Detecting Semantic Clones of Unseen Functionality

AUTHOR ACCEPTED MANUSCRIPT

Authors: Konstantinos Kitsios, Francesco Sovrano, Earl T. Barr, Alberto Bacchelli

Conference: 2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE), Seoul, Republic of Korea, 16-20 November 2025

DOI: 10.1109/ASE63991.2025.00112

IEEE required notice

© 2025 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

The manuscript begins on the following page.

Detecting Semantic Clones of Unseen Functionality

Konstantinos Kitsios
University of Zurich
Zurich, Switzerland
konstantinos.kitsios@uzh.ch

Francesco Sovrano
University of Zurich
Zurich, Switzerland
francesco.sovrano@uzh.ch

Earl T. Barr
University College London
London, UK
e.barr@ucl.ac.uk

Alberto Bacchelli
University of Zurich
Zurich, Switzerland
bacchelli@ifi.uzh.ch

***Abstract*—Semantic code clone detection is the task of detecting whether two snippets of code implement the same functionality (e.g., Sort Array). Recently, many neural models achieved near-perfect performance on this task. These models seek to make inferences based on their training data. Consequently, they better detect clones similar to those they have seen during training and may struggle to detect those they have not. Developers seeking clones are, of course, interested in both types of clones. We confirm this claim through a literature review, identifying three practical clone detection tasks in which the model's goal is to detect clones of a functionality even if it was trained on clones of different functionalities. In light of this finding, we re-evaluate six state-of-the-art models, including both task-specific models and generative LLMs, on the task of detecting clones of unseen functionality. Our experiments reveal a drop in F1 of up to 48% (average 31%) for task-specific models. LLMs perform on par with task-specific models without explicit training for clone detection, but generalize better to unseen functionalities, where F1 drops up to 5% (average 3%) instead.**

We propose and evaluate the use of contrastive learning to improve the performance of existing models on clones of unseen functionality. We draw inspiration from the computer vision and natural language processing fields where contrastive learning excels at measuring similarity between two objects, even if they come from classes unseen during training. We replace the final classifier of the task-specific models with a contrastive classifier, while for the generative LLMs we propose contrastive in-context learning, guiding the LLMs to focus on the differences between clones and non-clones. The F1 on clones of unseen functionality is improved by up to 26% (average 9%) for task-specific models and up to 5% (average 3%) for LLMs.

Data and material:* *https://doi.org/10.5281/zenodo.17238379

***Index Terms*—clone detection, AI4SE**

I. Introduction

Code clones are snippets of code that implement the same functionality; this functionality is sufficiently intricate that a developer prefers to reuse the code snippet rather than regenerate it from scratch [1, 2]. It is this latter constraint that separates clones from single statements or other short programming idioms. Clones are typically introduced through copy-and-paste [3]. Although clones can be used to reduce coupling and prepare subsequent development [4], they can also have negative consequences, including increased maintenance costs [3, 5], reduced software quality [6], increased risk of introducing bugs [3, 7], reduced code reusability [8], and difficulty in modifying the code [3].

If two code snippets are syntactically similar they are called *syntactic clones*, otherwise they are called *semantic clones* [3]. For example, a recursive and a dynamic-programming implementation of a function that returns the n^{th} Fibonacci number are semantic clones. Syntactic clone detection is typically solved with token-matching and tree-matching algorithms [9]. On the other hand, semantic clone detection is undecidable according to Rice's theorem [10], so we can only approximately solve it. Following the rise of deep learning, numerous neural clone detection models have been proposed for this purpose [11–19]. Neural clone detection models are trained in a supervised setting on task-specific datasets. The most widely used dataset is BigCloneBench (BCB) [20], which contains 43 functionalities. We refer to a *functionality* as a non-obvious sequence of statements to accomplish a clearly and compactly definable program task [1, 2], e.g., `Sort Array` or `Fibonacci`. For each one of the 43 functionalities, BCB contains both clone pairs (two code snippets that both implement the functionality) and non-clone pairs (two code snippets, only one of which implements the functionality).

For the training and evaluation of neural models, the functionalities in the dataset are split into training and test sets uniformly at *random*, leading to some clone pairs of the `Fibonacci` functionality ending up in the training set and some others in the test set (see Figure 1a). In this setting, state-of-the-art (SOTA) models achieve outstanding results of up to 97.8% F1 [14]. This near-perfect F1 concerns the task of detecting clones of functionalities whose clones also appear in the training set. For example, the models excel at the task of detecting clones of the `Fibonacci` functionality when they are trained on clones of the `Fibonacci` functionality. Although this task is meaningful, we investigate the importance of the task of detecting clones of the `Fibonacci` functionality when the model is trained on clones of functionalities *other than* `Fibonacci`. In the former case, we say that clones of the `Fibonacci` functionality are *seen during training* while in the latter case, *unseen during training*. Through an initial motivation study, we find that detecting clones of unseen functionality is a more practical, naturally-occurring task, yet, SOTA models are evaluated on clones of seen functionality.

This motivates us to evaluate and improve the performance of SOTA models on this task. To this aim, we evaluate the performance of three task-specific models and three generative LLMs on clones of unseen functionality, measuring an F1 of up to 71.5% for task-specific models and up to 69% for LLMs. LLMs are more robust at detecting clones of unseen functionalities, with an average F1 drop of 3% compared to

TABLE I: Summary of the datasets employed in our study.

Dataset	# unique funct.	# clone pairs	# non-clone pairs	Language
BCB_s	43	8.6M	258K	Java
$BCB_{s'}$	23	2300	2300	Java
SCB	-	1K	1K	C & Java
OJ	104	3K	47K	C

their F1 on seen functionalities, while the same drop is 31% for task-specific models.

The performance drop on clones of unseen functionality motivates us to devise techniques that increase the performance of existing models on this task. We propose and evaluate the use of contrastive learning (CL), drawing inspiration from the computer vision and natural language processing fields where measuring similarity between two objects emerges in tasks like face verification and sentence similarity. Recent work in these fields shows the superiority of CL [21] because its training explicitly optimizes for similarity [22–25]. Moreover, Koch et al. [26] showed that CL generalizes well to classes *unseen during training*. Thus, we investigate whether contrastive learning can increase the performance of models on clones of unseen functionality. We replace the final classifier of the task-specific models (e.g., CodeBERT trained for clone detection) with a contrastive classifier. For generative LLMs, we propose an analogous prompting technique that guides the LLM toward the differences between clone and non-clone pairs. Results show increased performance in 9 out of 12 experiments and no difference in 3 experiments. We qualitatively analyze an experiment where CL did not increase performance and find that CL led to learning a different, more strict definition of a clone. Our work led to the following main research contributions:

- empirical evidence of a significant performance drop of six SOTA models on clones of unseen functionality;
- a dataset and an evaluation methodology for evaluating future clone detectors on clones of unseen functionality;
- empirical evidence that CL in post-training improves the performance of state-of-the-art task-specific models on clones of unseen functionality;
- contrastive clone prompting, a novel prompting technique that improves the performance of state-of-the-art generative LLMs on clones of unseen functionality.

II. Background and Related Work

We present here the background of the datasets and models used in our experiments, and we also discuss related work. In Section IV-B, we justify the selection of these models among other state-of-the-art.

A. Background

Datasets. BigCloneBench (BCB) is the most widely used benchmark for clone detection (see Section III) and is built on Java. The initial version BCB_{v1} [20] starts with the authors selecting ten target functionalities frequently encountered in

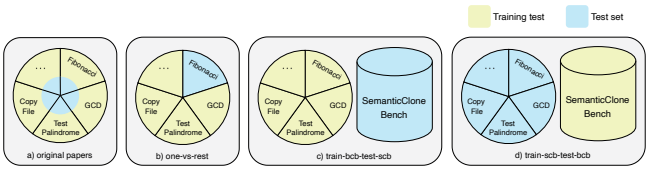


Fig. 1: Overview of the train/test splits used in four evaluation methods. Pie slices represent functionalities of the BCB dataset and the cylinder represents the SCB dataset. Although split a) is used in the literature, only the splits b), c), and d) measure the cross-functionality performance.

Java projects. For each functionality, they define search heuristics[1] and apply them to a large corpus of open-source Java code [27] to retrieve code snippets that *potentially* implement the functionality. The retrieved code snippets are manually inspected by humans to determine if they actually implement the intended functionality. Furthermore, the authors write their own code snippets that implement the intended functionality. Finally, all the pairs that implement the intended functionality are clone pairs while the rest of the pairs within the intended functionality are non-clone pairs. This results in 6.2M clone pairs and 258K non-clone pairs. BCB_{v2} [28] contains 43 functionalities with 8.6M clone pairs and 279K non-clone pairs. The large number of pairs and the class imbalance led researchers [13–15] to adopt a subset, which we call BCB_s, by sampling 20K clone and 20K non-clone pairs from BCB_{v2}.

SemanticCloneBench (SCB) [29] is built by mining Stack Overflow, considering two correct answers to the same question as a clone pair. It contains 4000 clone pairs uniformly distributed across four languages (Java, Python, C, C#).[2] The dataset does not contain non-clone pairs, so researchers [11] artificially created 1000 non-clone pairs by combining answers to different questions.

The OpenJudge (OJ) dataset [30] consists of correct solutions to 104 C problems (functionalities). Two programs that implement the same functionality are a clone pair; non-clone pairs are created by pairing programs that implement different functionalities. The version used in the literature consists of 50K pairs, 93% of which are non-clones [13, 15, 19, 31]. A summary of the datasets is shown in Table I.

Models. CodeBERT [32] is a general-purpose model for code. For semantic clone detection [11, 33, 34], pairs of snippets separated by a `[SEP]` token are given as input and the model outputs a vector representation for each pair. The representation is fed into a fully-connected layer with softmax activation [35] to predict if the snippets are clones or not.

ASTNN [13] is a code representation model designed to address the vanishing gradient issues caused by long ASTs. It achieves this by breaking down the AST of a given code snippet into smaller AST sequences, which are individually encoded and then fed into a bidirectional RNN that produces

[1]The list of heuristics is available in the PostgreSQL version of the dataset.
[2]We consider the 1000 Java pairs only, since BCB is also built on Java.

the representation. Despite using RNNs—that are outperformed by Transformers [36] in nearly every task [37]—it still achieves near SOTA performance on software engineering tasks. For semantic clone detection, the representations of the two snippets are subtracted and the result is fed into a fully-connected layer with sigmoid activation [35] to predict whether they are clones or not.

CodeGrid [14] is a code representation model that leverages the spatial information of code. A code snippet is encoded using a token encoder (e.g. Word2Vec [38]). The encoding of each token is placed in a grid that preserves the code layout, which is fed into a Convolutional Neural Network (CNN) to obtain the representation. For semantic clone detection, the representations of the two snippets are subtracted and the result is fed into a Support Vector Machine (SVM) classifier [39] to predict whether they are clones or not.

The three models above are *task-specific* classification models that are trained on clone datasets to classify clones. Recently, generative large language models (LLMs) have excelled in classification tasks without being explicitly trained on the classification objective [40, 41]. They achieve this through in-context learning where the models are prompted with a few examples of clones before presenting the pair under classification. We study three such models, shown in Table IV.

B. Related Work

We present here prior work related to contrastive learning for clone detection and generalizable clone detection.

Contrastive Learning for Clone Detection. Contrastive learning has been applied in prior clone detection research: ContraCode [42] is a pretrained code model trained with CL for clone detection. Sia-RAE [43] detects clones by linking two recursive autoencoders with a comparator network while Zubkov et al. [44] compare different paradigms of CL for clone detection. SJBCD [45] is a clone detector trained using CL on Java bytecode instead of raw code. C4 [46] combines CodeBERT with CL to detect clones across different programming languages, while CC2Vec applies CL on two self-attention layers to efficiently generate token representations. Contrastive learning has also been used in the pretraining stage, e.g., in UniXcoder [47] and ContraBERT [48], where it outperformed the baseline training method on clone detection.

The goal of these works is to develop effective CL architectures. As such, they are complementary to our goal, which is to increase the performance of existing models in the unseen functionality setting that naturally appears in real-world tasks. Our findings highlight the importance of these works even more, making CL a prominent go-to architecture for future clone detectors. Additionally, our work goes further than task-specific models by proposing a contrastive prompting technique for in-context learning with LLMs.

Generalizable Clone Detection. Three studies have investigated cross-functionality clone detection. Arshad et al. [11] aim to measure the cross-functionality performance of CodeBERT by training on all the functionalities of BCB and testing on SCB, reporting a drop in F1 of 41%. Liu et al. [31] partition the functionalities of OJ in 7 parts, train two AST-based models [12, 13] in 1 part and test it in the other 6, reporting weak cross-functionality performance. Khajezade et al. [49] also follow the 7-part setting and augment it with cross-language evaluation. Our contribution differs from these previous works in three ways. First, we present a novel systematic reasoning about the importance of cross-functionality performance in real-world applications, as we found no such reasoning in the literature. Second, we evaluate both task-specific models and generative LLMs, highlighting the differences in their generalization abilities. Third, we propose techniques to partially mitigate the observed performance drop, which only Khajezade et al. [49] do for task-specific models, while we also extend to generative LLMs.

Task-specific clone detectors have also been evaluated in cross-project [50], cross-dataset [51, 52], and cross-language [41, 53, 54] settings, which are complementary to cross-functionality: two different projects may share functionality, while many distinct functionalities exist within a single project, and the same holds for cross-language and cross-dataset.

III. Motivation: Clones of Unseen Functionality are Overlooked

Our work aims to evaluate and narrow the gap between semantic clone detection research and its practical applications. To this end, we study the occurrence of unseen functionalities—introduced informally in Section I and defined formally in Section IV—in (a) real-world tasks, and (b) the evaluation of SOTA models proposed in recent literature. We find that clones of unseen functionality appear in real-world tasks and yet have been overlooked in the literature.

First-Principles Argument. To investigate the relevance of detecting clones of unseen functionality in real-world applications, we begin with a first-principles combinatorial argument. We claim that when generating source code in the order of millions of lines of code (LOC), the creation of new, unseen functionalities is unavoidable. Researchers have found that clones make up a big proportion of software systems, ranging from 7% to 23% [55–58]. This large number of clones can be explained by only one of the following propositions:

(P1) many clones of *few* functionalities exist, or
(P2) many clones of *many* functionalities exist.

We claim that (P1) cannot hold: In the study of the uniqueness of source code [59], Gabel and Su show that (i) blocks of code with length greater than seven LOC are mostly "unique". From Definition 1 and Definition 2, two blocks of code are clones only if the functionality they implement is intricate enough that it makes sense for the developer to reuse it rather than rewrite it from scratch. Hence, (ii) we assume that most cloned functionalities in real-world projects are around or beyond the seven LOC threshold. Combining (i) and (ii), we conclude that most of the cloned functionalities in real-world projects are unique functionalities. Hence, we claim that (P2)

holds, and many cloned functionalities exist in software systems. Given that the number of functionalities in the training set of a clone detector ranges from 10 (BCB_{v1}) to 104 (OJ), we conclude that *functionalities unseen during training will dominate real-world software systems*, and deployed models will be required to detect clones of unseen functionalities.

Literature Review of Real-World Applications. We buttress our first-principles argument with evidence from our small-scale literature review on real-world applications of clone detection. We first search for papers whose title contains the keyword `clone` in the last ten years of ICSE, both in the main and Software Engineering in Practice tracks. Note that our goal is not an extensive literature review, but rather to provide evidence that, in real-world applications, a model should detect clones not only of seen functionalities, but also of new, unseen functionalities. We find 15 papers, of which we keep those that present an application of clone detection, discarding those that only evaluate a tool against a benchmark. This results in only one paper, so we apply the same criteria to the papers mentioned in two well-established surveys by Roy et al. [3] and Svajlenko et al. [60]. This leads to a total of three papers that satisfy our criteria, which we analyze here.

Ishihara et al. [61] run syntactic clone detection to a collection of 13K open-source Java projects. Their goal is to locate frequently cloned functionalities to implement them as external libraries. They found 56 such functionalities, an example being the sorting of a JTable, which was independently added to the Java SE 6. To succeed at this task, a clone detector should detect clones of table sorting algorithms even if it has not seen such clones in the training set.

In another application, Laguë et al. [62] studied communications software and found empirical evidence regarding the benefits of *Preventive Control*, which involves the use of a clone detector in the development process to prevent new clones from entering the system. In the modern code review process [63], this is implemented by integrating a clone detector in the pipeline before merging a pull request and many commercial tools support it (e.g., Sonar). If the clone detector only detects clones of functionalities it has seen in the training, the reported performance of up to 98% F1 is an overestimation.

Microsoft used syntactic clone detection to find security vulnerabilities [64, 65]. Once developers discovered a vulnerability, they searched for clones of the vulnerable code. For example, developers discovered a snippet that could cause potential heap corruption. After searching for clones of this snippet, they found three more snippets that could cause similar heap corruptions. The clone detector should not only find clones of known vulnerable snippets but also unseen ones to be useful in this scenario. Although the study focused on syntactic clones, the authors stated their intention to pursue semantic clones in future work [65]. The evidence from our literature review provides substantial support to our first-principles reasoning, and by combining them we reach the following observation:

Observation 1. We find three applications where detecting clones of unseen functionality is the core task.

Evaluation Methods Used in the Literature. Then, we investigate the evaluation methods applied in the literature. We search in the last five years of the ICSE for neural models that achieve SOTA performance on clone detection, and also include models from top software engineering venues that reference them as a baseline and outperform them. We find nine models whose details are shown in Table II. We group the models by their input format: text-based models extract representations from raw code, while AST-based models feed the Abstract Syntax Tree (AST) of the code into RNNs or Graph Neural Networks for this purpose. Image-based approaches extract representations by feeding the image signal of code into CNNs. Finally, GraphCodeBERT [34] accepts hybrid input, i.e., it combines text information with data flow.

By analyzing the evaluation methods used in the nine models of Table II, we extract the following information. The most frequently used dataset is BCB (9 papers), followed by OJ (4 papers) and GoogleCodeJam (GCJ) [66] (1 paper). All three datasets are functionality-based, i.e., they consist of a number of functionalities (43, 104, 12, respectively) and contain clones of each functionality. *All the models are evaluated on the task of detecting clones of seen functionalities*: the datasets are split uniformly at random into training and test sets, resulting in functionalities of the test set appearing in the training set, as shown in Figure 1a.

Observation 2. The nine studied state-of-the-art models are evaluated on the task of detecting clones of *seen* functionality.

Our systematic search of the last five years of ICSE did not yield any generative LLMs, however, such models are effective in classification tasks through in-context learning [67], even surpassing task-specific models at times [41]. Since the relevance of LLMs for software engineering tasks is rising, we take up the task of studying the effect of unseen functionalities in the in-context examples of LLMs as well, laying the ground for future evaluations of LLMs for clone detection.

IV. Methodology

First, we formally define the main concepts of this paper.

Definition 1. *A **functionality** is a non-obvious sequence of code statements to accomplish a clearly and compactly definable task.*

Examples of functionalities are `Fibonacci` (calculate the n^{th} Fibonacci number), `GCD` (Calculate the Greatest Common Divisor of two integers), or `Test Palindrome` (check if a string reads the same from both ends). The term *"non-obvious"* differentiates a functionality from single statements or short programming idioms. This makes a functionality intricate enough that a developer prefers to reuse the code snippet rather than regenerate it from scratch.

Definition 2. *Two code snippets form a **clone pair** iff they implement the same functionality.*

A member of a clone pair is called a clone. By negation, two code snippets that do not implement the same functionality form a non-clone pair. While intuitive, this definition of a clone is undecidable because it rests on code equivalence [10]. It also leaves intentionally imprecise whether two snippets are clones when they implement unshared—in addition to shared—behavior, an issue we take up in Section V-C. It does, however, align with the definition used in the literature [20].

Definition 3. *A clone in the test set is a clone of an **unseen functionality**, if the model has not explicitly learned to detect clones of this functionality.*

The term *learning* in the above definition includes both training for *task-specific models* and in-context learning for *LLMs*. For task-specific models, if the training set contains pairs of the functionalities `Test Palindrome` and `GCD` and the test set contains pairs of `Fibonacci`, we say that the latter are clones of an unseen functionality. This definition considers as unseen a functionality also in the case where a snippet that implements it appeared in the pretraining corpus of LLMs (e.g., GPT-4o). However, the LLMs must not have learned *explicitly* to classify clone pairs of the said functionality, e.g., through finetuning or in-context learning.

We refer to the task of detecting clones of unseen functionality as *cross-functionality* clone detection and to the performance of models in this task as cross-functionality performance.

A. Research Questions

The study that motivates this work, in Section III, reveals that (a) detecting clones of unseen functionality occurs in applications and (b) SOTA models are evaluated on clones of *seen* functionality. This motivates the evaluation of models' cross-functionality performance. To this aim, we evaluate three task-specific models and three generative LLMs on settings where the clone pair under classification comes from a functionality unseen in the finetuning dataset (for task-specific models) or the in-context examples (for generative LLMs).

RQ1. How well do state-of-the-art models detect clones of unseen functionality?

To conclude, we propose and evaluate the use of contrastive learning to improve cross-functionality performance. We draw inspiration from adjacent research areas where CL showcases impressive performance in measuring similarity between two objects, even when they come from classes unseen during training. We adapt three task-specific models by replacing their final classifier with a contrastive classifier and also propose an analogous prompting technique for LLMs, called contrastive in-context learning, to answer our last research question.

TABLE II: Task-specific model's performance on clones of seen functionality. The models we evaluate in this work appear highlighted.

Model	Input format	F1	P	R	A
CodeBERT [11]	Text	95.0	99.0	91.0	95.0
Toma [18]	Text	90.0	93.0	88.0	-
ASTNN [13]	AST	93.8	99.8	88.3	-
xASTNN [19]	AST	96.6	**99.9**	93.5	-
FA-AST [66]	AST	95.0	96.0	94.0	-
CDLH [12]	AST	82.0	92.0	74.0	-
WySiWiM [15]	Image	94.8	95.3	94.3	-
CodeGrid [14]	Image	**97.8**	99.6	**96.1**	-
GraphCodeBERT [34]	Hybrid	95.0	94.8	93.4	-

RQ2. To what extent does contrastive learning improve model performance on clones of unseen functionality?

B. RQ1 — Cross-functionality Performance

The results of Section III motivate us to quantify the cross-functionality performance of SOTA models. To do so, we propose four evaluation methods designed to evaluate the models on clones of unseen functionality. Model performance is measured by accuracy, F1 score, precision, and recall.

Model Selection. Experimenting with all nine models of Table II would be computationally intensive, so we group them by input format and select one model from each group. CodeBERT is the best-performing *text-based model* so we select it. From the four *AST-based models*, CDLH and FA-AST are evaluated on a version of BCB that was later marked as problematic by Krinke et al. [68], who found that the version contains incorrectly labeled non-clone pairs. From the remaining two models, xASTNN does not come with a replication package, so we select ASTNN. From the *image-based models*, we select the best performer (CodeGrid). Finally, GraphCodeBERT was also evaluated on the same problematic version of BCB [68], so we do not use it in our experiments. We end up with three models, which are highlighted in Table II. Regarding generative LLMs, we select three SOTA models shown in Table IV, covering commercial, open-source, and reasoning models.

Dataset Selection. The models are evaluated on the BCB dataset in their original papers, so the first dataset we use is BCB. To increase the generality of our results, we also use the OJ dataset (Section V-C). In the literature, different deviations from the original BCB are used. For example, three papers evaluate CodeBERT for code clone detection, with two of them [33, 34] using the problematic version of BCB [68]. Hence, for this work, we adopt the results from the third paper [11] that uses a custom subset of BCB_{v2} of size 1000.

ASTNN and CodeGrid are evaluated on BCB_s which was sampled from BCB_{v2}. BCB_{v2} is imbalanced in terms of functionalities: Although it contains 43 functionalities, 70% of the non-clone pairs and 54% of the clone pairs belong to the `Copy File` functionality alone [68]. By analyzing BCB_s we come to similar findings. The `Copy File` functionality makes up 60% of the dataset and the three most frequent

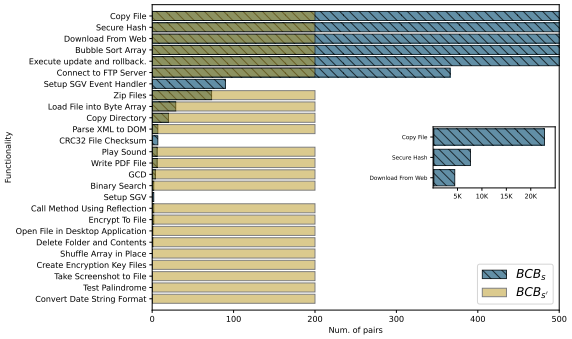


Fig. 2: Number of pairs in each functionality of the two BCB dataset versions. The top-3 functionalities (also shown in the inset axis) make up 90% of BCB_s.

functionalities make up 90% of the dataset. Moreover, only 10 out of the 43 functionalities have at least ten pairs in BCB_s. The full distribution is shown in blue in Figure 2.

To mitigate the functionality imbalance, we introduce $\text{BCB}_{s'}$ that is obtained from BCB_{v2} by performing functionality-aware sampling and thresholding: If a functionality has more than N pairs, we sample N of them uniformly at random so that functionalities like `Copy File` do not overshadow the others. If a functionality has fewer than N pairs, we discard it. Finally, we ensure class balance by selecting $M = \frac{N}{2}$ clone pairs and M non-clone pairs, similarly to BCB_s. By selecting $M = 100$, the thresholding retains 23 functionalities, while for larger M the number of functionalities decreases steeply. Thus, the dataset contains 100 clone pairs and 100 non-clone pairs for each of the 23 functionalities, for a total of 4600 pairs. The new distribution of the retained functionalities is shown in Figure 2. We use a random seed for the sampling and publicly release the code for creating the dataset to ensure reproducibility. $\text{BCB}_{s'}$ mitigates the imbalance threat that comes with BCB_s [68] so we use this version of BCB in our experiments, and also encourage its use in future research.

Evaluating Task-Specific Models. We propose the three evaluation methods of Figure 1 to capture cross-functionality performance of task-specific models that require explicit training. We start with the *one-vs-rest* evaluation. Given a dataset with F functionalities denoted by $\mathcal{F} = \{1, 2, ..., F\}$, we propose a sequence of F experiments where, in experiment i, the training set consists of code pairs that implement the functionalities $\mathcal{F} - \{i\}$ and the test set consists of code pairs that implement the functionality i. $\text{BCB}_{s'}$ has $F = 23$ functionalities, so we obtain 23 tuples of (accuracy, F1, precision, recall) for each model. To compare the cross-functionality performance with the performance on clones of seen functionality, we employ the *one-sample Wilcoxon signed-rank test* [69] with the alternative hypothesis that the 23 F1 values measured by the *one-vs-rest* evaluation are lower than the single value reported in the original papers.

The *one-vs-rest* evaluation satisfies the condition that functionalities in the test set do not appear in the training set; however, all functionalities come from the $\text{BCB}_{s'}$ dataset. To account for potential within-dataset biases, we also employ the *train-bcb-test-scb* evaluation that uses all the functionalities of $\text{BCB}_{s'}$ for training and tests the trained model on the SCB dataset. We follow the exact permutation of Arshad et al. [11] to create non-clone pairs for SCB by combining code snippets that belong to different clone pairs, because they manually verified a fraction of the non-clone pairs. The resulting dataset is balanced and contains 2000 pairs. Finally, we employ the complementary method *train-scb-test-bcb* where the model is trained on SCB and tested on $\text{BCB}_{s'}$.

The hyperparameters of the models are kept unchanged for comparison with the original papers. Moreover, we only train the last layer of CodeBERT to avoid overfitting given the relatively small dataset size (e.g., 2K for SCB). We call the models CodeBERT_B, ASTNN_B, and CodeGrid_B respectively, where *B* stands for Baseline, to distinguish them from the contrastive learning variations (RQ2) and report the results in Section V. Experiments were conducted on a Linux machine with an NVIDIA Tesla T4 GPU (16 GB memory) and an AMD CPU (4 cores, 32 GB memory).

Evaluating Generative LLMs. The ability of generative LLMs to perform on par with task-specific models in classification comes with substantially higher cost. To keep the cost of our LLM experiments viable, we run an a priori power analysis ($\alpha = 0.05, power = 0.8$) using Fisher's exact test to detect a small effect size. The analysis determined that 404 predictions are required to detect a change between two LLM prompts, i.e., baseline and contrastive. This dataset size is typical in adjacent software engineering tasks [70, 71] that evaluate LLMs. We sample 404 pairs of the $\text{BCB}_{s'}$ dataset uniformly at random and use them to evaluate the LLMs.

LLMs are used in classification tasks like clone detection through in-context learning [40]: they are prompted with clone and non-clone examples along with the new pair to classify. To evaluate LLMs on clones of *seen* functionality, we select the clone examples to have the same functionality as the pair under classification, while to evaluate them on *unseen* functionality we select the clone examples from different functionalities. We note that according to Definition 3, a functionality is unseen even if snippets implementing it were present in the pretraining stage of the LLMs; however, the LLM must not have explicitly learned to classify clones of these functionalities, e.g., through in-context learning. We provide one clone and one non-clone pair to keep the prompt length small and use a temperature of 0 to enable reproducibility, following previous work [72]. We refer to this prompt as *baseline*, because the clone and non-clone examples are selected uniformly at random, unlike the contrastive prompt in Section IV-C. An outline of the prompt is shown in Figure 4 (excluding the blue line), and we provide the full prompt in our replication package [73]. Through a system command, we required the LLM to generate the explanation prior to the decision, enabling the use of the explanation as

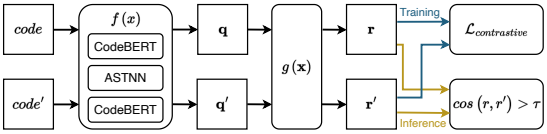


Fig. 3: The contrastive learning architecture we use for code clone detection.

```
Determine whether {c1},{c2} is a clone pair or not,
based on the following definition: {Definition}
Clone example: {x},{y}
Non-clone example: {z},{w} # only in baseline
Non-clone example: {x},{w} # only in contrastive
```

Fig. 4: Prompt outline. In contrastive clone prompting, the clone and the non-clone example share the snippet **x**.

a means of self-correction [74]. Since our focus is on unseen functionalities and not on devising the optimal prompt, we do not experiment with alternative prompting schemes. We present the results in Section V.

C. RQ2 — Contrastive Learning Improvements

Contrastive learning [21] is a neural network architecture designed for similarity learning that excels at identifying similar objects while discriminating between dissimilar ones. It explicitly maximizes the similarity of the vector representations of semantic clone pairs, in contrast to the baseline architectures described in Section II that implicitly maximize this similarity, because their main objective is to minimize the classification loss function, hence being more prone to overfitting patterns seen during training [75]. CL is applicable to task-specific models that are trained on task-specific classification datasets; however, an analogous prompting technique called *contrastive prompting* or *in-context contrastive learning* has emerged for in-context learning with LLMs [70]. We discuss both techniques in more detail below.

Contrastive Learning. Contrastive learning training forces the representations of similar objects to converge in the representation space while driving the representations of dissimilar objects apart. We adopt the CL architecture shown in Figure 3 because it is straightforward and it was successfully used for clone detection before [44]. We do not experiment with other architectures since our aim is to demonstrate the effectiveness of CL in detecting clones of unseen functionality, and we and hope future work will elaborate on this.

The function $f(x)$ embeds the two snippets, $code$ and $code'$, in a representation space. Both $f(x)$, $code$, and $code'$ are model-agnostic, hence CL can be applied regardless of the model or the input format (text, AST, image). Consequently, if $f(x)$ captures semantic similarity, then CL captures semantic similarity too. The intermediate representations $\mathbf{q}$ and $\mathbf{q}'$ are then propagated through $g(\mathbf{x})$, which is a hyperparameter. The resulting representations $\mathbf{r}$ and $\mathbf{r}'$ are fed into the contrastive loss function that is minimized in the training phase:

$$\mathcal{L} = \frac{1}{2N}\sum_{i=1}^{N} y_i \left(\|\mathbf{r}_i - \mathbf{r}'_i\|\right)^2 + (1-y_i)\left[\max(0, m - \|\mathbf{r}_i - \mathbf{r}'_i\|)\right]^2 \quad (1)$$

In Equation (1),

- the subscript i denotes the i^{th} pair of the dataset,
- y_i is 1 if $code_i$ and $code'_i$ are clones and 0 otherwise,
- N is the number of pairs in the dataset,
- $\|\cdot\|$ denotes the Euclidean norm, and
- m is the margin that controls the minimum distance between dissimilar pairs and is a hyperparameter.

From Equation (1) we see that in the training phase, the representations of clone pairs ($y_i = 1$) are attracted while those of non-clone pairs ($y_i = 0$) are repelled. In the inference phase, the prediction for a new pair is given by Equation (2):

$$\hat{y_i} = \begin{cases} 0, & \text{if } \cos(\mathbf{r}_i, \mathbf{r}'_i) < \tau \\ 1, & \text{if } \cos(\mathbf{r}_i, \mathbf{r}'_i) \geq \tau \end{cases} \quad (2)$$

where $\cos(\mathbf{a}, \mathbf{b})$ denotes the cosine similarity and the threshold τ is set to $\tau = 0.5$ following previous work [11, 13–15].

We replace the final classifier of the three task-specific models with the contrastive classifier of Figure 3 and re-run the experiments of Section IV-B. We call the resulting models CodeBERT$_{CL}$, ASTNN$_{CL}$ and CodeGrid$_{CL}$ respectively. We determine the hyperparameters m and $g(\mathbf{x})$ using grid search while the rest of the hyperparameters remain the same as the baseline models. The grid search consists of three values for $m \in \{0.5, 5, 50\}$, covering the range of small, moderate, and big values, and two values for g: $g(\mathbf{x}) = \mathbb{1}$ (identity function) and $g(\mathbf{x}) = BN$ (Batch Normalization), with the best combination being ($m = 0.5$, $g(\mathbf{x}) = BN$).

Contrastive In-Context Learning for LLMs. Equation (1) cannot be used with generative LLMs that are trained to predict the next token in a sequence. Instead, we use *contrastive in-context learning*, a technique used in related tasks like vulnerability detection [70] where the LLM is provided with both a vulnerable code snippet and its fixed version before asking it to classify a new code as vulnerable or not. This way, the LLM focuses on the differences between vulnerable and non-vulnerable code, similar to contrastive learning. Steenhoek et al. [70] find that contrastive in-context learning performs better than all other prompting schemes in vulnerability detection. We adapt it for clone detection by providing one clone pair and one non-clone pair that share a common snippet, guiding the LLM to focus on the differences. Our baseline and contrastive prompts are summarized in Figure 4 and the full prompts are available in our replication package [73].

V. Results

In this section, we answer our research questions.

A. RQ1 — Cross-functionality Performance

We measure the cross-functionality performance of the selected models and compare it with their performance on the task of detecting clones of seen functionalities.

TABLE III: Task-specific model performance across different evaluation methods (columns) on the BCB and SCB datasets. Models perform significantly worse on unseen functionalities (last three columns), and contrastive learning (CL) partially mitigates this drop.

Evaluation →	on seen functionalities				one-vs-rest				train-bcb-test-scb				train-scb-test-bcb			
Model ↓	A	F1	P	R	A	F1	P	R	A	F1	P	R	A	F1	P	R
CodeBERT$_B$ [11]	95.0	95.0	99.0	91.0	82.3	85.5	**90.6**	84.3	49.8	65.7	**96.1**	49.9	50.9	63.2	**84.4**	50.6
CodeBERT$_{CL}$					**82.6**	**85.7**	89.2	**86.3**	**84.8**	**84.1**	89.0	**81.0**	**67.1**	**67.6**	68.8	**66.5**
ASTNN$_B$ [13]		93.8	99.8	88.3	**70.8**	**77.3**	70.5	**90.7**	48.7	65.2	49.4	**96.2**	51.0	42.7	51.4	36.5
ASTNN$_{CL}$					70.6	73.6	**72.8**	79.7	**62.3**	**70.7**	**57.8**	91.2	**58.2**	**68.3**	**55.0**	**90.1**
CodeGrid$_B$ [14]		97.8	99.6	96.1	56.0	66.7	54.1	**88.0**	51.0	63.8	50.6	86.3	52.1	49.8	52.3	35.5
CodeGrid$_{CL}$					**70.3**	**75.6**	**70.2**	87.0	**51.3**	**64.4**	**50.8**	**88.0**	**64.0**	**67.2**	**61.7**	**73.7**

TABLE IV: LLM performance on seen and unseen functionalities of the BCB dataset. LLMs perform worse on unseen functionalities, a drop that contrastive in-context learning (CL) partially mitigates. $p < .05$ in F1 between baseline and CL is indicated with *.

Evaluation →	on seen functionalities				on unseen functionalities			
Model ↓	F1	A	P	R	F1	A	P	R
llama-3.3-70b-versatile$_B$	0.63	0.66	0.7	0.57	0.61	0.68	**0.78**	0.50
llama-3.3-70b-versatile$_{CL}$					**0.66***	**0.69**	0.73	**0.60**
gpt-4o-2024-08-06$_B$	0.64	0.68	0.73	0.57	0.59	0.63	0.70	0.51
gpt-4o-2024-08-06$_{CL}$					**0.62***	**0.64**	**0.66**	**0.59**
deepseek-r1-distill-qwen$_B$	0.70	0.70	0.69	0.70	0.68	**0.68**	**0.68**	0.68
deepseek-r1-distill-qwen$_{CL}$					**0.69**	0.67	0.65	**0.73**

Task-Specific Models. The evaluation methods *train-bcb-test-scb* and *train-scb-test-bcb* consist of one experiment each (see Figure 1) and the performance of the baseline models CodeBERT$_B$, ASTNN$_B$ and CodeGrid$_B$ is shown in Table III. The comparison of the "*train-bcb-test-scb*" and "*train-scb-test-bcb*" columns of Table III with the "*on seen functionalities*" column reveals that the performance of all three models drops up to 48% in F1 (average 31%) when the functionalities of the test set do not appear in the training set.

The *one-vs-rest* evaluation method consists of 23 experiments (Figure 1b); hence, we have 23 tuples of (accuracy, F1, precision, recall) for each model. In Table III, we report the average of the 23 experiments and provide the full distribution in our replication package [73]. By employing the *one-sample Wilcoxon signed-rank test* [69], we find that the F1 in the 23 experiments is lower than the F1 reported in the original papers ($p < 0.01$). The same holds for accuracy, while for precision and recall there are cases where the models have low predictive power, leading to nearly 100% precision with 50% recall or the opposite. Since the trade-off between precision and recall can be controlled through the hyperparameter τ of Equation (2), we refrain from studying precision and recall independently and study the F1 that combines the two [76]. The best-performing task-specific model is CodeBERT, with an average F1 of 71.5% across the three evaluation methods.

Generative LLMs. We present the LLM results in Table IV, where the models with suffix `B` indicate the baseline prompt. The best-performing LLM on unseen functionalities is DeepSeek with 69% F1, on par with task-specific models. All models perform worse on unseen functionalities compared to seen ones, with an average drop of 3% and a maximum drop of 5%; the reasoning model DeepSeek is affected the least. The performance drop is smaller than the task-specific models (average 31%). This is to be expected: LLMs have orders of magnitude more parameters and are pretrained in a corpus that probably contains many of the functionalities in the dataset, although as unstructured text rather than clone pairs. Yet, the F1 still drops up to 5% when the in-context learning examples come from a different functionality.

> **Finding 1**. Task-specific models achieve an F1 of up to 71.5% and LLMs up to 69% on unseen functionalities. LLMs generalize better to unseen functionalities, experiencing an average F1 drop of 3%, compared to task-specific models that experience 31% avg. F1 drop.

B. RQ2 — Contrastive Learning Improvements

We investigate the use of contrastive learning to improve the performance of existing models on unseen functionalities. Table III shows the results for the *train-bcb-test-scb* and *train-scb-test-bcb* evaluation methods. We compare the baseline variation of each model with its respective CL variation, where the best-performing variation is shown in bold. Results indicate that the CL variation outperforms the baseline in F1 and accuracy in all six experiments. The largest improvement is for ASTNN under the *train-scb-test-bcb* method where contrastive learning increased the F1 by 26%. The two cases where a baseline has higher precision is because of low predictive power: the F1 of the CL variation is higher in both cases. The same holds for one case where a baseline has higher recall.

The average of each measure for the *one-vs-rest* method is shown in Table III, while the full distribution is available in our replication package [73]. To compare the CL variation with its baseline counterpart, we employ the *Wilcoxon signed-rank test* [69]. We find that CL outperforms the baseline in the case of CodeGrid ($p < 0.01$) both in accuracy and F1 while for CodeBERT and ASTNN there is no statistically significant difference. Overall, we find that the CL variation performs better than the baseline in seven out of nine experiments, whereas in two experiments performance is unchanged.

Finally, Table IV shows the results of our generative LLM experiments, where the `CL` subscript indicates the contrastive prompt. To test if the contrastive prompt performs better than the baseline one, we run the Wilcoxon signed-rank test over 10-fold cross-validation F1 scores [77]. We find statistically significant improvement for gpt-4o ($p = .02$, $r = .62$) and llama ($p = .04$, $r = .55$), while DeepSeek experiences no significant improvement, which we attribute to its advanced reasoning. The average increase in F1 is 3%, smaller than the task-specific models (9%).

Finding 2. Contrastive learning increases the F1 of task-specific models on clones of unseen functionality by up to 26% (avg. 9%). Contrastive in-context learning increases the F1 of llama (5%) and gpt-4o (3%), with DeepSeek being unaffected.

We manually investigate the only case where CL performs worse than the baseline in terms of average F1, although the difference is still not statistically significant and could be just an effect of random noise. This happens to the ASTNN model under the *one-vs-rest* evaluation. We select three functionalities uniformly at random, namely `Copy File`, `GCD`, and `Test Palindrome`, and analyze the three experiments where the test set contains only these functionalities. We examine the pairs in the test set which ASTNN$_B$ correctly classified and ASTNN$_{CL}$ incorrectly classified. We find 40 (20%) such pairs for `Copy File`, of which 37 are predicted to be non-clones by ASTNN$_{CL}$ while their ground truth is clones (false-negatives).

By manually inspecting the 37 false-negatives, we find that 33 of them are non-clones under the Krinke et al. [68] strict definition of clones which requires clones to implement some functionality as their main or only purpose. We discuss this definition at greater length in Section V-C. For example, in a clone (according to BCB ground truth) pair, one snippet copies a file from one location to another while the second snippet additionally imports a certificate into a Java KeyStore, which involves a series of file operations. ASTNN$_{CL}$ classified this pair as a non-clone and a similar pattern is followed in 33 out of the 37 false-negatives. By inspecting the pairs for the `GCD` and `Test Palindrome` functionalities, *we find that all the pairs labeled as clones in the dataset are actually clones under the strict definition*. Moreover, ASTNN$_{CL}$ outperforms ASTNN$_B$ when the test set contains these two functionalities. This provides evidence for why CL did not improve ASTNN performance in the *one-vs-rest* evaluation: it learned a stricter clone definition while the test set uses a more permissive definition for some functionalities, like `Copy File`. In Section VI, we propose areas for future work based on these qualitative results.

Another result comes from the interpretation of the *one-vs-rest* experiments, which measure how well models can detect clones of functionality f given that they have never seen clones of f in training. For example, CodeBERT$_{CL}$ achieves 95% accuracy when tested on the functionality `Shuffle Array in Place`, while ASTNN$_{CL}$ and CodeGrid$_{CL}$ achieve 73% and 52% respectively. Similarly, ASTNN$_{CL}$ is the only model that achieves near-perfect performance when tested on the `Open URL in System Browser` functionality with an accuracy of 96% and CodeGrid$_{CL}$ is the only model that performs strongly on the `Secure Hash` functionality with an accuracy of 94%. This observation suggests that the inherently complex nature of semantic clones may not be fully captured with a single code representation.

TABLE V: Performance of the three task-specific models on the OJ dataset. The contrastive learning variation outperforms the baseline variation in all three models in F1.

Evaluation →	train-oj-test-scb				train-scb-test-oj			
Model ↓	A	F1	P	R	A	F1	P	R
CodeBERT$_B$ [11]	56.6	27.9	16.8	82.0	**87.7**	17.9	20.5	**16.0**
CodeBERT$_{CL}$	**79.0**	**76.1**	**66.6**	**88.7**	69.2	**23.3**	**71.0**	13.9
ASTNN$_B$ [13]	66.4	59.5	**74.9**	49.3	75.6	23.1	14.5	**55.7**
ASTNN$_{CL}$	**72.1**	**70.5**	74.7	**66.7**	**91.8**	**37.3**	**37.5**	37.2
CodeGrid$_B$ [14]	50.0	**66.7**	50.0	**100**	57.5	14.5	8.4	**55.0**
CodeGrid$_{CL}$	**54.0**	57.9	**53.4**	63.1	**68.7**	**14.9**	**9.0**	42.8

C. Evaluating on Strict Clones

To strengthen our findings, we want to generalize our results to datasets other than BCB, which uses a partial definition of clones. Definition 2 leaves what constitutes the *same functionality* intentionally imprecise. BCB considers two snippets to implement the same functionality (and therefore to be clones) when they both implement one of the 43 predefined functionalities, regardless of any additional functionalities those snippets may implement. For example, consider two code snippets: the first one copies a file to a local directory; the second one opens an FTP connection, copies a file to the FTP server, and closes the FTP connection. According to BCB, the two snippets form a clone pair.

Krinke et al. [68] consider another, much stricter definition of clones: *two snippets form a clone pair if they implement functionality f as their main or only purpose.* Armed with this definition, they manually examined 100 clones of the `Copy File` functionality to check whether they are still clones. They find that 86% of the pairs would not be considered clones under this strict definition. They only inspected clones of the `Copy File` functionality, which is usually seen as part of bigger methods, in contrast to other functionalities of BCB like `Fibonacci` or `GCD`, which are more likely to be seen as standalone methods as we observed in Section V-B.

There is no universally accepted, decidable definition of clone, perhaps as a consequence of treading in undecidable waters. For instance, Krinke *et al.*'s strict definition leaves open the question of just what constitutes "main or only". Perhaps to avoid difficulties such as these, some think the definition should depend on the end-user application [55]. In any event, we want our results to be independent of the definition used by BCB. For this reason, we re-run all our experiments after replacing BCB with the OJ dataset, which follows Krinke *et al.*'s strict definition.

The results for the task-specific models are summarized in Table V. We see that, similarly to Table III, CL outperforms the baseline in F1 in 5/6 experiments, while in the other experiment the baseline simply predicts everything to be a clone due to the class imbalance of OJ, leading to higher F1 but lower accuracy. The *one-vs-rest* evaluation does not apply to the OJ dataset because OJ does not contain non-clone pairs for each functionality. We also note that the low precision, recall, and F1 are due to the class imbalance of the OJ dataset

described in Section II. Regarding LLMs, we notice that they perform particularly well on OJ, with F1 ranging from 95% to 99% (full table in our material [73]), regardless of the prompting strategy. This indicates that LLMs perform better under a strict definition of clones, as a result of dealing with less ambiguity. This set of experiments broadens the scope of our experimental conclusions.

VI. Discussion

In this section, we discuss the implications of our findings. With our results, we hope to *steer the focus toward evaluations that test the models on clones of unseen functionality*. Our analysis provides novel evidence that this evaluation is more aligned with real-world scenarios, making it a prominent option for evaluating future models.

Even after the contrastive learning improvements, there is more headroom for improvement in both task-specific models and LLMs. For task-specific models, for example, future work can investigate whether and how adding more functionalities to a dataset can increase the cross-functionality performance. From our experiments, we see that training the task-specific models on 23 functionalities of $\text{BCB}_{s'}$ is not enough. Although BCB can be expanded, the process of doing so is manual, thus raising the question of whether generative AI can be employed to create large datasets, as was recently attempted successfully with GPT [78, 79]. Going a step further, we may need to rethink whether the optimal structure of a semantic clone dataset should be functionality-based like BCB, OJ, and GCJ in the first place. The SCB dataset that was recently proposed is not functionality-based and contains 1000 Java clone pairs, each one implementing a unique functionality. However, the small size of only 1000 clones hinders its use for training task-specific models.

Our qualitative analysis suggests that CL in task-specific models may not be effective in predicting partial clones. Further investigation is required to understand how the definition of clones affects the performance of different architectures. For example, it seems worth investigating whether and how changing the hyperparameters m and τ of Equation (1) and Equation (2) respectively can improve the performance of CL on non-strict clones. Regarding LLMs, we find that while they achieve near-perfect performance on strict clones, they also struggle on partial clones, leaving headroom for improvement through enhanced multi-step prompting or finetuning on partial clone pairs.

Our application of *CL increases cross-functionality performance of existing models in 9/12 experiments*. Although CL has been used for detecting clones before, we are the first to show that it improves the performance of existing models in the unseen functionality setting. Given that it is a model-agnostic technique that can work with any model, future research in clone detection can incorporate this finding. The optimal architecture and hyperparameters for CL can also be the focus of further studies.

Finally, we find that some functionalities are only captured with one code representation. Although the sample is small, it suggests that a single representation of code (text, AST, image) may not be sufficient to universally solve the complex problem of semantic clone detection. This reveals a potentially fertile area of future research toward the development of ensemble models that combine information from text, AST, and image sources of code.

VII. Threats to Validity

Here, we discuss the threats to the validity of our study. **Model and Dataset Selection.** We acknowledge the threat due to the selected models and limit our conclusions to these models only. To minimize this threat, we selected models based on a meaningful distinguishing feature: the input format, selecting a text-based, an AST-based, and an image-based model. We also incorporated LLMs, further broadening our scope. A threat to validity in LLM evaluation is data leakage, or memorization of some training data [80]. The fact that LLMs perform slightly worse than task-specific models in BCB reduces the threat of memorization, but memorization is still possible in the OJ dataset. To mitigate this threat, we also use task-specific models that we train from scratch to reach our conclusions.

Existing datasets for semantic clone detection are a threat to validity [13, 15, 68, 81]. To minimize the functionality imbalance of BCB, which is a threat to validity in other works [15, 68], we introduced a balanced version. It was also shown that BCB_{v2} contains unlabeled clones [68], making the precision potentially inaccurate. To mitigate this, researchers measure precision by manually assessing a small subset of the dataset [82]. We do not follow this process because our goal is not an exact precision score, but rather empirical evidence of (a) weak cross-functionality performance and (b) improved cross-functionality performance with CL. Other measures like accuracy and F1 provide evidence to minimize the threat of a potentially inaccurate calculation of precision. Finally, the granularity of some BCB functionalities is low, like `Copy File` whose implementation is only three lines in Java [68]. Our experiments on OJ, whose functionalities are not trivial, mitigate this threat.

Additional Sources of Distribution Shift. To measure cross-functionality performance of task-specific models, we introduced three evaluation methods. Two of them involve training a model on a dataset and evaluating it on another one. Since the functionalities of the two datasets differ, these methods inherently capture cross-functionality performance. However, there may be additional sources of distribution shift between the two datasets, like different coding styles or application domains, that could lead to lower performance. This poses the threat that the performance drop is not *exclusively* due to weak cross-functionality performance. We argue that based on how the datasets are constructed, coding styles or application domains should not play a significant role: BCB contains code snippets from 25K projects varying in application domain and authorship. Similarly, SCB contains 1K stackoverflow answers on different topics from different authors. This diversity should

prevent a model trained on BCB from overfitting to a specific coding style or domain.

Nevertheless, to further mitigate this threat, we have also evaluated cross-functionality performance within the BCB dataset, in our *one-vs-rest* evaluation method, where we keep one functionality of the BCB dataset in the test set and use the rest of the functionalities for training. The results from all three settings agree; this triangulation increases our confidence that there is weak performance on clones of unseen functionality.

Resampling BCB. We introduced $\text{BCB}_{s'}$ by resampling BCB_{v2} in a functionality-aware manner, to mitigate the functionality imbalance [15, 68]. We used the balanced $\text{BCB}_{s'}$ in our experiments, which introduces the threat that the performance drop reported in RQ1 (Section V-A) may be due to different subsets of the BCB dataset. To mitigate this threat, we conduct three more experiments: we train and evaluate the three models using the evaluation method of the original papers on the $\text{BCB}_{s'}$ dataset. Results show that the performance of all three models on $\text{BCB}_{s'}$ is equally high as in the original papers (F1 is 94.0%, 97.8%, and 92.5% respectively). This gives further confidence that the performance drop we observe in RQ1 (cross-functionality performance compared to the performance reported in the original papers) should not be an effect of the different BCB version but of the weak generalization of the models to clones of unseen functionality.

VIII. Conclusion

We present an in-depth analysis of the cross-functionality performance of clone detection models. We start by highlighting the relevance of detecting clones of unseen functionality to real-world applications. This motivates us to study the evaluation methodology of state-of-the-art models, finding that they were evaluated on the task of detecting clones of seen functionalities. By testing six models on clones of unseen functionality, we show that there is a significant drop in performance. This leaves ample room for future research toward achieving high cross-functionality performance.

Finally, we propose and evaluate the use of contrastive learning to increase the cross-functionality performance of task-specific models by replacing their final classifier with a contrastive classifier. In addition, we introduce an analogous technique for clone detection with generative LLMs, called contrastive in-context learning. We provide evidence that, in most cases, contrastive learning improves cross-functionality performance, making it a prominent option for future research.

Acknowledgments

K. Kitsios, F. Sovrano, and A. Bacchelli gratefully acknowledge the support of the Swiss National Science Foundation through the SNSF Project 200021_197227.

References

[1] R. M. Krawitz, *Code Clone Discovery Based on Functional Behavior*. Nova Southeastern University, 2012.

[2] C. Fang, Z. Liu, Y. Shi, J. Huang, and Q. Shi, "Functional code clone detection with syntax and semantics fusion learning," in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 516–527.

[3] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queen's School of computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[4] C. J. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, pp. 645–692, 2008.

[5] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.

[6] F. Rahman, C. Bird, and P. Devanbu, "Clones: What is that smell?" *Empirical Software Engineering*, vol. 17, pp. 503–530, 2012.

[7] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *2009 IEEE 31st International Conference on Software Engineering (ICSE)*. IEEE, 2009, pp. 485–495.

[8] N. Tsantalis, D. Mazinanian, and G. P. Krishnan, "Assessing the refactorability of software clones," *IEEE Transactions on Software Engineering*, vol. 41, no. 11, pp. 1055–1090, 2015.

[9] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with bigclonebench," in *2015 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2015, pp. 131–140.

[10] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical society*, vol. 74, no. 2, pp. 358–366, 1953.

[11] S. Arshad, S. Abid, and S. Shamail, "Codebert for code clone detection: A replication study," in *2022 IEEE 16th International Workshop on Software Clones (IWSC)*. IEEE, 2022, pp. 39–45.

[12] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code," in *IJCAI*, 2017, pp. 3034–3040.

[13] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 783–794.

[14] A. K. Kaboré, E. T. Barr, J. Klein, and T. F. Bissyandé, "Codegrid: A grid representation of code," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1357–1369.

[15] P. Keller, A. K. Kaboré, L. Plein, J. Klein, Y. Le Traon, and T. F. Bissyandé, "What you see is what it means! semantic representation learning of code based on visualization and transfer learning," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 2, pp. 1–34, 2021.

[16] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 261–271.

[17] J. Zeng, K. Ben, X. Li, and X. Zhang, "Fast code clone detection based on weighted recursive autoencoders," *IEEE Access*, vol. 7, pp. 125 062–125 078, 2019.

[18] F. Siyue, S. Wenqi, W. Yueming, Z. Deqing, L. Yang, and J. Hai, "Machine learning is all you need: A simple token-based approach for effective code clone detection," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE, 2024.

[19] Z. Xu, M. Zhou, X. Zhao, Y. Chen, X. Cheng, and H. Zhang, "xastnn: Improved code representations for industrial practice," *arXiv preprint arXiv:2303.07104*, 2023.

[20] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 476–480.

[21] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a" siamese" time delay neural network," *Advances in neural information processing systems*, vol. 6, 1993.

[22] G. Koch, R. Zemel, R. Salakhutdinov *et al.*, "Siamese neural networks for one-shot image recognition," in *ICML deep learning workshop*, vol. 2, no. 1. Lille, 2015.

[23] J. Mueller and A. Thyagarajan, "Siamese recurrent architectures for learning sentence similarity," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016.

[24] M. Allamanis, E. T. Barr, S. Ducousso, and Z. Gao, "Typilus: Neural type hints," in *Proceedings of the 41st acm sigplan conference on programming language design and implementation*, 2020, pp. 91–105.

[25] X. Wu, H. Li, N. Yoshioka, H. Washizaki, and F. Khomh, "Refining gpt-3 embeddings with a siamese structure for technical post duplicate

detection," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2024, pp. 114–125.
[26] R. Koschke, "Large-scale inter-system clone detection using suffix trees," in *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, 2012, pp. 309–318.
[27] A. S. E. Group, "Ijadataset 2.0," http://secold.org/projects/seclone, January 2013.
[28] J. Svajlenko and C. K. Roy, "Bigcloneeval: A clone detection tool evaluation framework with bigclonebench," in *2016 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2016, pp. 596–600.
[29] F. Al-Omari, C. K. Roy, and T. Chen, "Semanticclonebench: A semantic code clone benchmark using crowd-source knowledge," in *2020 IEEE 14th International Workshop on Software Clones (IWSC)*. IEEE, 2020, pp. 57–63.
[30] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016.
[31] C. Liu, Z. Lin, J.-G. Lou, L. Wen, and D. Zhang, "Can neural clone detection generalize to unseen functionalities," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 617–629.
[32] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.
[33] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021.
[34] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.
[35] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.
[36] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.
[37] S. Karita, N. Chen, T. Hayashi, T. Hori, H. Inaguma, Z. Jiang, M. Someki, N. E. Y. Soplin, R. Yamamoto, X. Wang *et al.*, "A comparative study on transformer vs rnn in speech applications," in *2019 IEEE automatic speech recognition and understanding workshop (ASRU)*. IEEE, 2019, pp. 449–456.
[38] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.
[39] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, pp. 273–297, 1995.
[40] Z. Zhang and T. Saber, "Assessing the code clone detection capability of large language models," in *2024 4th International Conference on Code Quality (ICCQ)*. IEEE, 2024, pp. 75–83.
[41] M. Khajezade, J. J. Wu, F. H. Fard, G. Rodríguez-Pérez, and M. S. Shehata, "Investigating the efficacy of large language models for code clone detection," in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, 2024, pp. 161–165.
[42] P. Jain, A. Jain, T. Zhang, P. Abbeel, J. E. Gonzalez, and I. Stoica, "Contrastive code representation learning," *arXiv preprint arXiv:2007.04973*, 2020.
[43] C. Feng, T. Wang, Y. Yu, Y. Zhang, Y. Zhang, and H. Wang, "Sia-rae: A siamese network based on recursive autoencoder for effective clone detection," in *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2020, pp. 238–246.
[44] M. Zubkov, E. Spirin, E. Bogomolov, and T. Bryksin, "Evaluation of contrastive learning with various code representations for code clone detection," *arXiv preprint arXiv:2206.08726*, 2022.
[45] B. Wan, S. Dong, J. Zhou, and Y. Qian, "Sjbcd: A java code clone detection method based on bytecode using siamese neural network," *Applied Sciences*, vol. 13, no. 17, p. 9580, 2023.
[46] C. Tao, Q. Zhan, X. Hu, and X. Xia, "C4: Contrastive cross-language code clone detection," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 2022, pp. 413–424.
[47] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "UniXcoder: Unified cross-modal pre-training for code representation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, S. Muresan, P. Nakov, and A. Villavicencio, Eds. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 7212–7225. [Online]. Available: https://aclanthology.org/2022.acl-long.499/
[48] S. Liu, B. Wu, X. Xie, G. Meng, and Y. Liu, "Contrabert: Enhancing code pre-trained models via contrastive learning," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2476–2487.
[49] M. Khajezade, F. H. Fard, and M. S. Shehata, "Evaluating few-shot and contrastive learning methods for code clone detection," *Empirical Software Engineering*, vol. 29, no. 6, p. 163, 2024.
[50] M. Gharehyazie, B. Ray, M. Keshani, M. S. Zavosht, A. Heydarnoori, and V. Filkov, "Cross-project code clones in github," *Empirical Software Engineering*, vol. 24, pp. 1538–1573, 2019.
[51] N. Mehrotra, N. Agarwal, P. Gupta, S. Anand, D. Lo, and R. Purandare, "Modeling functional similarity in source code with graph-based siamese networks," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 3771–3789, 2022.
[52] H. Yu, W. Lam, L. Chen, G. Li, T. Xie, and Q. Wang, "Neural detection of semantic code clones via tree-based convolution," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 70–80.
[53] N. Mehrotra, A. Sharma, A. Jindal, and R. Purandare, "Improving cross-language code clone detection via code representation learning and graph neural networks," *IEEE Transactions on Software Engineering*, vol. 49, no. 11, pp. 4846–4868, 2023.
[54] N. Mehrotra and R. Purandare, "Decoding the dna of code: an ai-infused approach to detect code cloning in software systems," Ph.D. dissertation, IIIT-Delhi, 2024.
[55] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE, 1998, pp. 368–377.
[56] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Proceedings of 2nd Working Conference on Reverse Engineering*. IEEE, 1995, pp. 86–95.
[57] C. J. Kapser and M. W. Godfrey, "Supporting the analysis of clones in software systems," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 2, pp. 61–82, 2006.
[58] Mayrand, Leblanc, and Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *1996 Proceedings of International Conference on Software Maintenance*. IEEE, 1996, pp. 244–253.
[59] M. Gabel and Z. Su, "A study of the uniqueness of source code," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, 2010, pp. 147–156.
[60] J. Svajlenko and C. K. Roy, "A survey on the evaluation of clone detection performance and benchmarking," *arXiv preprint arXiv:2006.15682*, 2020.
[61] T. Ishihara, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Inter-project functional clone detection toward building libraries-an empirical study on 13,000 projects," in *2012 19th Working Conference on Reverse Engineering*. IEEE, 2012, pp. 387–391.
[62] B. Laguë, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl, "Assessing the benefits of incorporating function clone detection in a development process," in *1997 Proceedings International Conference on Software Maintenance*. IEEE, 1997, pp. 314–321.
[63] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 712–721.
[64] Y. Dang, D. Zhang, S. Ge, R. Huang, C. Chu, and T. Xie, "Transferring code-clone detection and analysis to practice," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 2017, pp. 53–62.
[65] Y. Dang, D. Zhang, S. Ge, C. Chu, Y. Qiu, and T. Xie, "Xiao: Tuning code clones at hands of engineers in practice," in *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012, pp. 369–378.
[66] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 261–271.
[67] Q. Dong, L. Li, D. Dai, C. Zheng, J. Ma, R. Li, H. Xia, J. Xu, Z. Wu, T. Liu *et al.*, "A survey on in-context learning," *arXiv preprint*

arXiv:2301.00234, 2022.

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

[69] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[70] B. Steenhoek, M. Mahbubur Rahman, M. K. Roy, M. Sanjida Alam, H. Tong, S. Das, E. T. Barr, and W. Le, "To Err is Machine: Vulnerability Detection Challenges LLM Reasoning," *arXiv e-prints*, p. arXiv:2403.17218, Mar. 2024.

[71] N. Mündler, M. Müller, J. He, and M. Vechev, "Swt-bench: Testing and validating real-world bug-fixes with code agents," *Advances in Neural Information Processing Systems*, vol. 37, pp. 81 857–81 887, 2024.

[72] F. Sovrano, A. Bauer, and A. Bacchelli, "Large language models for in-file vulnerability localization can be "lost in the end"," in *Companion Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, ser. FSE 2025. New York, NY, USA: Association for Computing Machinery, 2025.

[73] Anonymous, "Detecting clones of unseen functionalitiy replication package," https://anonymous.4open.science/r/clones-paper-replication-package-FD2B/, 2025.

[74] R. Kamoi, Y. Zhang, N. Zhang, J. Han, and R. Zhang, "When can llms actually correct their own mistakes? a critical survey of self-correction of llms," *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 1417–1440, 2024.

[75] N. Keshtmand, R. Santos-Rodriguez, and J. Lawry, "Understanding the properties and limitations of contrastive learning for out-of-distribution detection," in *International Conference on Pattern Recognition*. Springer, 2022, pp. 330–343.

[76] C. M. Bishop, *Pattern Recognition and Machine Learning*. New York, NY, USA: Springer, 2006.

[77] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *Journal of Machine learning research*, vol. 7, no. Jan, pp. 1–30, 2006.

[78] A. I. Alam, P. R. Roy, F. Al-Omari, C. K. Roy, B. Roy, and K. A. Schneider, "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)*. IEEE, 2023, pp. 1–13.

[79] P. R. Roy, A. I. Alam, F. Al-omari, B. Roy, C. K. Roy, and K. A. Schneider, "Unveiling the potential of large language models in generating semantic and cross-language clones," *arXiv preprint arXiv:2309.06424*, 2023.

[80] J. Sallou, T. Durieux, and A. Panichella, "Breaking the silence: the threats of using llms in software engineering," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, 2024, pp. 102–106.

[81] J. Krinke and C. Ragkhitwetsagul, "How the misuse of a dataset harmed semantic clone detection," *arXiv preprint arXiv:2505.04311*, 2025.

[82] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "Cclearner: A deep learning-based clone detection approach," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 249–260.