From Prompts to Properties: Rethinking LLM Code Generation with Property-Based Testing

Dibyendu Brinto Bose
brintodibyendu@vt.edu
Virginia Tech
Blacksburg, Virginia, USA

Abstract

Large Language Models (LLMs) have shown promise in automated code generation, but ensuring correctness remains a significant challenge. Traditional unit testing evaluates functional correctness but often fails to capture deeper logical constraints. We apply Property-Based Testing (PBT) as an alternative evaluation strategy to StarCoder and CodeLlama on MBPP and HumanEval. Our results reveal that while pass@k evaluation shows moderate success, PBT exposes additional correctness gaps. A significant portion of generated solutions only partially adhere to correctness properties (30–32%), while 18–23% fail outright. Property extraction is also imperfect, with 9–13% of constraints missing. These findings highlight that unit test-based evaluations may overestimate solution correctness by not capturing fundamental logical errors. Our study demonstrates that combining unit testing with PBT can offer a more comprehensive assessment of generated code correctness, revealing limitations that traditional verification approaches miss.

CCS Concepts

• **Software and its engineering → Functionality**.

Keywords

Code generation, property-based-testing, Large language model(LLM)

ACM Reference Format:
Dibyendu Brinto Bose. 2025. From Prompts to Properties: Rethinking LLM Code Generation with Property-Based Testing. In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), June 23–28, 2025, Trondheim, Norway.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3696630.3728702

1 Introduction

Recent advances in Large Language Models (LLMs) have transformed software engineering. For instance, LLMs are increasingly used to support software development tasks [20, 24], such as code generation [14]. Recently, these models have significantly improved their ability to generate code [9, 13], making them valuable tools for software development and automation [4]. For example, GitHub suggests that approximately half of developers' code across programming languages—and over 60% of Java code—was generated by GitHub Copilot.[1] However, LLM-generated code can face numerous issues, such as hallucination [12], security [23] and miss important functionalities [2]. Thus, techniques are needed to assess the quality of LLM-generated code.

The predominant method for evaluating and refining code generated by LLMs relies on unit testing [17, 18], which primarily verifies specific input-output pairs. While unit tests serve as a useful validation tool, they capture only a limited aspect of computational thinking. This approach overlooks the deeper structural and behavioral correctness of the code, which is essential for robust software development [3]. As a result, LLM-generated code may pass unit tests while still containing fundamental logical errors, failing in edge cases or unforeseen scenarios [15].

To address these limitations, we investigate the role of Property-Based Testing (PBT) as a complementary verification mechanism for LLM-generated code. Instead of solely relying on example-based validation, PBT enables a structured assessment of whether generated solutions adhere to fundamental correctness constraints. We examine how well LLMs can extract and incorporate properties into code generation and assess the effectiveness of PBT in capturing logical correctness beyond traditional unit tests.

In this study, we explore the impact of PBT in LLM-driven code generation and investigate how different verification approaches influence the quality of generated programs. The following key questions guide our research:

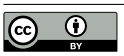
RQ1: How effectively can Large Language Models (LLMs) generate correct and executable code when guided by property-based testing?

RQ2: How does unit test-based evaluation compare to property-based testing in assessing LLM-generated code correctness?

To answer these questions, we followed the following pipeline to apply property-based reasoning to LLM-generated code. Our approach consists of follwing key steps:

- Prompting LLMs to derive key correctness properties from problem descriptions and generate multiple code solutions guided by these extracted properties
- Verifying correctness through both property-based testing (PBT) and unit-test-based evaluation

We evaluate two source code focused open source LLMs—including *StarCoder* [10],[2] and *Codellama*[3]—on the HumanEval[4] and Mostly Basic Python Problems (MBPP)[5] datasets, measuring their success



FSE Companion '25, Trondheim, Norway

ACM ISBN 979-8-4007-1276-0/2025/06
https://doi.org/10.1145/3696630.3728702

[1] https://github.blog/2023-02-14-github-copilot-now-has-a-better-ai-model-and-new-capabilities/
[2] https://huggingface.co/blog/starcoder
[3] https://github.com/meta-llama/codellama
[4] https://github.com/openai/human-eval
[5] https://github.com/google-research/google-research/tree/master/mbpp

in generating functionally correct and logically sound code. Our contributions are as follows:

- We conduct a comparative evaluation of multiple LLMs (StarCoder and CodeLlama) under both property-based testing and unit-test-based verification
- We assess the impact of generating multiple solutions (pass@k) in improving correctness across evaluation strategies
- We provide empirical insights into the strengths and limitations of PBT versus unit testing in guiding and validating LLM-generated code

Our analysis and source code are given in this GitHub repository.[6]

2 Related Work

2.1 LLM-Based Code Generation

Large Language Models (LLMs) have demonstrated impressive capabilities in code generation, yet ensuring correctness remains a significant challenge. Prior research has explored multiple strategies to enhance the robustness of LLM-generated programs [7, 11]. LLM-based fuzzing [16] has been introduced to improve test case diversity, addressing limitations in code coverage and generalization. Similarly, as seen in tools like JQF, coverage-guided fuzzing refines input generation based on execution feedback to improve bug detection [21]. However, these methods primarily optimize input diversity rather than enforcing correctness properties. Other works have attempted to mitigate errors by improving requirement comprehension, such as ClarifyGPT [15], which prompts LLMs to ask clarifying questions before code generation. Additionally, domain-specific LLMs, such as those tailored for RTL design automation [13], highlight the importance of structured constraints and high-quality training datasets in enhancing model reliability. While these approaches refine the quality of LLM-generated code, they do not systematically enforce correctness constraints throughout the generation process.

2.2 Property-Based Testing

Property-based testing (PBT) offers an alternative paradigm by defining correctness properties that must hold across a wide range of inputs. Unlike traditional unit tests, which validate example-based correctness, PBT systematically evaluates whether generated functions satisfy higher-level structural and logical constraints [17]. Formal verification methods, such as QuickChick [6], extend PBT by integrating property validation within proof assistants, enabling rigorous correctness verification. Previously, researchers focused on automatically synthesizing property-based tests (PBTs) from API documentation using LLMs [25] and evaluating their validity, soundness, and property coverage. In contrast, our study investigates the effectiveness of PBT as a verification strategy for LLM-generated code by extracting correctness properties and assessing compliance through execution-based evaluation.

3 Methodology

We investigate the effectiveness of property-based reasoning in LLM-driven code generation by evaluating two source-code-focused LLMs, StarCoder and CodeLlama. We evaluate PBT with these models by leveraging two widely used benchmarks: HumanEval and MBPP. The experiments are conducted using Hugging Face model deployments on Google Colab.[7]

The visualization of our process for property generation and source code generation is displayed in Figure 1. We first extract correctness properties by prompting the LLMs with problem descriptions and explicitly asking them to list essential constraints that a correct implementation should satisfy. After that, we asked the LLM to provide solutions using those generated properties.

3.1 Datasets

To evaluate the impact of property-based reasoning on LLM-driven code generation, we conduct experiments on two widely used benchmarks: HumanEval and MBPP (Mostly Basic Programming Problems). These datasets provide complementary evaluation settings—HumanEval emphasizes functional correctness through docstring-based specifications, while MBPP includes explicit unit tests, allowing us to compare unit-test-based evaluation with property-based verification.

HumanEval consists of 164 Python problems, each including a problem description, function signature, and example-based assertions embedded within docstrings. Since HumanEval lacks explicit unit tests, we adopt an alternative evaluation strategy by using docstring-based functional correctness checks and applying property-based testing (PBT) to validate correctness constraints extracted from problem descriptions.

MBPP comprises 974 Python problems, structured with a problem statement, reference implementation, and executable unit tests. The availability of predefined unit tests allows us to directly evaluate standard pass@k metrics while also benchmarking property-based verification against traditional unit-test-based correctness assessment.

We randomly sampled 100 problems from each dataset to facilitate manual property mapping and analyzed their structural characteristics. The MBPP sample consists of shorter programs (avg. 5.93 lines of code) with a higher number of function calls (4.32) but fewer loops (0.7) and conditionals (0.75), reflecting its focus on basic algorithmic tasks. In contrast, HumanEval problems are longer (avg. 8.16 lines) with more loops (1.03) and conditionals (1.61), indicating greater logical complexity. Additionally, HumanEval relies on assertions (avg. 7.96 per task) for correctness verification instead of standalone unit test cases.

3.2 Prompting for Property Extraction and Code Generation

We employ structured instruction-based prompting [19] to extract correctness properties and generate code solutions. The prompts enforce a strict output format to ensure the responses are structured and useful.

[6] https://anonymous.4open.science/r/pbt_generation-D465/README.md

[7] https://colab.research.google.com/

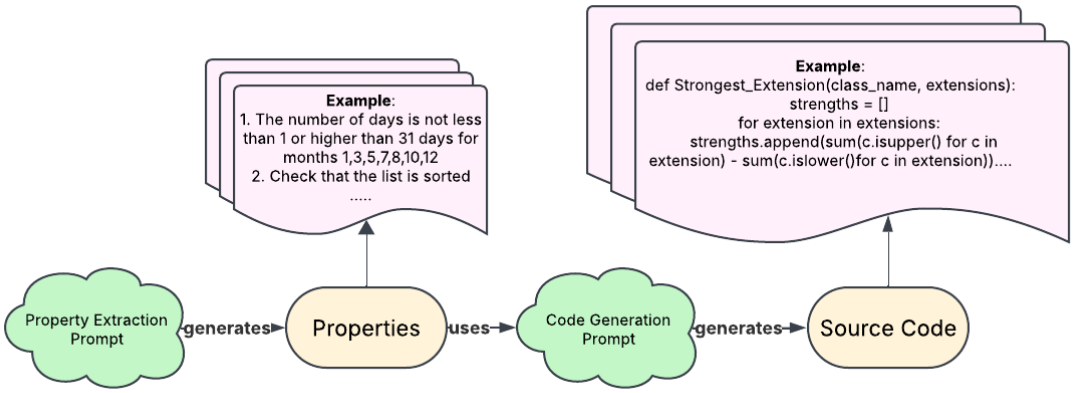


Figure 1: Pipeline of property generation and source code generation.

Property Extraction Prompt:

> **Task:** Extract key correctness properties from the following problem description.
> **Instructions:** - Identify essential correctness properties the function must satisfy. - Output properties in bullet-point format.
> **Problem Statement:** `<INSERT PROBLEM DESCRIPTION>`
> **Extracted Properties:**

Code Generation Prompt:

> **Task:** Generate a Python function that satisfies the given correctness properties.
> **Instructions:** - Ensure the function adheres to all listed properties. - Use clean and efficient Python code.
> **Properties:** `<INSERT EXTRACTED PROPERTIES>`
> **Generated Code:**

3.3 Property-Based Extraction and Mapping

After extracting properties, we perform manual property mapping, where the first author categorizes the extracted properties into valid, redundant, missing, or incorrect based on the generated response of LLMs. This ensures that the extracted properties align with the expected behavior before using them for code generation and verification. Table 1 summarizes the key property types considered in our study.

To generate solutions, we provide the LLMs with a structured prompt discuss in subsection 3.2 that includes both the problem statement and the extracted correctness properties. The models generate ten independent solutions per task, allowing us to evaluate pass@1, pass@5, and pass@10, which measure the probability that at least one of the top-k generated solutions is correct.

3.4 Evaluation Strategy

We employ the following evaluation strategies:

- **Unit Testing (MBPP)**: MBPP includes predefined unit test cases, which we use to evaluate the generated code directly. Since HumanEval does not provide explicit unit tests, we assess correctness by using its docstring specifications to validate expected behaviors.
- **Property-Based Testing (PBT)**: We generate property-based test cases using the Hypothesis library[8], leveraging the manually mapped properties to create systematic tests. Specifically, for each problem, we define test cases that validate whether the generated code adheres to the expected properties. For example, given a function that sorts a list, a mapped property could be: "The output list should be sorted in ascending order." Using Hypothesis, we automatically generate randomized input lists and assert that the function's output remains sorted:

```
from hypothesis import given, strategies as st

@given(st.lists(st.integers()))
def test_sorted_property(lst):
    assert solution(lst) == sorted(lst)

test_sorted_property()
```

4 Results

4.1 Answer to RQ1:

We analyze property extraction performance and compliance with property-based testing (PBT) to evaluate how effectively LLMs extract and utilize correctness properties.

Table 2 shows that StarCoder extracts valid properties in 76% of MBPP cases and 74% of HumanEval cases, while CodeLlama achieves 72% and 70%, respectively. However, both models exhibit redundancy (15–18%) and miss constraints in 9–13% of cases, indicating that property extraction is imperfect and can omit key correctness criteria.

[8] https://hypothesis.readthedocs.io/en/latest/

Table 1: Key property types used in Property-Based Testing and their corresponding explanations.

Property Type	Example Explanation
Correct Output Consistency	The output must match expected values for given inputs.
Data Type Consistency	The function should return the correct data type as expected (e.g., list, float, boolean).
Edge Case Handling	The function should correctly handle empty lists, zero values, negative numbers, etc.
Sorting Constraint	If a function returns a list, it should be sorted in the expected order.
No Side Effects	The function should not modify its input parameters in-place unless explicitly required.
Idempotency	If a function is applied multiple times, the result should remain the same (e.g., sorting functions).
Inverse Functionality	If applicable, applying a function's inverse should return the original value (e.g., encryption/decryption functions).
Associative or Commutative Property	If applicable, operations should be associative/commutative (e.g., addition, multiplication).
Range Constraints	Outputs should be within a defined range (e.g., probabilities between 0 and 1).
Uniqueness Constraints	The function should return unique elements where required (e.g., set operations).
Numerical Stability	Functions dealing with floating-point arithmetic should not lose precision.
String Handling Consistency	Functions that process strings should be case-sensitive/case-insensitive as specified.

Table 2: Property Extraction Performance: Valid, Redundant, and Missing Properties Across MBPP and HumanEval datasets (100 samples).

Model	Dataset	Valid (%)	Redundant (%)	Missing (%)
StarCoder	MBPP	76	15	9
CodeLlama	MBPP	72	18	10
StarCoder	HumanEval	74	14	12
CodeLlama	HumanEval	70	17	13

Table 3: Property-Based Testing Compliance: Fully Passed, Partially Passed, and Failed Cases Across MBPP and HumanEval datasets (100 samples).

Model	Dataset	Property-Based Testing Compliance (%)		
		Fully Passed	Partially Passed	Failed
StarCoder	MBPP	52	30	18
CodeLlama	MBPP	48	32	20
StarCoder	HumanEval	49	29	22
CodeLlama	HumanEval	43	31	23

When examining compliance with extracted properties (Table 3), only 52% of StarCoder's MBPP-generated solutions fully pass PBT, with 30% partially correct and 18% failing. Similarly, 49% of HumanEval solutions fully adhere to extracted properties, while 29% are partially correct and 22% fail outright. CodeLlama exhibits slightly lower compliance, with 48% full PBT adherence in MBPP and 43% in HumanEval.

These findings indicate that, **while LLMs can extract and use correct properties to guide code generation, they frequently miss constraints and struggle to fully adhere to extracted properties**. This suggests property-based reasoning provides structured guidance but requires improved property extraction and enforcement.

4.2 Answer to RQ2:

Table 4: Pass@k Performance for StarCoder and CodeLlama on MBPP dataset (100 samples).

Model	Pass@k	Min	Mean	Max
StarCoder	Pass@1	0.00	0.491	1.00
	Pass@5	0.01	0.521	0.74
	Pass@10	0.07	0.588	0.81
CodeLlama	Pass@1	0.00	0.535	1.00
	Pass@5	0.03	0.573	0.71
	Pass@10	0.09	0.61	0.85

To compare unit-test-based and property-based evaluations, we analyze Pass@k scores (Tables 4 and 5) alongside PBT results.

For unit test-based evaluation, StarCoder achieves a Pass@1 of 0.491 on MBPP and 0.46 on HumanEval, increasing to 0.588 and 0.533 at Pass@10, respectively. CodeLlama exhibits higher Pass@1 scores of 0.535 (MBPP) and 0.500 (HumanEval), improving to 0.61

Table 5: Pass@k Performance for StarCoder and CodeLlama on HumanEval dataset (100 samples).

Model	Pass@k	Min	Mean	Max
StarCoder	Pass@1	0.00	0.46	1.00
	Pass@5	0.02	0.517	0.78
	Pass@10	0.06	0.533	0.86
CodeLlama	Pass@1	0.00	0.500	1.00
	Pass@5	0.06	0.52	0.75
	Pass@10	0.08	0.594	0.82

and 0.594 at Pass@10. These scores suggest that unit tests provide a structured way to measure correctness but rely on predefined test cases that may not capture logical constraints.

In contrast, PBT compliance results show that even when solutions pass unit tests, they may still fail fundamental correctness properties. For example, in MBPP, 52% (StarCoder) and 48% (CodeLlama) fully pass PBT, which is lower than their Pass@10 scores (0.588 and 0.61, respectively). This suggests that unit tests alone may overestimate correctness, since a solution passing predefined test cases might still violate correctness constraints captured by PBT. **Overall, unit test-based evaluation provides a more optimistic correctness estimate, while property-based testing exposes additional logical and structural flaws in generated code.** This underscores the complementary nature of PBT and unit testing, where PBT serves as an additional verification layer beyond traditional test cases.

5 Discussion and Implications

Our findings underscore both the limitations and potential of property-based testing (PBT) in guiding LLM-driven code generation.

Limitations While property extraction introduces explicit correctness constraints, its effectiveness is inconsistent. As seen in Table 2, a considerable fraction of extracted properties were either redundant (15–18%) or missing (9–13%), highlighting the challenges LLMs face in consistently capturing all necessary constraints. The presence of redundant properties suggests models may overgenerate constraints, while the missing ones indicate that critical correctness conditions may be overlooked. This inconsistency limits the direct applicability of extracted properties.

Potential Despite these challenges, PBT introduces a valuable correctness framework that extends beyond unit testing. Table 3 shows that while 52% of StarCoder's generated solutions for MBPP fully complied with PBT, a significant fraction (30%) were only partially correct. This suggests that property-guided generation improves structural adherence. However, LLMs still struggle to fully integrate all constraints into their generated solutions. Furthermore, the discrepancy between Pass@k performance and PBT compliance indicates that many solutions that pass unit tests still violate essential correctness properties. For instance, StarCoder achieves a mean Pass@10 of 0.588 on MBPP, yet only 52% of its solutions fully satisfy property-based constraints, illustrating the limitations of unit test-only evaluation.

Implications Developers increasingly use LLMs to generate code, yet report lacking trust in the accuracy of generated output [1]. Our results have important implications for validating LLM-generated code. First, unit test-based evaluation alone provides an incomplete correctness measure, failing to capture deeper structural flaws. PBT offers a more nuanced perspective by enforcing logical constraints, ensuring that generated code adheres to expected behaviors beyond input-output mappings. However, the inconsistency in property extraction suggests the need for improved property refinement mechanisms, such as ranking, filtering, or human-in-the-loop validation, to enhance the quality of extracted constraints. In addition, LLMs struggle with strict property adherence, indicating that more effective mechanisms are needed to guide models toward better constraint integration.

Research shows PBT improves correctness enforcement and confidence in complex systems [8], yet it is rarely used in practice [5]. Given this fact alongside our findings, we suggest that a hybrid approach combining unit testing and property-based reasoning can provide a more reliable evaluation of LLM-generated code. While unit tests provide a quick correctness measure, PBT ensures that the generated solutions follow fundamental logical constraints. Future research could explore methods to strengthen property extraction reliability, enhance property adherence in code generation, and integrate reinforcement learning or retrieval-based prompting to optimize PBT-driven LLM performance.

6 Limitations

Our study provides valuable insights into the role of PBT in evaluating LLM-generated code, but several limitations must be acknowledged.

6.1 Internal Validity

Our study focuses on a limited set of LLMs (StarCoder and CodeLlama), and our results may not fully generalize to other models with different architectures or training paradigms. Differences in model capabilities, particularly in reasoning and property extraction, may influence comparative outcomes. Future work is needed to investigate the effectiveness of property-based testing of LLM-based code generation with other models.

6.2 External Validity

We utilized HumanEval and MBPP, which are widely recognized benchmarks for code generation. However, these datasets may not fully capture the complexities of real-world software. Our results may not generalize in practical settings with larger and more intricate codebases, where correctness constraints are more nuanced and domain-specific. Additionally, our experiments were conducted in a controlled environment. We sampled 100 random problems from each dataset to enable feasible computation for property mapping. This sampling strategy, while necessary for practicality, may introduce bias and limit the generalizability of our findings. Furthermore, the manual property validation process introduces the possibility of human errors, potentially affecting the accuracy of our property mappings and evaluations.

7 Future Work

Future research should focus on improving the scalability and reliability of PBT in LLM-driven code generation. We will investigate hybrid unit and PBT approaches. One critical direction is developing more robust property extraction methods, as our results indicate that current LLMs often miss essential constraints. Leveraging prompt engineering, fine-tuning strategies or external knowledge sources could enhance the accuracy of extracted correctness properties. Another promising direction is refining LLM adherence to extracted properties. While LLMs can generate property-guided code, ensuring strict enforcement of these constraints remains challenging. Exploring reinforcement learning, retrieval-augmented generation (RAG), or constraint-aware decoding techniques could improve model compliance with correctness properties.

8 Conclusion

This study explores the role of property-based reasoning in improving the correctness of LLM-generated code. By evaluating StarCoder and CodeLlama on the MBPP and HumanEval datasets, we demonstrated that explicitly guiding models with correctness properties enhances code generation performance. Our findings reveal that while PBT provides structured verification, LLMs struggle with redundant and missing constraints, limiting their ability to adhere to extracted correctness properties fully. Additionally, we compare unit testing and PBT—highlighting their complementary nature. Unit tests measure correctness based on predefined test cases, yet PBT verifies adherence to extracted constraints, capturing logical and structural correctness aspects beyond example-based validation. These results suggest a hybrid approach combining both methods offers a more comprehensive framework for assessing LLM-generated code correctness.

9 Acknowledgment

I would like to thank my advisor, Dr. Chris Brown, for his invaluable guidance throughout this work. The original idea for this project was contributed by Haoran Zhang from Penn State University, as a follow-up to the prior meta-study [22].

References

[1] [n. d.]. AI | 2024 Stack Overflow Developer Survey — survey.stackoverflow.co. https://survey.stackoverflow.co/2024/ai#sentiment-and-usage-ai-select. [Accessed 02-08-2024].

[2] Dibyendu Brinto Bose and Chris Brown. 2024. An Empirical Study on Current Practices and Challenges of Core AR/VR Developers. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops*. 233–238.

[3] Tristan Coignion, Clément Quinton, and Romain Rouvoy. 2024. A performance study of llm-generated code on leetcode. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. 79–89.

[4] Arthur Lisboa Corgozinho, Marco Tulio Valente, and Henrique Rocha. 2023. How Developers Implement Property-Based Tests. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 380–384. doi:10.1109/ICSME58846.2023.00049

[5] Arthur Lisboa Corgozinho, Marco Tulio Valente, and Henrique Rocha. 2023. How developers implement property-based tests. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 380–384.

[6] Maxime Dénès, Catalin Hritcu, Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. 2014. QuickChick: Property-based testing for Coq. In *The Coq Workshop*, Vol. 125. 126.

[7] Christopher Foster, Abhishek Gulati, Mark Harman, Inna Harper, Ke Mao, Jillian Ritchey, Hervé Robert, and Shubho Sengupta. 2025. Mutation-Guided LLM-based Test Generation at Meta. *arXiv preprint arXiv:2501.12862* (2025).

[8] Harrison Goldstein, Joseph W Cutler, Daniel Dickstein, Benjamin C Pierce, and Andrew Head. 2024. Property-based testing in practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[9] Qiuhan Gu. 2023. LLM-Based Code Generation Method for Golang Compiler Testing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 2201–2203. doi:10.1145/3611643.3617850

[10] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! arXiv:2305.06161 [cs.CL] https://arxiv.org/abs/2305.06161

[11] Ziyu Li and Donghwan Shin. 2024. Mutation-based Consistency Testing for Evaluating the Code Understanding Capability of LLMs. In *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering - Software Engineering for AI* (Lisbon, Portugal) *(CAIN '24)*. Association for Computing Machinery, New York, NY, USA, 150–159. doi:10.1145/3644815.3644946

[12] Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, Li Zhang, Zhongqi Li, and Yuchi Ma. 2024. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971* (2024).

[13] Shang Liu, Wenji Fang, Yao Lu, Jing Wang, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. 2024. RTLCoder: Fully Open-Source and Efficient LLM-Assisted RTL Code Generation Technique. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024), 1–1. doi:10.1109/TCAD.2024.3483089

[14] Zhijie Liu, Yutian Tang, Xiapu Luo, Yuming Zhou, and Liang Feng Zhang. 2024. No need to lift a finger anymore? assessing the quality of code generation by chatgpt. *IEEE Transactions on Software Engineering* (2024).

[15] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, ChenXue Wang, Shichao Liu, and Qing Wang. 2024. ClarifyGPT: A Framework for Enhancing LLM-Based Code Generation via Requirements Clarification. *Proc. ACM Softw. Eng.* 1, FSE, Article 103 (July 2024), 23 pages. doi:10.1145/3660810

[16] Yaroslav Oliinyk, Michael Scott, Ryan Tsang, Chongzhou Fang, Houman Homayoun, et al. 2024. Fuzzing {BusyBox}: Leveraging {LLM} and Crash Reuse for Embedded Bug Unearthing. In *33rd USENIX Security Symposium (USENIX Security 24)*. 883–900.

[17] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: coverage-guided property-based testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 398–401. doi:10.1145/3293882.3339002

[18] Zoe Paraskevopoulou, Cătălin HriŢcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. 2015. Foundational Property-Based Testing. In *Interactive Theorem Proving*, Christian Urban and Xingyuan Zhang (Eds.). Springer International Publishing, Cham, 325–343.

[19] Archiki Prasad, Peter Hase, Xiang Zhou, and Mohit Bansal. 2022. Grips: Gradient-free, edit-based instruction search for prompting large language models. *arXiv preprint arXiv:2203.07281* (2022).

[20] Daniel Russo. 2024. Navigating the complexity of generative ai adoption in software engineering. *ACM Transactions on Software Engineering and Methodology* 33, 5 (2024), 1–50.

[21] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 1 (2024), 85–105. doi:10.1109/TSE.2023.3334955

[22] Chenglei Si, Diyi Yang, and Tatsunori Hashimoto. 2024. Can llms generate novel research ideas? a large-scale human study with 100+ nlp researchers. *arXiv preprint arXiv:2409.04109* (2024).

[23] Claudio Spiess, David Gros, Kunal Suresh Pai, Michael Pradel, Md Rafiqul Islam Rabin, Susmit Jha, Prem Devanbu, and Toufique Ahmed. 2024. Quality and Trust in LLM-generated Code. *arXiv e-prints* (2024), arXiv–2402.

[24] Rosalia Tufano, Antonio Mastropaolo, Federica Pepe, Ozren Dabic, Massimiliano Di Penta, and Gabriele Bavota. 2024. Unveiling ChatGPT's Usage in Open Source Projects: A Mining-based Study. In *Proceedings of the 21st International Conference on Mining Software Repositories*. 571–583.

[25] Vasudev Vikram, Caroline Lemieux, Joshua Sunshine, and Rohan Padhye. 2023. Can large language models write good property-based tests? *arXiv preprint arXiv:2307.04346* (2023).