Quantifying Memorization Advantage in Code LLMs

Alberick Euraste Djire
SnT / TruX
University of Luxembourg
Luxembourg, Luxembourg
AI4D Excellence Centre (CITADEL)
Ouagadougou, Burkina Faso
euraste.djire@uni.lu

Abdoul Kader Kaboré
University of Luxembourg
Luxembourg, Luxembourg
abdoulkader.kabore@uni.lu

Jordan Samhi
SnT / TruX
University of Luxembourg
Luxembourg, Luxembourg
jordan.samhi@uni.lu

Earl Barr
University College of London
London, United Kingdom
e.barr@ucl.ac.uk

Jacques Klein
SnT / TruX
University of Luxembourg
Luxembourg, Luxembourg
jacques.klein@uni.lu

Tegawendé F. Bissyandé
SnT / TruX
University of Luxembourg
Luxembourg, Luxembourg
tegawende.bissyande@uni.lu

Abstract

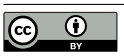
The lack of transparency regarding the code datasets used during LLM training creates substantial challenges in detecting, evaluating, and mitigating data leakage. This paper applies a perturbation-based approach to quantify the "memorization advantage" of LLMs across various coding tasks by measuring the performance gap between a model's handling of data it has likely encountered during training versus novel inputs. Our comprehensive analysis examines 8 open-source code LLMs across 19 benchmark datasets spanning four distinct categories: standard code generation, code understanding, security vulnerability detection, and bug identification. The results reveal significant variations in sensitivity patterns, with models like StarCoder exhibiting substantially higher sensitivity scores (up to 0.8) on certain benchmarks like APPS compared to models like QwenCoder, which maintained consistently lower values (<0.4) across most benchmarks, suggesting fundamental differences in their generalization process and their learned knowledge. Different task categories also showed distinct patterns, with code summarization demonstrating low sensitivity (<0.3) and test generation tasks exhibiting significantly higher values (0.4-0.7, $p < 0.001$).

Interestingly, our analysis of the widely-used CVEFixes and Defects4J benchmarks, frequently suspected of data leakage in the research community, reveals unexpectedly low memorization advantage scores across all models. Defects4J demonstrated significantly lower sensitivity (0.2-0.4, $p < 0.01$) compared to other program repair benchmarks (0.5-0.8), while CVEFixes showed consistently low values below 0.1. These findings challenge prevailing concerns about these datasets' validity for evaluating code LLMs and suggest that models may be effectively generalizing from this data rather than merely memorizing it. Our findings provide critical insights into the generalization capabilities of code LLMs and emphasize the need for more robust evaluation frameworks, particularly in security-related domains where the widest range of sensitivity distributions (from <0.1 to >0.8) indicates variable generalization challenges.

CCS Concepts

• **General and reference** → Empirical studies; **Evaluation**; *Performance*; • **Computing methodologies** → **Natural language generation**.

ACM Reference Format:
Alberick Euraste Djire, Abdoul Kader Kaboré, Jordan Samhi, Earl Barr, Jacques Klein, and Tegawendé F. Bissyandé. 2026. Quantifying Memorization Advantage in Code LLMs. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3764554



ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3764554

1 Introduction

Large Language Models (LLMs) represent a paradigm shift in computational linguistics, fundamentally transforming how systems process and generate human language in various tasks such as text completion or translation. However, the capabilities of these models now extend beyond traditional linguistic boundaries into other fields such as software engineering where the "naturalness" [26] of code constitutes fertile ground for the application of LLMs.

Recent advances in LLM reasoning capabilities have significantly improved their problem-solving ability, allowing them to tackle increasingly complex computational challenges through structured analytical processes [22]. This has directly translated to better performance in code-related tasks, where models now demonstrate advanced abilities in algorithmic design or prediction of logical sequences. Today, a large family of specialized models targeting code analysis and processing have been presented in the literature, achieving state-of-the-art performance in a variety of software-related tasks, including test generation [13, 62], vulnerability detection [10, 48], code summarization [52], and program repair [17, 20, 33] .

However, the promising reported results come with recurrent concerns about their reliability, notably due to suspicions of data leakage [4, 43, 65]. Indeed, because of privacy or commercial considerations, the literature often omits key information on the datasets that were used to train or fine-tune models. Yet, in the absence of

data transparency, it becomes difficult to assess the true generalization capabilities of an LLM for a given task, since the claimed performance improvements might stem from prior exposure to evaluation data during training, leading to inflated results.

Measuring data leakage is very challenging. Many existing approaches require knowledge either about the internal details of the models [8, 15] or some details of part of datasets used during training [9]. More recently, novel ideas have emerged that reason about the observations of how LLMs outputs behave when inputs are perturbed [59]. In this paper, we build on these ideas and implement an approach that evaluates the sensitivity of the LLM to subtle input perturbations in order to measure its **memorization advantage**. This advantage is defined as *the difference in performance between when the model is dealing with 'seen' versus when it is dealing with 'unseen' data*. Formally, the memorization advantage *ma* for a model M can be defined as:

$$ma(M, x, y) = |p_\theta(y|x) - p_\theta(y|x'(x))|$$

Where (x, y) is a sequence from the training data, $x'(x)$ is a function that returns a similar but unseen sequences and $p_\theta(y|x)$ is the model's probability of generating y given x.

This paper. Given persistent debates in the community around the likely inflated performance of LLMs due to potential data leakage, we propose an extensive investigation to shed light into the extent of the problem. Our empirical experiments consider eight (8) popular code LLMs which we apply to twenty (20) different benchmarks across five (5) classical tasks of software engineering: code summarization, vulnerability detection, test generation, program repair, and code synthesis.

We examine the phenomenon (memorization advantage) from two complementary viewpoints: one focusing on identifying potentially contaminated benchmarks, and the other on identifying models with unusual performance patterns on certain data:

- **Benchmark Memorization Advantage**: Given a model m and a set of benchmarks, $B = \{b_0, b_1, ...b_n\}$, we quantify the memorization advantage of a subset $B^* \subset B$ as a statistically significant difference in distribution of input sensitivity scores on this subset compared to the rest, suggesting potential data contamination or memorization of B^* by m.
- **Model Memorization Advantage**: Given a benchmark b and a set of models, $M = \{m_0, m_1, ..., m_n\}$, we quantify the memorization advantage of a model m_i as the statistically significant difference in the distribution of input sensitivity scores of m_i compared to others on b, suggesting that m_i has a systematic advantage in the evaluated data set.

The main contributions of this paper are:
❶Application of prompt perturbation sensitivity analysis to quantify memorization advantage in code LLMs, providing an empirical framework for distinguishing between memorization and generalization effects in model performance
❷Empirical evaluation of 8 state-of-the-art code LLMs across 19 benchmark datasets spanning code generation, test generation, program repair, and vulnerability detection tasks.
❸Statistical analysis results challenging prevailing assumptions about data leakage in widely-used benchmarks (particularly CVEFixes and Defects4J), revealing unexpectedly low memorization advantage scores that suggest genuine generalization rather than memorization
❹Identification of significant variations in sensitivity patterns across models with comparable parameter counts, demonstrating that architectural design choices and training methodologies substantially impact generalization capabilities
❺Task-specific insights showing that code summarization tasks exhibit the strongest generalization across all models, while test generation and context-dependent program repair present the greatest generalization challenges
Artifacts. We make all our artifacts available at:

https://github.com/Berickal/CodeLLM_Memo

2 Background

2.1 Memorization vs. Generalization in Language Models

LLMs function as probabilistic systems that extract and generalize patterns from training data. The fundamental challenge these models face is balancing between pattern generalization and maintaining fidelity to their training data for reliable output generation [9, 24]. This creates an inherent tension in model development and evaluation.

The literature presents multiple definitions for memorization in LLMs. A prominent definition focuses on exact sequence reproduction, where models output verbatim content from their training data [9, 44]. This phenomenon appears across various tasks including completion, question-answering, and logical reasoning [24]. Though memorization is typically associated with overfitting during fine-tuning phases [49, 53, 60], recent research suggests it may actually be a prerequisite for effective generalization, with models showing higher sensitivity to memorized outliers than to common patterns [54].

Generalization, by contrast, describes a model's capability to identify and leverage underlying patterns from training data to generate novel outputs. Formally, for an input x from space $\mathcal{X}$, a generalizing model produces outputs by computing the expected value over its learned distribution:

$$f(x) = \mathbb{E}_{y \sim p(y|x)}[y] \tag{1}$$

where $f(x)$ represents the model's output, and the expected value is calculated over outputs y sampled from the conditional probability distribution $p(y|x)$ that the model has learned.

2.2 Challenges in Detecting Memorization in Code LLMs

Detecting memorization in code LLMs presents unique challenges compared to general-purpose language models:

2.2.1 Data Provenance Opacity. Unlike some general-purpose LLMs with partially documented training sources, code LLMs typically lack transparency regarding training datasets [6]. This opacity complicates determination of which code examples a model has encountered during training.

2.2.2 Code Similarity Evaluation. Code memorization detection is complicated by implementation variability for similar functionality. Standard text similarity metrics may inadequately determine whether a model has memorized specific implementations or genuinely understood programming concepts [2, 46].

2.2.3 Benchmark Contamination. Many standard code benchmarks are publicly available and may be included in various code LLMs' training data [51]. This potential contamination undermines traditional evaluation approaches for assessing true model capabilities.

2.3 Perturbation-Based Approaches for LLM Robustness Evaluation

Perturbation-based approaches offer promising methodologies for evaluating LLMs by examining performance changes when inputs undergo systematic modification. Literature examples include:

(1) **Adversarial Perturbations**: Research has demonstrated how character-level and word-level perturbations reveal vulnerabilities in language models [18, 32], with specific extensions to code models [63].
(2) **Counterfactual Data Augmentation**: Techniques generating counterfactual examples through minimal input perturbations reveal model robustness characteristics [21, 36].
(3) **Program Transformation Analysis**: Studies have explored how systematic code transformations affect model performance, providing insights into code model robustness [24, 47].

For code LLMs, various types of perturbations can be considered: (1) **Syntactic Perturbations**: Modifications preserving program semantics while altering syntax, such as variable renaming or code reformatting [31, 47]. (2) **Semantic Perturbations**: Changes slightly altering program behavior, like modifying constant values or changing loop boundaries [23]. And (3) **Structural Perturbations**: Alterations to code organization while preserving functionality, including function reordering or module restructuring [55].

2.4 The Perturbation Sensitivity Hypothesis on Memorization

Prior work has demonstrated that neural network generalization capabilities can be evaluated through sensitivity to data perturbations [14, 61]. When presented with text samples for completion tasks, models may produce correct outputs regardless of whether they memorized or interpolated. However, introducing perturbations reveals distinctive behavioral patterns. With memorized content, even minor perturbations (e.g., random bit flips at 2% rate) cause abrupt performance degradation, while interpolated understanding shows more gradual performance decline as perturbation increases as illustrated in Figure 1.

This Perturbation Sensitivity Hypothesis (PSH) offers a methodological framework to distinguish between memorization and generalization in LLMs. According to PSH, models that have memorized specific content demonstrate high sensitivity to input perturbations, while models effectively interpolating show greater robustness when inputs are perturbed [14, 19].

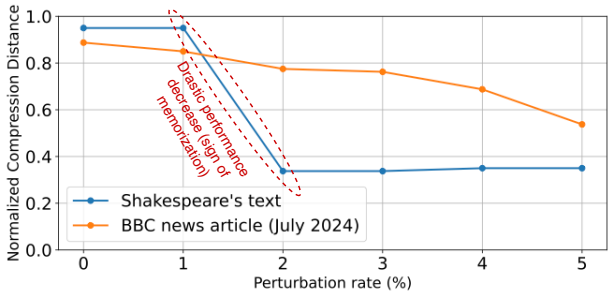


Figure 1: GPT_4o text completion *performance falloff* for a memorized Shakespeare poem submitted to perturbations vs gradual performance decline with a recent BBC text (could not part of the training set of GPT_4o).

2.5 Memorization Advantage in LLM Evaluation

In this work, our definition of *memorization advantage* extends the perturbation sensitivity concept by quantifying performance disparities between a model's handling of familiar versus novel inputs. This metric is particularly valuable for evaluating LLMs when data leakage might significantly impact assessment results.

2.5.1 Definition and Measurement. Memorization advantage directly quantifies a model's sensitivity to input perturbations, reflecting the differential robustness between training and non-training data.

Given a task T, an input X and the set of generated perturbed inputs $X^* = \{x_0^*, x_1^*, ...x_n^*\}$, we prompt the LLM and collect a set of output sets $Y^* = \{Y_0^*, Y_1^*, ...Y_n^*\}$ with $Y_k^* = \{y_{(k,1)}^*, y_{(k,2)}^*, \dots y_{(k,i)}^*\}$ being the set of outputs associated with the perturbed input x_k^*. Indeed, for each input, we prompt the LLM i times to obtain i sample outputs. This is done in order to help establish statistical significance of results as single runs might be outliers and not representative of true model capabilities.

For a given output set Y_k^* (yielded by an LLM prompted with perturbed input x_k^*) and a reference output Y, we compute a metric on the performance variation as follows:

$$m(Y_k^*) = \frac{1}{i}\sum_{j=1}^{i} distance(y_{(k,j)}^*, Y) \tag{2}$$

where *distance* implements a task-dependent edit distance function.

To quantify the model's sensitivity to the perturbations on input X, we compute the maximum performance falloff when the model is subjected to increasingly intense perturbations:

$$sensitivity(X) = \max_{j\in 1,..,k-1}(m(Y_j^*) - m(Y_{j+1}^*)) \tag{3}$$

2.5.2 Significance in Code LLMs. For code-generating LLMs, memorization advantage carries particular importance due to several factors:

(1) **Benchmark Contamination**: Popular code benchmarks may have been incorporated into training data, artificially inflating performance metrics [35, 64].
(2) **Security Implications**: In security-critical applications like vulnerability detection, memorizing known vulnerabilities without genuine understanding could create false confidence in model capabilities [45].

(3) **Generalization Assessment**: Differentiating between memorization and true generalization is essential for understanding a model's practical utility in software development contexts [3, 11].

3 Experimental Setup

This section presents our experimental framework for quantifying memorization advantage in code LLMs. We first describe our methodology for measuring perturbation sensitivity (Section 3.1), followed by details on the evaluated models (Section 3.2), the tasks and datasets used in our evaluation (Section 3.3), and our perturbation techniques (Section 3.4). Together, these components form a robust evaluation pipeline that enables systematic analysis of memorization versus generalization across diverse code tasks.

3.1 Methodology

Our methodology is grounded in the Perturbation Sensitivity Hypothesis (PSH), which posits that models demonstrate higher sensitivity to perturbations on memorized inputs compared to those where true generalization has occurred.

The core intuition driving our approach is that when a model has memorized specific code snippets or patterns, its performance will degrade significantly when those inputs undergo even minor perturbations. Conversely, when a model has successfully learned the underlying patterns and principles (interpolation), it will maintain more consistent performance despite input variations.

Figure 2 illustrates this principle with a comparison between StarCoder2 and CodeLlama on a code completion task on an example selected from the MBPP benchmark. The yielded outputs demonstrate how StarCoder2 exhibits higher sensitivity to "prompt" perturbations compared to CodeLlama, suggesting higher memorization advantage on this sample.

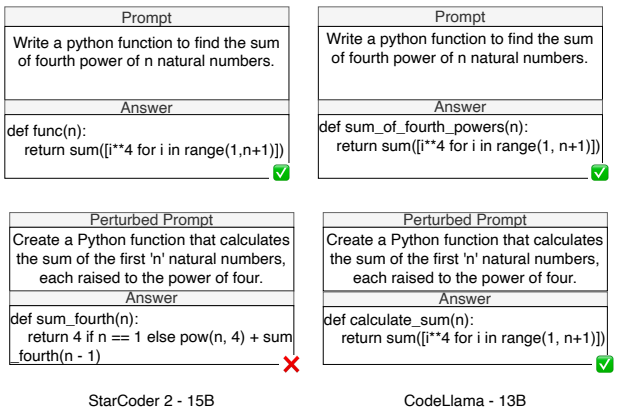


Figure 2: Memorization advantage example: impact of prompt perturbation on the outputs of StarCoder and CodeLlama

Algorithm 1 formalizes our perturbation sensitivity evaluation procedure. For each input, ❶ we apply progressively stronger (but still small[1]) perturbations, ❷ measure the model's performance on each perturbed variant, and ❸ calculate the maximum performance drop between consecutive perturbation levels as our sensitivity metric.

Input: Model M, Prompt template Prt, Input X, Perturbation function σ
Data: Perturbation rate max $pr_{max} = 5$, Answers per prompt $ans_{max} = 3$, Evaluation function $Perf$
Output: Maximum perturbation sensitivity score

```
X* ← [ ] ;                                   // Array of perturbed inputs
Y* ← [ ] ;                                   // Array of model responses
m ← [ ] ;                                    // Performance differences
// Generate perturbed inputs at increasing perturbation rates
for k = 0 to pr_max do
    X*.append(σ(X, k));
end
// Collect LLM responses for each perturbed input
for k = 0 to pr_max do
    prompt ← Prt(X*[k]);
    answers ← [ ];
    for i = 0 to ans_max do
        answers.append(M(prompt));
    end
    Y*.append(answers);
end
// Calculate performance differences between consecutive perturbation levels
for k = 0 to pr_max − 1 do
    eval ← mean(Perf(Y*_k)) − mean(Perf(Y*_{k+1}));
    m.append(eval);
end
return max(m)
```

Algorithm 1: Input Perturbation Sensitivity Evaluation

Using this approach, we compute perturbation sensitivity scores for each benchmark and model combination. We then analyze the distribution of these scores to quantify memorization advantage at the benchmark level: a low sensitivity on a given benchmark suggests that the model has developed robust generalization capabilities in the neighborhood of the benchmark instances, maintaining consistent performance despite input variations.

Conversely, benchmarks exhibiting statistically significant higher sensitivity distributions indicate potential limitations in the model's generalization capabilities. This high sensitivity can be attributed to either:

(1) **Memorization effects**: The model performs well only on exact or near-exact matches to training examples but fails to maintain performance under perturbations.
(2) **Knowledge gaps**: The model lacks sufficient exposure to the underlying patterns or principles necessary to solve the perturbed variants of these problems.

To identify statistically significant differences in sensitivity distributions, we employ the Mann-Whitney U test with Bonferroni correction for multiple comparisons, considering a significance level of $\alpha = 0.05$. This non-parametric approach avoids assumptions about the underlying distribution of sensitivity scores. Moreover, we repeat the process 03 times while considering the average of the sensitivity for each sample considered to ensure the coherence of the results. Also, in order to mitigate the LLMs randomness, we fix the temperature at 0.3 and the top_k at 0.5.

[1]Here, by small, we mean perturbations that would not change a human's response to the perturbed query. Like adversarial examples, our perturbations are constrained to be those a human would ignore or fail to notice.

3.2 Models Considered

We selected a diverse set of state-of-the-art code LLMs representing different model families, architectures, and training methodologies. This diversity enables us to investigate whether memorization advantage patterns vary systematically across different model development approaches. The models included in our analysis are:

- **DeepSeek-Coder-V2 (16B)** is an open-source family of code language models built on the DeepSeek-V2 foundation. It was pretrained on a diverse dataset comprising 60% source code, 10% mathematical corpus, and 30% natural language from CommonCrawl and GitHub repositories. This balanced composition optimizes the model for a wide range of code-related tasks while maintaining strong natural language understanding capabilities [16].
- **Qwen2.5-Coder (14B)** is a code-specialized model series built on the Qwen2.5 architecture, trained on multilingual code repositories. It implements an innovative hybrid attention mechanism specifically designed to handle both local and global code dependencies, enabling efficient processing of complex codebases with nested structures and long-range relationships [29].
- **StarCoder (15B):** is developed through an open-scientific collaboration, this open-source code model was trained on The Stack V2 corpus. The training data includes permissively licensed code, unlicensed files, and specialized datasets like APPS, CodeContest, and GSM8K to enhance mathematical reasoning and algorithmic problem-solving capabilities [41].
- **CodeLlama (13B)** is a Meta's code-specialized model built on LLama 2, developed through a cascade of training and fine-tuning steps. This progressive approach gradually enhanced the base model's capabilities on code-related tasks while maintaining its foundation in natural language understanding, creating a versatile model for various software engineering applications [50].
- **Codestral (22B):** is an open-weight generative AI model explicitly designed for code generation tasks and trained on a diverse dataset of 80+ programming languages. Hovewer, this model is only available in the size 22B [1].
- **OpenCoder (8B)** is an open and reproducible code LLM family available in two sizes (1.5B and 8B). It was pretrained using the RefineCode corpus, a reproducible dataset of 960 billion tokens across 607 programming languages, incorporating over 130 language-specific rules with customized weight assignments [28].
- **WizardCoder (33B)** is a StarCoder variant that applies the Evol-Instruct method to evolve Code Alpaca data generated through self-instruction. The pretrained StarCoder model is then fine-tuned with this evolved data, creating a model with enhanced instruction-following capabilities for code generation tasks [42].
- **Magicoder (7B)** uses the novel OSS-Instruct methodology that leverages open-source code to create contextually relevant instruction-response pairs. This approach enables code generation that adheres to real-world best practices and community standards. The model was developed using CodeLLama-Python-7B as its base architecture [57].

3.3 Benchmarks: Tasks and Datasets

We evaluate memorization advantage across three categories involving five distinct code-related tasks, using a total of 19 benchmark datasets. These tasks represent critical dimensions of software engineering that have emerged as essential benchmarks for assessing the capabilities of code LLMs, covering the full spectrum from generation to comprehension to repair. Considering these dimensions allows us to investigate whether patterns of memorization advantage vary across different types of code understanding and generation challenges. The task categories are **NL2Code Tasks** (Code generation), **Code2Code Tasks** (Test generation & Program repair) and **Code2NL Tasks** (Vulnerability detection & Code summarization).

3.3.1 Code Generation. Our evaluation of code generation capabilities includes six diverse benchmarks. **HumanEval** [11] contains 164 hand-written programming problems with test cases, focusing on function completion tasks in Python that emphasize reasoning and algorithmic skills rather than API knowledge. **APPS** [25], the Assessment of Programming Problems for Students, features 10,000 problems at varying difficulty levels with input/output examples and function signatures. **MBPP** [3], the Mostly Basic Python Programming dataset, comprises 974 Python programming tasks of simple to moderate difficulty, designed for evaluating code generation capabilities. **LBPP** [43],a collection of 161 Python programs with corresponding unit tests, specifically created to address data contamination concerns and constructed to be "fresh" (not leaked at it time of releasing in July 2024). **CodeContest** [38] provides competition-level programming problems from platforms like Codeforces, including 13,610 problems with test cases across multiple languages. Finally, **XLCost** [67], the Cross-Lingual Code Intelligence Evaluation Suite, focuses on multilingual code generation tasks across various programming languages.

3.3.2 Program Repair. For program repair, we consider three established benchmarks. **QuixBugs** [39] is a dataset containing 40 programs derived from the Quixey Challenge, implemented in both Python and Java. Each program features a precisely identified one-line defect, accompanied by passing (when possible) and failing test cases. **Defects4J** [34] is a comprehensive collection of 854 reproducible bugs extracted from real-world Java projects. Each bug is isolated with a test case that triggers the failure, making it one of the most widely used benchmarks for evaluating automated program repair techniques in industrial-scale software. **ConDefects** [58] is an extensive benchmark comprising 1,254 faulty Java programs and 1,625 faulty Python programs, each paired with precise fault location information and corresponding repaired versions. This dataset is specifically designed to evaluate contextual program repair capabilities in a multi-language setting, offering realistic defect scenarios with ground truth fixes.

3.3.3 Test Generation. For test generation, we rely on two specialized benchmarks. **BigCodeBench** [68] is a benchmark designed to evaluate LLMs with practical and challenging programming tasks that reflect real-world development scenarios. The benchmark consists of carefully curated programming problem pairs, each including an instructional prompt, canonical solution, and comprehensive unit tests, enabling rigorous assessment of models' practical coding capabilities. **TestEval** [56] provides a collection of 210 Python programs sourced from online programming platforms, specifically designed to evaluate LLMs' capabilities in test

case generation rather than code implementation. This benchmark assesses a model's ability to understand existing code functionality and generate appropriate test cases that verify correctness, focusing on a critical but often overlooked aspect of the software development lifecycle. We also added for this task the benchmark QuixBug which proposed for each program a unit test case associated.

3.3.4 Vulnerability Detection. For vulnerability detection, we include six diverse benchmarks. **CVEFixes** [5] provides a collection of code patches that address Common Vulnerabilities and Exposures (CVEs), paired with their vulnerable counterparts. **VulDetectBench** [40] offers a benchmark suite for evaluating vulnerability detection systems across multiple programming languages, with annotated vulnerable code regions. **VulnPatchPairs** [48] contains paired datasets of vulnerable code snippets and their corresponding security patches from real-world software. **Devign** [66] features a collection of 40 CVEs collected from 4 popular C libraries, with function-level vulnerability annotations. **ReVeal** [10] includes a collection of past vulnerabilities from two open-source projects (Linux Debian Kernel and Chromium) and their associated patches. **DiverseVul** [12] provides an extensive collection of 18,945 vulnerable functions across 150 Common Weakness Enumerations (CWEs) and 330,492 non-vulnerable functions, extracted from 295 projects.

3.3.5 Code Summarization. For code summarization, we utilize **CodeSearchNet** [30], a large-scale collection containing over 2 million code-docstring pairs across six programming languages (Python, Java, JavaScript, PHP, Go, and Ruby), designed for code search and summarization tasks. This benchmark enables evaluation of models' ability to generate natural language descriptions from code snippets across diverse programming paradigms and syntax. In addition, we also consider a collection of 69,708 pairs of ⟨API sequence, code, summary⟩ collected from Java projects published on Github from 2015 to 2016 and used to train the model TLCode-Sum [27].

3.4 Perturbation Methods

We employ task-specific perturbation methods that introduce surface-level variations:

3.4.1 Natural Language Perturbations. For **NL2Code tasks**, notably code generation, with natural language prompts, we generate five progressively altered versions of each input using the BART [37] model for controlled paraphrasing. These perturbed prompts (i.e., rephrased) are ordered by their cosine distance from the original prompt, ensuring a gradual *syntactic* drift while maintaining the core requirements and functionality described in the prompt. Figure 3 illustrates how varying intensity applied to an MBPP sample affect StarCoder2's performance in code generation. Interestingly, the outputs generated from both the original input and the low-perturbed input successfully passed all associated tests. However, the execution of the output produced from the high perturbed input fails (*TypeError* on argument of *isinstance*).

3.4.2 Code Perturbation. For **Code2Code or Code2NL tasks**, which take code as input, we progressively rename variables using randomly generated identifier names[2]. The perturbation intensity

[2]We leave semantic perturbations to future work.

Figure 3: Examples Outputs by StarCoder2 when applying perturbations to prompt instructions from MBPP)

increases with the number of variables renamed, creating a controlled gradient of syntactic variation. Figure 4 illustrates the perturbations applied to a QuixBugs code sample at different intensity levels and the corresponding code outputs from the StarCoder2 model prompted for program repair task. We observe that the repaired code produced by the LLM for the original input and the low-perturbed input are correct (w.r.t test suite), while the ouput generated for the high-altered, but semantically equivalent, buggy versions is found to be incorrect (w.r.t test suite).

For both perturbation types, we maintain five distinct perturbation levels ($pr_{max} = 5$), enabling fine-grained analysis of performance degradation patterns. The perturbation functions are designed to be deterministic and reproducible, facilitating comparison across different models and experiments.

4 Experimental Results

This section presents our empirical findings on memorization advantage across different code-related tasks and models. For each task category, we analyze the perturbation sensitivity distributions and highlight statistically significant patterns. Lower sensitivity values indicate more robust generalization (interpolation), while higher values suggest potential memorization or knowledge gaps.

4.1 Code Generation

Figure 5 illustrates the sensitivity distributions across code generation benchmarks. The majority of models exhibit relatively low sensitivity values (< 0.4) across most code generation benchmarks, suggesting robust interpolation capabilities. QwenCoder and CodeLlama consistently demonstrate the lowest sensitivity scores, indicating superior generalization capabilities for fundamental coding tasks.

Among the models evaluated, StarCoder exhibits a markedly different pattern. It demonstrates significantly elevated sensitivity on the APPS benchmark (~ 0.8, $p < 0.01$ using Mann-Whitney U

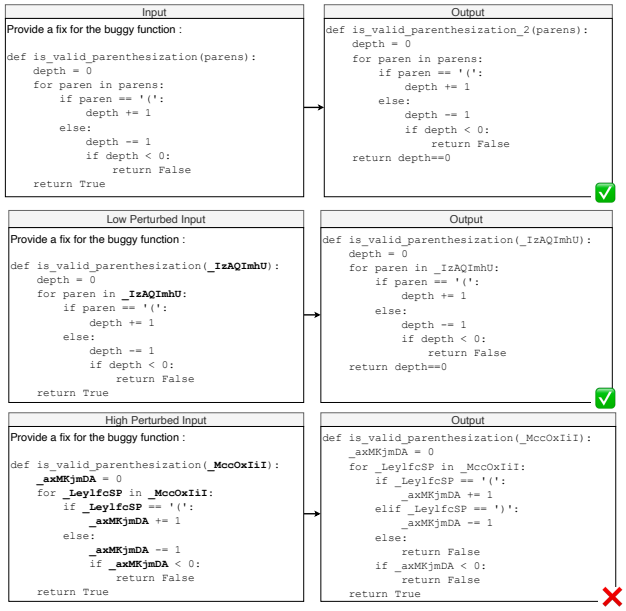


Figure 4: Example Outputs by StarCoder when apply perturbations to input prompt from QuixBugs - see supplementary file for more details about test failures

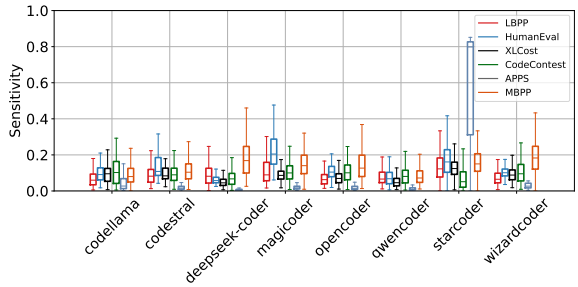


Figure 5: Perturbation sensitivity distributions across code generation benchmarks for all evaluated models. *Lower values indicate stronger generalization capabilities, while higher values suggest potential memorization or brittle understanding.*

test with Bonferroni correction), substantially deviating from both its performance on other benchmarks and from other models' performance on APPS. This pronounced elevation suggests potential memorization behavior specifically for this benchmark, which may indicate training data contamination.

When comparing across benchmarks, we find that the MBPP benchmark consistently elicits elevated sensitivity values across the model spectrum compared to other code generation tasks. This suggests that MBPP's problems may require more specific knowledge patterns that are less amenable to generalization, despite their ostensibly basic nature.

Key Insights ☛ Code Generation

- *Most models demonstrate strong generalization capabilities for basic code generation tasks*
- *StarCoder shows significant evidence of potential memorization specifically on the APPS benchmark*
- *MBPP presents unique generalization challenges across all evaluated models, suggesting its problems may require more specific knowledge patterns*
- *QwenCoder and CodeLlama exhibit the most robust generalization profiles for code generation*

4.2 Test Generation

Figure 6 presents sensitivity distributions for test generation benchmarks. Test generation requires deeper code comprehension capabilities than basic code generation. All evaluated models demonstrate substantially higher sensitivity values for test generation tasks (0.4-0.7 median range) compared to basic code generation tasks (0.2-0.4 median range). This statistically significant difference ($p < 0.001$, paired t-test) suggests that tasks requiring deeper code understanding are more susceptible to perturbation effects, reflecting the increased complexity of generating tests versus implementing functionality.

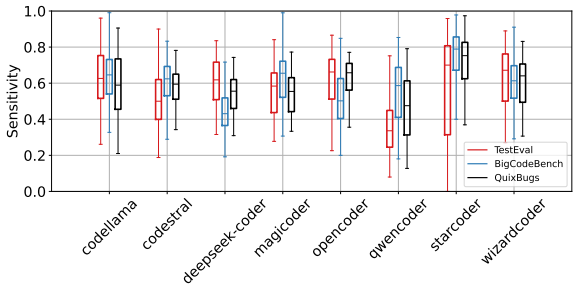


Figure 6: Perturbation sensitivity distributions across test generation benchmarks. *Higher values indicate greater sensitivity to input perturbations, suggesting potential challenges in generalizing test generation capabilities.*

The consistency across benchmarks reveals that TestEval, QuixBugs, and BigCodeBench exhibit remarkably similar sensitivity patterns across models (Pearson correlation $r > 0.85$ between benchmark sensitivities). This high correlation suggests these benchmarks evaluate fundamental test generation capabilities that transcend specific model architectures or training methodologies, revealing inherent generalization boundaries for this task. StarCoder consistently registers sensitivity values at the upper bound of the distribution, reinforcing its tendency toward higher sensitivity observed in other task categories.

Key Insights ☛ Test Generation

- *Test generation tasks consistently show higher perturbation sensitivity than code generation across all models, which suggests that test generation represents a more challenging generalization task for current code LLMs*
- *All three test generation benchmarks exhibit similar sensitivity patterns, suggesting similar fundamental test targets/patterns*

4.3 Program Repair

Figure 7 presents sensitivity distributions for program repair benchmarks. Defects4J demonstrates notably lower sensitivity (0.2-0.4) compared to other program repair benchmarks (0.5-0.8), representing a statistically significant difference ($p < 0.01$, Mann-Whitney U test). This unexpected robustness challenges prevailing concerns about data leakage in this widely-used benchmark, suggesting models may have developed genuine generalization capabilities for the types of bugs represented in Defects4J rather than merely memorizing specific instances.

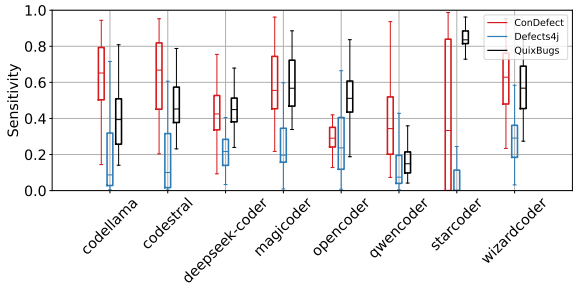


Figure 7: Perturbation sensitivity distributions for program repair benchmarks. *The varying sensitivity patterns across benchmarks suggest different levels of generalization challenges for bug detection and repair tasks.*

Benchmark-specific patterns reveal varying challenges. ConDefect consistently elicits the highest sensitivity values across most models (median ~ 0.7), indicating that repair tasks requiring broader contextual understanding present particular generalization challenges. This aligns with the benchmark's design focus on bugs that span multiple functions or require module-level comprehension, suggesting current models' generalization capabilities may be more limited for context-dependent repairs.

StarCoder exhibits exceptionally high sensitivity on bug detection tasks, particularly evident with the ConDefect benchmark, while QwenCoder lacks measurable results for certain benchmarks in this category, potentially indicating limitations in its training data coverage for program repair tasks.

Key Insights ☛ Program Repair

- *Defects4J shows surprisingly low sensitivity, challenging literature concerns about data leakage in this widely-used benchmark*
- *Context-dependent bugs (ConDefect) present the greatest generalization challenge across all models*
- *The variability in sensitivity across repair benchmarks indicates differing levels of generalization difficulty based on bug context and complexity*
- *Models demonstrate varying generalization capabilities for different types of bugs, suggesting specialized knowledge*

4.4 Vulnerability Detection

Figure 8 illustrates sensitivity distributions across security-oriented benchmarks. The security domain presents the widest range of sensitivity distributions among all task categories (ranging from < 0.1 to > 0.8), suggesting pronounced variability in generalization capabilities for security-related tasks. This variability likely reflects the diverse nature of security vulnerabilities and the different approaches needed to identify them.

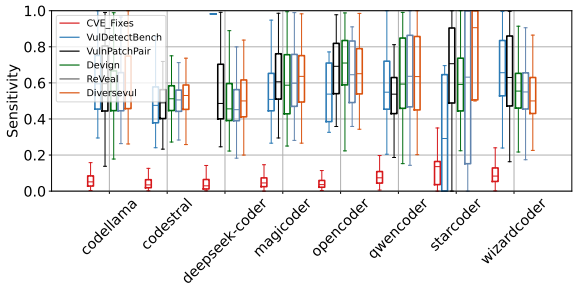


Figure 8: Perturbation sensitivity distributions across vulnerability detection benchmarks. *Note the exceptionally low sensitivity for CVEFixes across all models, contrasting with moderate to high sensitivity for other security benchmarks.*

Particularly noteworthy is the exceptional robustness observed in one benchmark: CVEFixes demonstrates exceptionally low sensitivity (consistently below 0.1) across all evaluated models, representing a statistically significant deviation ($p < 0.001$, Kruskal-Wallis test) from other security benchmarks. This unexpected robustness challenges previous assumptions about potential data leakage in this benchmark and suggests models have developed strong, generalized understanding of common vulnerability patterns rather than memorizing specific CVE instances.

The remaining security benchmarks (VulDetectBench, VulnPatchPair, Devign, ReVeal, and DiverseVul) present similar sensitivity distributions with moderate to high values (0.4-0.8) across models. This consistency suggests these benchmarks evaluate similar underlying capabilities related to vulnerability identification, with the higher sensitivity values indicating ongoing challenges in generalizing security knowledge.

Key Insights ☛ Vulnerabilty Detection

- *Security benchmarks show the widest variance in sensitivity among all task categories*
- *CVEFixes exhibits remarkably low sensitivity across all models, challenging assumptions about data leakage*
- *Most security benchmarks show moderate to high sensitivity, indicating generalization challenges for security tasks*
- *The exceptional performance on CVEFixes suggests models may have successfully generalized common vulnerability patterns, rather than memorizing specific instances*

4.5 Code Summarization

Figure 9 illustrates the sensitivity distribution in code summarization benchmarks. All evaluated models demonstrate remarkably low sensitivity values for code summarization tasks (median values < 0.3), suggesting robust interpolation capabilities across the model spectrum. This indicates that models have developed generalizable understanding of the relationship between code structure and natural language descriptions, making this task less susceptible to perturbation effects than other code-related tasks.

The consistently low sensitivity values observed across different model architectures suggest that code summarization may be a task where current code LLMs have developed particularly strong

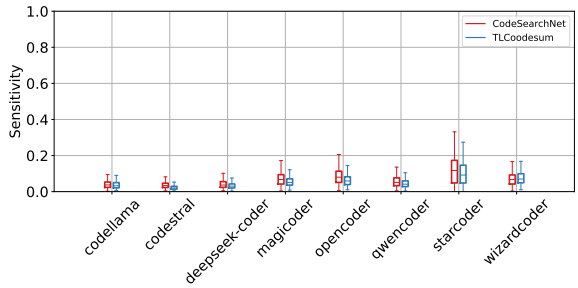


Figure 9: Perturbation sensitivity distributions for code summarization benchmarks. *The consistently low sensitivity values across all models suggest robust generalization capabilities for this task category.*

generalization capabilities, possibly due to the abundance of code-documentation pairs in pre-training datasets and the more standardized nature of the task compared to open-ended code generation or complex program repair.

Key Insights ☛ Code Summarization

- *All models demonstrate strong generalization capabilities for code summarization tasks*
- *The consistently low sensitivity across different model architectures suggests code summarization may be a task where current LLMs excel*
- *Potential explanations include abundant code-documentation pairs in pre-training data and the relatively standardized nature of the task*
- *Code summarization represents the task with the most robust generalization among all evaluated categories*

5 Discussion

Our analysis of memorization advantage across code LLMs reveals important insights about model generalization capabilities, benchmark reliability, and evaluation methodologies. In this section, we discuss the broader implications of our findings, their limitations, and directions for future research.

5.1 Key Findings and Their Implications

Our perturbation sensitivity analysis revealed several notable patterns with important implications for code LLM development and evaluation:

5.1.1 Task-Dependent Generalization Capabilities. Our results show that code LLMs exhibit varying degrees of generalization capabilities across different tasks. This task-dependent pattern suggests that different capabilities may require different evaluation approaches. For tasks where models have developed strong generalization (e.g., code summarization), standard benchmarks may provide reliable assessments. However, for tasks with higher sensitivity (e.g., test generation), researchers should employ more rigorous evaluation methods that account for potential memorization effects.

5.1.2 Benchmark Contamination vs. Genuine Generalization. One of our most surprising findings was the exceptionally low sensitivity observed for certain benchmarks that have been previously suspected of data contamination, particularly CVEFixes and Defects4J. This finding challenges the common assumption that strong performance on widely available benchmarks necessarily indicates memorization or data leakage. It suggests that some benchmark patterns may be easier to generalize, perhaps due to their similarity to common programming patterns or because they represent fundamental concepts that are well represented in diverse training data. It is possible that very high duplication of a particular datum may train models to tolerate more noise around that datum, achieving a sort of locally constrained generalization to the immediate neighborhood of a memorized datum, while still not achieving actual generalization, but instead just pushing the task performance sensitivity farther out from the datum. This may be what we are seeing with CVEFixes and Defects4J.

5.1.3 Model-Specific Patterns. The significant variations in sensitivity patterns across models with comparable parameter counts (e.g., StarCoder vs. CodeLlama) indicate that architectural design choices and training methodologies substantially impact generalization capabilities. This finding has important implications for model development, confirming recent findings (cf. DeepSeek) that simply scaling up model size or training on more data may be less effective than thoughtful architectural design and training methodology. The consistent generalization advantages of instruction-tuned models may suggest that alignment techniques may improve not only safety but also fundamental generalization capabilities.

5.2 Mitigating Memorization Advantage in Evaluation

Our findings highlight the need for more robust evaluation frameworks that account for potential memorization advantage effects. We propose several approaches to mitigate these issues:

5.2.1 Dynamic Benchmarks. Static benchmarks present inherent risks of contamination as they become widely used and potentially included in training datasets. Dynamic benchmarks that are regularly updated with new examples offer a promising alternative. By continuously refreshing benchmark content, the probability of data leakage is minimized, and evaluations better reflect true model capabilities rather than memorization effects.

Implementation of dynamic benchmarks could follow several strategies:

- **Time-stamped versioning**: Creating benchmark versions with clear temporal boundaries allows researchers to select versions predating a model's training cutoff.
- **Procedural generation**: For certain tasks, programmatically generating new benchmark examples with controlled complexity and characteristics can ensure evaluation on unseen examples.
- **Human-in-the-loop curation**: Incorporating expert input to continuously develop new benchmark examples that target specific capabilities while avoiding patterns from existing benchmarks.

5.2.2 Perturbation-Based Evaluation. Our research demonstrates the value of perturbation-based evaluation approaches in assessing

memorization advantage. We recommend incorporating perturbation sensitivity analysis as a standard component of code LLM evaluation frameworks. This approach provides deeper insights than performance metrics alone and can help identify potential memorization effects.

Specifically, we recommend:
①Reporting perturbation sensitivity alongside standard performance metrics.
②Employing diverse perturbation types that preserve semantic meaning.
③Setting benchmark-specific thresholds for acceptable sensitivity levels.
④Comparing sensitivity distributions across models on the same benchmarks.

5.2.3 Cross-Domain Evaluation. Our results revealed that models often exhibit different sensitivity patterns across task domains. This suggests that comprehensive evaluation should include diverse tasks spanning multiple domains to provide a more complete picture of a model's generalization capabilities.

We recommend evaluation frameworks that systematically assess models across the full spectrum of code-related tasks, from generation to comprehension to repair. This approach helps identify domain-specific strengths and weaknesses while providing a more nuanced understanding of overall generalization capabilities.

5.3 Theoretical Implications

Our findings contribute to the broader theoretical understanding of memorization and generalization in large language models:

5.3.1 Memorization as a Spectrum. The varied sensitivity patterns observed across tasks and benchmarks suggest that memorization versus generalization is not a binary distinction but rather a spectrum. Models may partially memorize certain patterns while developing genuine generalization capabilities for others, and the boundary between these processes is often blurry.

This perspective aligns with recent theoretical work suggesting that memorization may be a necessary precursor to generalization [9, 19]. Our results provide empirical support for this view in the context of code LLMs, showing that even models with strong generalization capabilities exhibit varying degrees of sensitivity across different tasks and benchmarks.

5.3.2 Task-Specific Generalization Boundaries. The consistent sensitivity patterns observed within task categories suggest that different tasks may have inherent generalization boundaries—thresholds beyond which current model architectures struggle to develop robust, generalizable capabilities. These boundaries appear to correlate with task complexity and the degree to which tasks require deep semantic understanding versus pattern recognition.

This finding has implications for model architecture design, suggesting that specialized architectures or training methodologies may be necessary to overcome generalization boundaries for particularly challenging tasks. It also suggests that evaluation frameworks should account for these task-specific boundaries when interpreting performance metrics.

5.4 Limitations and Threats to Validity

While our study provides valuable insights into memorization advantage in code LLMs, several limitations and threats to validity should be acknowledged:

Unknown Training Data. A fundamental limitation of our study is the lack of complete knowledge about the specific datasets used to train the evaluated models. Without definitive information about training data composition, we cannot make absolute claims about data leakage. Our perturbation sensitivity analysis provides strong evidence but cannot definitively prove memorization versus generalization in all cases.

Perturbation Method Limitations. Our perturbation methods focus on specific types of input variations that preserve semantic meaning. While these methods are effective for detecting certain types of memorization, they may not capture all possible forms of memorization or generalization. Different perturbation types might reveal different sensitivity patterns, and certain models might be more robust to particular perturbation types. We use a single code perturbation: alpha-renaming of identifiers to nonces. Cao et al. showed that LLMs can tolerate substantial noise in their inputs [7]: it may be that our code perturbation does not, for some memorized instances, push the model beyond its noise tolerance to expose a sudden performance drop.

Benchmark Selection. While we included a diverse set of benchmarks spanning multiple task categories, our selection is not exhaustive. Different benchmarks within the same task category might reveal different sensitivity patterns. Additionally, newly emerging tasks and benchmarks might present different generalization challenges not captured in our analysis.

Model Version Specificity. Our findings are specific to the particular versions of the models evaluated. Newer versions of these models, or models with different parameter counts or training methodologies, might exhibit different sensitivity patterns. The field of code LLMs is rapidly evolving, and our results represent a snapshot of current capabilities.

5.5 Future Research Directions

Based on our findings and limitations, we identify several promising directions for future research:

Architectural Innovations for Task-Specific Generalization. Our results highlight specific tasks and benchmarks where current models struggle to develop robust generalization capabilities. Future research could focus on developing specialized architectural components or training methodologies targeting these challenging domains, particularly test generation and context-dependent program repair.

Expanded Perturbation Methodologies. Developing more diverse and sophisticated perturbation methodologies would provide deeper insights into model generalization capabilities. Future work could explore other perturbations, including those that perturb semantics.

Longitudinal Studies of Model Evolution. Tracking changes in perturbation sensitivity across different versions of the same model family would provide valuable insights into how generalization capabilities evolve with advances in architecture, training data, and alignment techniques. Such longitudinal studies could help identify the innovations that most effectively improve generalization.

Human-Model Comparative Studies. Comparing the perturbation sensitivity patterns of code LLMs with those of human programmers would provide interesting insights into the differences between machine and human generalization. Such studies could help identify areas where models have developed human-like robustness versus areas where they rely on more brittle pattern matching.

6 Conclusion

Our large-scale investigation into memorization advantage in code LLMs provides important insights for both researchers and practitioners. By quantifying perturbation sensitivity across diverse tasks and benchmarks, we have revealed task-specific, model-specific, and benchmark-specific patterns that challenge existing assumptions about memorization and generalization.

These findings highlight the need for more nuanced evaluation frameworks that account for potential memorization effects while recognizing that strong performance on widely used benchmarks does not necessarily indicate data leakage. As code LLMs continue to advance and see broader adoption in software development workflows, understanding these generalization boundaries becomes increasingly important for developing reliable, robust systems.

The significant variations in sensitivity patterns across models with comparable parameter counts suggest that architectural design choices and training methodologies substantially impact generalization capabilities—a finding that has important implications for future model development. By focusing not just on absolute performance metrics but also on perturbation robustness, the field can develop more genuinely capable models that generalize effectively across the full spectrum of code-related tasks. Moreover,

Open Science. To promote transparency and facilitate reproducibility, we make our artifacts available to the community at:

https://github.com/Berickal/CodeLLM_Memo.git

The repository includes the experiment scripts and the results. In addition, A supplementary file containing complementary analysis has been included to provide further details about this work.

Acknowledgments

This work was supported by (1) the Luxembourg Ministry of Foreign and European Affairs through their Digital4Development (D4D) portfolio under the LuxWAyS project and (2) the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (Project NATURAL - Grant agreement N° 949014).

References

[1] 2024. Codestral | Mistral AI. https://mistral.ai/news/codestral.

[2] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.

[3] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[4] Simone Balloccu, Patrícia Schmidtová, Mateusz Lango, and Ondřej Dušek. 2024. Leak, cheat, repeat: Data contamination and evaluation malpractices in closed-source LLMs. *arXiv preprint arXiv:2402.03927* (2024).

[5] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. 30–39.

[6] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. 2022. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258* (2022).

[7] Qi Cao, Takeshi Kojima, Yutaka Matsuo, and Yusuke Iwasawa. 2023. Unnatural Error Correction: GPT-4 Can Almost Perfectly Handle Unnatural Scrambled Text. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 8898–8913. doi:10.18653/v1/2023.emnlp-main.550

[8] Nicholas Carlini, Daphne Ippolito, Matthew Jagielski, Katherine Lee, Florian Tramer, and Chiyuan Zhang. 2022. Quantifying memorization across neural language models. In *The Eleventh International Conference on Learning Representations*.

[9] Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, et al. 2021. Extracting training data from large language models. In *30th USENIX security symposium (USENIX Security 21)*. 2633–2650.

[10] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2022. Deep Learning Based Vulnerability Detection: Are We There Yet? 3280-3296 pages. doi:10.1109/TSE.2021.3087402

[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[12] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. 2023. Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. 654–668.

[13] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. ChatUniTest: A Framework for LLM-Based Test Generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) *(FSE 2024)*. Association for Computing Machinery, New York, NY, USA, 572–576. doi:10.1145/3663529.3663801

[14] Nurit Cohen-Inger, Yehonatan Elisha, Bracha Shapira, Lior Rokach, and Seffi Cohen. 2025. Forget What You Know about LLMs Evaluations - LLMs are Like a Chameleon. doi:10.48550/arXiv.2502.07445 arXiv:2502.07445 [cs].

[15] Verna Dankers and Ivan Titov. 2024. Generalisation First, Memorisation Second? Memorisation Localisation for Natural Language Classification Tasks. In *Findings of the Association for Computational Linguistics: ACL 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 14348–14366. doi:10.18653/v1/2024.findings-acl.852

[16] DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, Wangding Zeng, Xiao Bi, Zihui Gu, Hanwei Xu, Damai Dai, Kai Dong, Liyue Zhang, Yishi Piao, Zhibin Gou, Zhenda Xie, Zhewen Hao, Bingxuan Wang, Junxiao Song, Deli Chen, Xin Xie, Kang Guan, Yuxiang You, Aixin Liu, Qiushi Du, Wenjun Gao, Xuan Lu, Qinyu Chen, Yaohui Wang, Chengqi Deng, Jiashi Li, Chenggang Zhao, Chong Ruan, Fuli Luo, and Wenfeng Liang. 2024. DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence. doi:10.48550/arXiv.2406.11931 arXiv:2406.11931 [cs].

[17] Pantazis Deligiannis, Akash Lal, Nikita Mehrotra, and Aseem Rastogi. 2023. Fixing rust compilation errors using llms. *arXiv preprint arXiv:2308.05177* (2023).

[18] Javid Ebrahimi, Anyi Rao, Daniel Lowd, and Dejing Dou. 2018. Hotflip: White-box adversarial examples for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. 31–36.

[19] Vitaly Feldman and Chiyuan Zhang. 2020. Neural networks and the chomsky hierarchy. In *International Conference on Machine Learning*. PMLR, 3020–3029.

[20] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: a T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*. 935–947.

[21] Matt Gardner, Yoav Artzi, Victoria Basmov, Jonathan Berant, Ben Bogin, Sihao Chen, Pradeep Dasigi, Dheeru Dua, Yanai Elazar, Ananth Gottumukkala, et al. 2020. Evaluating models' local decision boundaries via contrast sets. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 1307–1323.

[22] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[23] Kavi Gupta, Peter Christadler, Hinrich Schütze, and Matt Weir. 2020. Synthesize, execute and debug: Learning to repair for neural program synthesis. In *Advances in Neural Information Processing Systems*. 17931–17942.

[24] Mario Hartmann, Robin Schottkämper, Borja Balle, and Florian Kerschbaum. 2023. SoK: Memorization in Machine Learning. *arXiv preprint arXiv:2309.17393* (2023).

[25] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps (2021). *arXiv preprint arXiv:2105.09938* (2021).

[26] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59, 5 (2016), 122–131.

[27] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing source code with transferred API knowledge. (2018).

[28] Siming Huang, Tianhao Cheng, Jason Klein Liu, Jiaran Hao, Liuyihan Song, Yang Xu, J. Yang, J. H. Liu, Chenchen Zhang, Linzheng Chai, Ruifeng Yuan, Zhaoxiang Zhang, Jie Fu, Qian Liu, Ge Zhang, Zili Wang, Yuan Qi, Yinghui Xu, and Wei Chu. 2024. OpenCoder: The Open Cookbook for Top-Tier Code Large Language Models. https://arxiv.org/pdf/2411.04905

[29] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186* (2024).

[30] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[31] Akshita Jha and Chandan K. Reddy. 2023. CodeAttack: code-based adversarial attacks for pre-trained programming language models. In *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence (AAAI'23/IAAI'23/EAAI'23)*. AAAI Press, Article 1670, 9 pages. doi:10.1609/aaai.v37i12.26739

[32] Di Jin, Zhijing Jin, Joey Tianyi Zhou, and Peter Szolovits. 2020. Is bert really robust? a strong baseline for natural language attack on text classification and entailment. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. 8018–8025.

[33] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1646–1656.

[34] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*. 437–440.

[35] Nikhil Kandpal, Eric Wallace, and Colin Raffel. 2022. Deduplicating training data makes language models better. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 8424–8445.

[36] Divyansh Kaushik, Eduard Hovy, and Zachary C Lipton. 2020. Learning the difference that makes a difference with counterfactually-augmented data. In *International Conference on Learning Representations*.

[37] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. 2019. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461* (2019).

[38] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-Level Code Generation with AlphaCode. *arXiv preprint arXiv:2203.07814* (2022).

[39] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*. 55–56.

[40] Yu Liu, Lang Gao, Mingxin Yang, Yu Xie, Ping Chen, Xiaojin Zhang, and Wei Chen. 2024. Vuldetectbench: Evaluating the deep capability of vulnerability detection with large language models. *arXiv preprint arXiv:2406.07595* (2024).

[41] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. StarCoder 2 and The Stack v2: The Next Generation. doi:10.48550/arXiv.2402.19173 arXiv:2402.19173 [cs].

[42] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. doi:10.48550/arXiv.2306.08568 arXiv:2306.08568 [cs].

[43] Alexandre Matton, Tom Sherborne, Dennis Aumiller, Elena Tommasone, Milad Alizadeh, Jingyi He, Raymond Ma, Maxime Voisin, Ellen Gilsenan-McMahon, and Matthias Gallé. 2024. On Leakage of Code Generation Evaluation Datasets. http://arxiv.org/abs/2407.07565 arXiv:2407.07565 [cs].

[44] Mustafa Safa Ozdayi, Charith Peris, Jack FitzGerald, Christophe Dupuy, Jimit Majmudar, Haidar Khan, Rahil Parikh, and Rahul Gupta. 2023. Controlling the extraction of memorized data from large language models via prompt-tuning. *arXiv preprint arXiv:2305.11759* (2023). https://arxiv.org/abs/2305.11759

[45] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2339–2356.

[46] Dinglan Peng, Shuxin Li, Pingchuan He, Yuhui Chen, Xiangyu Wu, Jianwei Sun, Shengyu Mudgal, Wenhan Ling, Yuxin Tian, Tristan Zhou, et al. 2021. Could neural networks understand code?. In *International Conference on Learning Representations*.

[47] Md Rafiqul Islam Rabin, Vincent J. Hellendoorn, and Mohammad Amin Alipour. 2021. Understanding neural code intelligence through program simplification *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 441–452. doi:10.1145/3468264.3468539

[48] Niklas Risse and Marcel Böhme. 2024. Uncovering the limits of machine learning for automatic vulnerability detection. In *33rd USENIX Security Symposium (USENIX Security 24)*. 4247–4264.

[49] Jason W. Rocks and Pankaj Mehta. 2022. Memorizing without overfitting: Bias, variance, and interpolation in overparameterized models. *Phys. Rev. Res.* 4 (Mar 2022), 013201. Issue 1. doi:10.1103/PhysRevResearch.4.013201

[50] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code Llama: Open Foundation Models for Code. doi:10.48550/arXiv.2308.12950 arXiv:2308.12950 [cs].

[51] Yagnik Sharma, Shruti Sharma, Kartik Subramanian, and Dhananjay Kalbande. 2022. Evaluation of Code Summarization Models: Aspects, Datasets, and Reproducibility. *ACM Transactions on Software Engineering and Methodology* 31, 4 (2022), 1–42.

[52] Ankita Nandkishor Sontakke, Manasi Patwardhan, Lovekesh Vig, Raveendra Kumar Medicherla, Ravindra Naik, and Gautam Shroff. 2022. Code summarization: Do transformers really understand code?. In *Deep Learning for Code Workshop*.

[53] Till Speicher, Mohammad Aflah Khan, Qinyuan Wu, Vedant Nanda, Soumi Das, Bishwamittra Ghosh, Krishna P. Gummadi, and Evimaria Terzi. 2024. Understanding Memorisation in LLMs: Dynamics, Influencing Factors, and Implications. http://arxiv.org/abs/2407.19262 arXiv:2407.19262 [cs].

[54] Michael Tänzer, Sebastian Ruder, and Marek Rei. 2021. Memorisation versus generalisation in pre-trained language models. *arXiv preprint arXiv:2105.00828* (2021).

[55] Wenhan Wang, Ge Li, Bo Shen, Xin Jin, Yuming Xue, and Mingzhe Qin. 2020. 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, 261–271.

[56] Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. 2025. TESTEVAL: Benchmarking Large Language Models for Test Case Generation. arXiv:2406.04531 [cs.SE] https://arxiv.org/abs/2406.04531

[57] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: Empowering Code Generation with OSS-Instruct. doi:10.48550/arXiv.2312.02120 arXiv:2312.02120 [cs].

[58] Yonghao Wu, Zheng Li, Jie M Zhang, and Yong Liu. 2023. Condefects: A new dataset to address the data leakage concern for llm-based fault localization and program repair. *arXiv preprint arXiv:2310.16253* (2023).

[59] Chulin Xie, Yangsibo Huang, Chiyuan Zhang, Da Yu, Xinyun Chen, Bill Yuchen Lin, Bo Li, Badih Ghazi, and Ravi Kumar. 2024. On Memorization of Large Language Models in Logical Reasoning. http://arxiv.org/abs/2410.23123 arXiv:2410.23123.

[60] Xiaoyong Yuan and Lan Zhang. 2022. Membership inference attacks and defenses in neural network pruning. In *31st USENIX Security Symposium (USENIX Security 22)*. 4561–4578.

[61] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. 2021. Understanding deep learning (still) requires rethinking generalization. *Commun. ACM* 64, 3 (2021), 107–115.

[62] Ying Zhang, Wenjia Song, Zhengjie Ji, Na Meng, et al. 2023. How well does LLM generate security tests? *arXiv preprint arXiv:2310.00710* (2023).

[63] Yuhao Zhang, Shiqi Wang, Haifeng Qian, Zijian Wang, Mingyue Shang, Linbo Liu, Sanjay Krishna Gouda, Baishakhi Ray, Murali Krishna Ramanathan, Xiaofei Ma, et al. 2024. CodeFort: Robust training for code generation models. *arXiv preprint arXiv:2405.01567* (2024).

[64] Chengrun Zhou, Wenhan Liu, Sifan Wang, Yue Sun, Rohan Panda, Yuandong Tian, Huan Ma, Ding Yu, Dunzhao Yang, Inder S Dhillon, et al. 2023. Large Language Models as Optimizers. *arXiv preprint arXiv:2309.03409* (2023).

[65] Kun Zhou, Yutao Zhu, Zhipeng Chen, Wentong Chen, Wayne Xin Zhao, Xu Chen, Yankai Lin, Ji-Rong Wen, and Jiawei Han. 2023. Don't Make Your LLM an Evaluation Benchmark Cheater. http://arxiv.org/abs/2311.01964 arXiv:2311.01964 [cs].

[66] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. arXiv:1909.03496 [cs.SE] https://arxiv.org/abs/1909.03496

[67] Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K. Reddy. 2022. XLCoST: A Benchmark Dataset for Cross-lingual Code Intelligence. arXiv:2206.08474 https://arxiv.org/abs/2206.08474

[68] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. 2024. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. *arXiv preprint arXiv:2406.15877* (2024).