.10671.

Yao, Y., Duan, J., Xu, K., Cai, Y., Sun, Z., Zhang, Y., 2024. A survey on large language model (LLM) security and privacy: The good, the bad, and the ugly. High-Confid. Comput. 4 (2), 100211. http://dx.doi.org/10.1016/j.hcc.2024.100211, URL https://www.sciencedirect.com/science/article/pii/S266729522400014X.

Yeo, S., Ma, Y.-S., Kim, S.C., Jun, H., Kim, T., 2024. Framework for evaluating code generation ability of large language models. ETRI J. 46 (1), 106–117. http://dx.doi.org/10.4218/etrij.2023-0357, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.4218/etrij.2023-0357. URL https://onlinelibrary.wiley.com/doi/abs/10.4218/etrij.2023-0357.

Yepis, E., 2023. Hype or not? AI2019s benefits for developers explored in the 2023 developer survey. URL https://stackoverflow.blog/2023/06/14/hype-or-not-developers-have-something-to-say-about-ai/.

Zakeri-Nasrabadi, M., Parsa, S., Ramezani, M., Roy, C., Ekhtiarzadeh, M., 2023. A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges. J. Syst. Softw. 204, 111796. http://dx.doi.org/10.1016/j.jss.2023.111796, URL https://www.sciencedirect.com/science/article/pii/S0164121223001917.

Zhao, H., Chen, H., Yang, F., Liu, N., Deng, H., Cai, H., Wang, S., Yin, D., Du, M., 2023. Explainability for large language models: A survey. arXiv:2309.01029.

Azeeza Eagal received her Masters of Science in Computer Science from North Carolina State University and her Bachelor of Science in Computer Science and her Bachelor of Arts in Classics from Truman State University. She is currently pursuing a Ph.D. in Computer Science at North Carolina State University. Her research focuses on AI for Software Engineering.

Kathryn (Katie) Stolee, Ph.D., is an Associate Professor with tenure in the Department of Computer Science at North Carolina State University. Her research in software engineering combines program analysis (e.g., refactoring, semantic code search, code-to-code search) with human factors (e.g., comprehension, reuse, information seeking behavior) to help developers improve their code quality and understanding.

John-Paul Ore is an Assistant Professor in the Department of Computer Science at NC State University. John-Paul's research interests are broadly in the areas of software engineering, robotics, program analysis, and system testing using high-resolution physical simulators. His research combines field robotics and software engineering (SE).