Toward Automated, Contamination-free Dafny Benchmark Generation

Changjie Wang
KTH Royal Institute of Technology
Sweden

Mariano Scazzariello
RISE Research Institutes of Sweden
Sweden

Dejan Kostić
KTH Royal Institute of Technology
Sweden

Marco Chiesa
KTH Royal Institute of Technology
Sweden

Abstract

Recently, Large Language Models (LLMs) have been widely applied to code generation, and recent studies have explored their ability to produce programs in verification languages, including Dafny. Although several benchmarks have been proposed to assess LLM performance in generating Dafny code, existing datasets suffer from several limitations, including no clear verification goals, limited diversity in verification complexity, potential contamination, and heavy reliance on manually collected examples that are publicly available online. In this work, we introduce TACoDafny, an automated pipeline for constructing Dafny verification benchmarks directly from task descriptions sourced from online programming repositories. As a preliminary result, we successfully generated 76 Dafny programs with verifiable properties derived from the task descriptions of the first 200 LeetCode problems. Our pipeline not only enables large-scale benchmark generation, but also facilitates the creation of additional verifiable Dafny code that can be leveraged to train models to further improve their verification capabilities.

1 Introduction

The rise of LLM-based code assistants. AI-assisted code generation is rapidly transforming modern software development workflows. Tools such as GitHub Copilot [16], Cursor AI [6], and Amazon Q Developer [1] accelerate programming tasks through natural language-to-code translation, autocompletion, automated refactoring. Although these systems significantly improve developer productivity, they frequently produce code that is incorrect or vulnerable due to model's hallucinations, insufficient reasoning capabilities, and adversarial poisoning [23]. As such, ensuring the correctness of LLM-generated code remains a fundamental challenge.

Formal verification as a way to improve LLM-generated code. A promising direction to increase code reliability is to leverage formal verification tools that automatically check whether a program satisfies its specified behavioral contract. Such contracts may include preconditions, postconditions, termination clauses, and state framing clauses, with postconditions defining the expected state after execution. A wide range of verification tools exists to reason about these contracts, from SMT-based verifiers to proof assistants and model checkers. Among them, Dafny [12] stands out as a verification-aware programming language with native support for design-by-contract, enabling automated checking of functional correctness.

Leveraging formal verification tools is challenging. Despite being more accessible than traditional proof assistants, formal verification tools like Dafny still require non-trivial manual effort, including writing auxiliary annotations, specifying termination arguments, and providing loop invariants or lemmas. This hinders large-scale adoption in practical industrial development settings. Recently, Large Language Models (LLMs) have demonstrated growing potential in generating not only executable code, but also verification-oriented code, such as specifications, invariants, and proofs [3, 10, 24]. From this perspective, proof generation can be viewed as a specialized form of code generation, even though it demands substantially deeper logical reasoning. As recent LLMs increasingly incorporate advanced reasoning mechanisms, this direction is becoming more feasible, motivating growing interest in verification-aware code synthesis, particularly in domains such as Dafny.

Dafny benchmarks lag behind established code generation benchmarks. LLMs performance is usually assessed through targeted benchmarks tailored to specific tasks, for instance, HumanEval [4], CodeContests [13], or BigCodeBench [25] for code generation, MATH [8] for mathematical reasoning, and NetConfEval [22] for network configuration synthesis. These benchmarks provide clearly defined objectives and structured difficulty levels, which enable systematic and reproducible evaluation of LLM capabilities.

Compared to mature code synthesis benchmarks, current Dafny evaluation benchmarks exhibit four critical shortcomings: (*i*) *unclear verification goals*, code is provided without specifying what should be verified; (*ii*) *lack of structure*, the benchmark is a collection of programs without, *e.g.,* a clear progression of difficulty; (*iii*) *contamination*, a publicly available benchmark can be included in training data, artificially increasing LLM performance [9, 19, 20]; and (*iv*) *manual creation*, benchmark's examples are handcrafted and therefore costly and time-consuming to expand. These limitations

hinder systematic, large-scale evaluation of LLMs in Dafny formal verification.

A contamination-free pipeline for Dafny benchmarks. In this work, we aim to address the limitations of existing benchmarks by introducing TACoDafny, an automated pipeline that generates Dafny verification tasks directly from widely used programming competition problems. Each task consists of an executable program paired with a corresponding postcondition.[1] This design enables both the program and its formal contracts to be validated against a suite of automatically generated test cases, ensuring that the executable code and postconditions are aligned with the original problem description. Since all tasks are synthesized automatically, users can regenerate the benchmark entirely from scratch, ensuring reproducibility and inherently mitigating risks of data contamination.

Furthermore, we introduce a public leaderboard, available at tacodafny.github.io, to compare model performance on TACoDafny. Rather than distributing a static benchmark, whose public data could end up in the training dataset of an LLM, we release a parameterized generation script, enabling transparent reproduction and fair comparison under consistent conditions.

Contributions. Our contributions are:

- **A fully automated benchmark generation pipeline for Dafny verification.** We design a reproducible pipeline that converts common programming competition problems into Dafny verification tasks, each consisting of an executable program and a contract to be verified.
- **Functional consistency between implementations and specifications.** Our pipeline ensures that both the executable code and its corresponding postcondition are validated using automatically generated test cases, guaranteeing alignment with the original problem intent.
- **A contamination-resistant, reproducible benchmark.** Instead of releasing a static dataset, we provide a generation script that allows researchers to recreate the entire benchmark, mitigating data leakage and facilitating fair comparison.

2 Related Work

Several benchmarks have been proposed to evaluate the performance of LLMs on Dafny code generation, but they exhibit several issues that reduce their effectiveness (see Sec. 1). In the following, we examine the existing Dafny benchmarks, highlighting their strengths and limitations. Table 1 provides a summary of the different datasets.

CloverBench. CloverBench [21] is one of the earliest benchmarks for Dafny code generation with LLMs, consisting of 60 handcrafted examples at a textbook level of difficulty. While it provides clear verification goals and a well-defined structure, its size is limited because all examples are created *manually*. In addition, since CloverBench is *publicly available*, it may have been included in model training data.

DafnySynthesis. DafnySynthesis [17] contains 153 samples, combining 50 manually curated examples with 103 generated by GPT-4 from the MBPP (Mostly Basic Python Programming) dataset. However, it only checks Dafny verification without ensuring *correct or strong postconditions*, offers no learning curve as MBPP covers *basic concepts*, and, like CloverBench, suffers from *contamination* since all samples are publicly available.

DafnyBench. DafnyBench [15] is one of the most comprehensive benchmarks for Dafny code generation, with over 700 programs collected from more than 100 GitHub repositories, including CloverBench and DafnySynthesis. However, it *lacks structure*, mixing examples without any indication of difficulty, and its *verification goals are unclear*, as it provides only Dafny code without natural language specifications. The dataset is created in a largely automated way, but like previous benchmarks, its main limitation is *contamination*, since it is built from publicly available repositories and is itself openly accessible.

DafnyGym. DafnyGym [18] is a benchmark of complex lemmas extracted from three real-world Dafny codebases. Since it is derived from such projects, it contains only difficult problems, with no structure or progression in difficulty. On the positive side, the dataset has *clear verification goals*, as the task is to generate a single missing assertion line. Its construction is semi-automated: while repositories are selected *manually*, the individual samples are generated automatically, making the dataset difficult to expand. Finally, like previous benchmarks, DafnyGym is based on existing Dafny repositories and is therefore subject to *contamination*.

Desiderata. As the above discussion shows, all existing Dafny benchmarks suffer from one or more shortcomings that limit their effectiveness in evaluating LLMs in Dafny code generation. A key issue is *contamination*, as most benchmarks are publicly available and may have been included in model training, thus inflating performance results. Another limitation is the absence of a structured difficulty curve: benchmark's datasets either mix examples of varying complexity or focus only on either simple or complex problems. This makes it difficult to assess the true capabilities of LLMs, as we cannot pinpoint the level at which models fail or determine the gap in their ability to generate correct code.

3 TACoDafny Creation

We curate TACoDafny, an automatically generated Dafny-based benchmark where each task consists of a program and a clearly defined verification goal expressed as a postcondition. In this paper, we mainly focus on LeetCode [11], a

[1] The postcondition is written in a "non-ghost" form, ensuring that it can be executed.

Name	Clear Verification Goals	Defined Structure	No Contamination	Automatic Generation
CloverBench [21]	✓	✓	✗	✗
DafnySynthesis [17]	✗	✗	✗	✗
DafnyBench [15]	✗	✗	✗	✗
DafnyGym [18]	✓	✗	✗	✗
TACoDafny	✓	✓	✓	✓

Table 1. Comparison among existing datasets.

widely adopted competitive programming platform, but our methodology is generalizable to other programming repositories such as AtCoder [2] and CodeForces [5]. Each problem includes a natural language description and example input-output pairs, with the objective of implementing a program that passes hidden tests. LLMs have demonstrated strong capabilities to generate solutions that pass such tests in mainstream programming languages [9, 14]. However, existing work has not extended this capability to Dafny on these platforms. Furthermore, no Dafny benchmark currently matches the scale or systematic rigor of these widely used datasets.

Dafny is particularly well-suited for benchmark generation due to its dual role as both a verification-oriented specification language and a compilable programming language. Once verified, Dafny programs can be compiled into languages such as C#, Java, or Go, enabling us to simultaneously validate functional correctness and ensure alignment between implementations and formal postconditions by running them on a few test cases. We now describe our pipeline for transforming LeetCode tasks into verifiable Dafny code.

Synthesizing Dafny program from natural language. As shown in Fig. 1, the first step of TACoDafny is to synthesize a correct Dafny program from a natural language description. We begin by collecting programming tasks from LeetCode along with their natural language descriptions and a reference implementation in a mainstream language (we adopt Python in our pipeline). Next, we ❶ prompt the LLM to implement the same code in Dafny, retrying as needed until a syntactically valid program is obtained. At this stage, we do not generate verification code, focusing instead on functional correctness. To check this, we ❷ ask the LLM to create a suite of test inputs with positive input-output pairs (i_p, o_p), ensuring that relevant edge cases are included. Then,

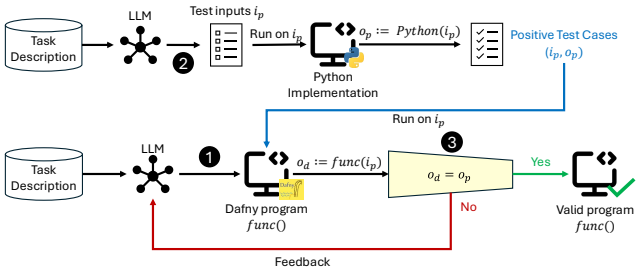

Figure 1. Pipeline for generating a functional correct Dafny program from natural language.

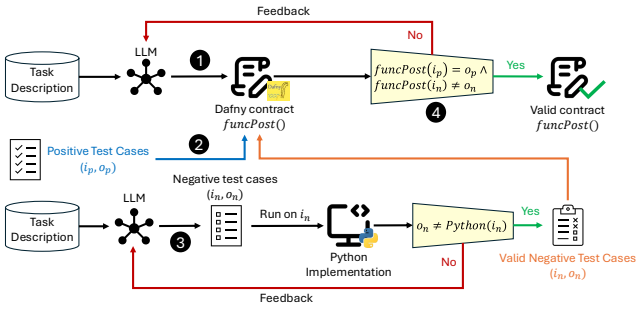

Figure 2. Pipeline to generate postcondition specifications.

both the Python and Dafny implementations are executed against these test cases, and we ❸ compare their outputs. For each task, we iteratively provide the LLM with failing test cases until the Dafny program consistently matches the reference Python implementation.

Synthesizing accurate postcondition specification. Once TACoDafny generates functionally correct Dafny implementations, we derive their formal specifications, focusing in particular on postconditions, since these capture the intended program behavior. Integrating the postconditions into the code yields a complete verification task. Fig. 2 shows our pipeline to generate postcondition specifications. We ❶ prompt the LLM to define the intended behavior as a single *non-ghost* predicate $funcPost(i, o)$, and to insert the corresponding predicate call into the method's postcondition. We then ❷ reuse the previous positive input-output pairs (i_p, o_p) to test the soundness of the candidate specification. In addition, we ❸ prompt the LLM to create negative examples (i_n, o_n), where o_n differs from the actual program output, which we automatically validate by executing the Python implementation. Since non-ghost predicates in Dafny are executable, we ❹ evaluate $funcPost(i, o)$ over all positive and negative examples, requiring that $funcPost(i_p, o_p) = \texttt{true}$ and $funcPost(i_n, o_n) = \texttt{false}$. Whenever a candidate specification fails any of these checks, we refine it through iterative feedback until a consistent and sound behavior specification is obtained.

Our final benchmark consists of tasks that combine an implementation with its corresponding postcondition. Note that in this work, we do not address the problem of providing full verification proofs for these postconditions; instead, we focus on generating specifications that are both functionally correct and verifiable.

Assessing postcondition specification strength. Our generation pipeline ensures that each synthesized program is coupled with a behavior specification that is executable and verifiable against a set of tests. However, beyond mere verifiability, it is crucial to assess the strength of the generated specification, *i.e.,* the extent to which it accurately constrains the intended program behavior. Intuitively, a desirable specification should be sufficiently strong such that, for

a given input i and a bounded search space, the corresponding output $func(i)$ is the unique solution that satisfies the postcondition $funcPost(i, o)$. If the specification is too weak (*e.g.,* $funcPost(i, o) = \texttt{true}$), then every output in the search space becomes admissible. Instead, a restrictive specification may eliminate all potential solutions. The challenge lies in formally characterizing this search space and determining a threshold to decide the strictness of a specification definition.

For TACoDafny, we rely on a heuristic-based assessment approach that works as follows. First, we prompt Claude 4 Sonnet to synthesize a definition of a finite search domain, denoted as $finiteSet(i)$, which bounds the set of candidate outputs for a given input i. Using this domain, we perform model counting to compute the cardinality of:

$$\{\, o \mid o \in \mathit{finiteSet}(i) \land funcPost(i, o) \,\}.$$

If the model count equals 1, we consider the specification as sufficiently strong within the defined search space.

Nevertheless, this method is subject to several limitations. A trivially small $finiteSet(i)$ may exclude the valid solution, leading to a model count of 0 even when a valid solution is known to exist based on previous unit testing steps. Conversely, an overly large search domain can result in model counting timeouts due to combinatorial explosion. Optimizing the generation of the exploration space is therefore a critical direction for future work. Additionally, some programming tasks inherently permit multiple valid solutions (*e.g.,* lists with interchangeable element orderings), where the model count may be greater than 1 yet still significantly smaller than the overall search space. Establishing quantitative thresholds to distinguish sufficiently strong specifications in such cases remains to be refined.

Preventing data contamination. TACoDafny's generation process is fully automated from a set of natural-language programming tasks description. We mitigate contamination in two ways: (i) we release only the code to generate the benchmark, instead of a static snapshot of instances created at evaluation time; and (ii) because the pipeline is fully automated, TACoDafny can be live-updated from newly released programming tasks, allowing evaluators to include only tasks published after a model's training cutoff.

4 Experimental Setup

Benchmark generation. We build TACoDafny using a publicly available LeetCode dataset from HuggingFace [7]. For program synthesis, we employ Claude 4 Sonnet to generate corresponding Dafny programs, as prior work has shown that Anthropic models demonstrate strong capabilities in Dafny code synthesis [15, 17]. We generate test cases using GPT-5, leveraging its advanced reasoning ability to ensure diverse and logically consistent inputs and outputs.

Running TACoDafny. The objective for each LLM is to produce Dafny code with sufficient specifications and proofs such that it successfully passes the Dafny verification process. We conduct a preliminary experiment evaluating model performance using the metric *verify@k*, where k denotes the maximum number of interaction rounds between the user and the model. A task is considered solved within k steps if the generated code verifies without errors after no more than k iterative refinement attempts.

It is worth noting that after each unsuccessful verification attempt, Dafny returns a detailed error message. We implement a feedback-driven interaction loop by appending the verifier feedback to the conversation history, enabling the LLM to iteratively refine the program until verification succeeds or the interaction budget is exhausted.

We evaluate TACoDafny on three proprietary models: Claude 4 Sonnet, GPT-5, and GPT-4o, and three open-weight models of different sizes: Llama3.1 8B, Qwen3-Coder 30B, and Llama3.3 70B.

5 Preliminary Results

5.1 Dataset

As an initial evaluation, we applied our benchmark generation pipeline to the first 200 LeetCode problems. This process yielded **87** Dafny programs that were executable and successfully passed all auto-generated test units. Among these, Claude 4 Sonnet was able to produce **76** executable, non-ghost formal specifications that function as valid postconditions within the verification process. All generated specifications satisfied their corresponding test units, indicating their basic correctness.

To further assess the strength of the specification, we conducted model counting checks on specific input cases. We found that **51** specifications imposed sufficiently strong constraints. The remaining specifications often led to the timeout problem we mentioned in Sec. 3, primarily due to the large exploration space introduced with a broad definition of $finiteSet()$.

5.2 Models Performance on TACoDafny

Claude 4 Sonnet achieves the highest overall performance. As shown in Fig. 3, Claude 4 Sonnet achieves a *verify@20* score of 39.5%, significantly outperforming all other models, including GPT-5. The GPT-5 series reaches a moderate score of 22%. Among open-weight models, Qwen3-Coder 30B performs the best, achieving a level comparable to the much larger Llama3.3 70B. In contrast, smaller models such as Llama3.1 8B consistently fail to solve any verification tasks, indicating the lack of Dafny code generation and reasoning capabilities.

Iterative feedback leads to steady performance gains. All models exhibit an increasing trend as k increases, confirming that iterative feedback in the form of error messages and verification failures helps models incrementally refine their code. For instance, Claude 4 Sonnet improves from

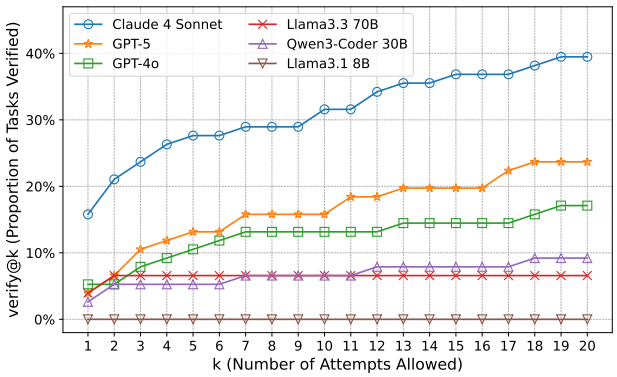

Figure 3. Model performance improves as more attempts are allowed for iterative correction of verification code.

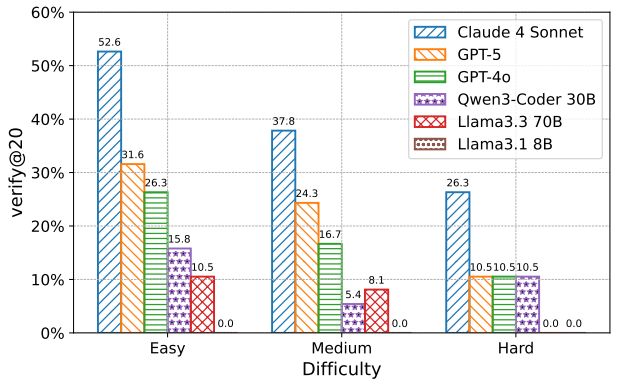

Figure 4. Tasks that are easier to implement are often easier to verify.

15.8% at $k = 1$ to 39.5% at $k = 20$. This increment demonstrates its ability to interpret feedback and systematically strengthen its verification logic.

Small models struggle to adhere to formal verification constraints. We conducted an examination of model traces of small models, and we noticed that they frequently resort to invalid shortcuts when facing complex verification tasks. Common failures include: (*i*) inserting placeholder statements, (*ii*) adding unverifiable assumptions to bypass proof obligations, or (*iii*) weakening or omitting the postcondition altogether. These behaviors indicate a lack of robustness in reasoning and a limited understanding of Dafny's formal verification semantics.

Programs that are easy to implement tend to be easy to prove. Fig. 4 presents *verify@20* performance grouped by LeetCode difficulty levels. All models show a consistent decline in success rates as the task difficulty increases. This suggests that problems requiring more complex implementations also tend to induce harder verification challenges, likely due to increased control-flow complexity, deeper invariants, and more intricate postconditions.

6 Discussion and Future Work

How do we ensure that all benchmark tasks are verifiable? Currently, TACoDafny does not guarantee that every task is fully verifiable; rather, we ensure that all programs pass a set of auto-generated unit tests. As with real-world software development, untested edge cases may still exist, potentially leaving some implementations or specifications unverifiable. Nonetheless, the majority of tasks are verifiable in practice, and different models demonstrate competitive performance across the benchmark, fulfilling our goal of enabling meaningful comparative evaluation.

Scaling to real-world scenarios. Our current experiments focus on beginner- or learner-level programming and verification tasks, constrained by the difficulty level of the programming platforms we explore. As a next step, we aim to extend our pipeline toward more complex industrial-grade codebases, enabling the evaluation of scalability and practical applicability in real-world verification settings.

Extending to ghost predicates for postconditions. We plan to extend our approach to generate ghost-based postconditions that can also be validated using test units. This requires establishing a provable equivalence between ghost definitions and corresponding executable code, where the latter can be used for testing purposes.

Developing a more rigorous evaluation methodology. Our current method for assessing the quality of generated formal specifications remains preliminary and unreliable. We aim to design a more principled evaluation framework, including a clearer definition of the exploration space, to reduce the likelihood of timeouts, and a quantitative threshold to decide the quality of the postcondition definition.

Beyond LLM evaluation. The automated pipeline produces scalable datasets of unverified but functionally correct Dafny programs, and executing cutting-edge models on the benchmark generates detailed traces of attempts and verification obligations. As Fig. 3 shows, small/medium models struggle on Dafny verification; all the dataset and traces in the generation and evaluation process can be used to train or fine-tune Dafny-specialized models to improve code generation and postcondition verification, enabling knowledge transfer from strong models to smaller ones.

7 Conclusions

In this paper, we introduced TACoDafny, a novel benchmark for evaluating LLMs performance in Dafny code generation. TACoDafny addresses the main limitations of previous benchmarks by offering a fully automated generation pipeline and contamination-free tasks, releasing only the scripts needed to produce them from natural language specifications. We assessed several LLMs on TACoDafny and observed that their ability to generate verifiable code declines as task difficulty increases. We also released an online

leaderboard for TACoDafny, enabling transparent tracking and comparison of LLM performance within the community. For future work, we plan to extend TACoDafny with more real-world tasks drawn from complex codebases and to refine our methodology for generating and evaluating the postcondition specifications.

References

[1] Amazon. 2025. Amazon Q Developer. https://aws.amazon.com/q/developer/

[2] AtCoder. 2025. AtCoder. https://github.com/atcoder/ac-library

[3] Saikat Chakraborty, Shuvendu K. Lahiri, Sarah Fakhoury, Madanlal Musuvathi, Akash Lal, Aseem Rastogi, Aditya Senthilnathan, Rahul Sharma, and Nikhil Swamy. 2024. Ranking LLM-Generated Loop Invariants for Program Verification. arXiv:2310.09342 [cs.PL] https://arxiv.org/abs/2310.09342

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374

[5] Codeforces. 2025. Codeforces. https://codeforces.com/

[6] Cursor. 2025. The AI Code Editor. https://cursor.com/

[7] Hugging Face. 2025. LeetCode repository on Hugging Face. https://huggingface.co/datasets/greengerong/leetcode

[8] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring Mathematical Problem Solving With the MATH Dataset. arXiv:2103.03874 [cs.LG] https://arxiv.org/abs/2103.03874

[9] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. *arXiv preprint arXiv:2403.07974* (2024).

[10] Andrei Kozyrev, Gleb Solovev, Nikita Khramov, and Anton Podkopaev. 2024. CoqPilot, a plugin for LLM-based generation of proofs. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 2382–2385. doi:10.1145/3691620.3695357

[11] Leetcode. 2025. LeetCode. https://leetcode.com/

[12] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*. 348–370. doi:10.1007/978-3-642-17511-4_20

[13] 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 J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097. arXiv:https://www.science.org/doi/pdf/10.1126/science.abq1158 doi:10.1126/science.abq1158

[14] LLM Stats. 2025. HumanEval Leaderboard. https://llm-stats.com/benchmarks/humaneval

[15] Chloe Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. 2024. DafnyBench: A Benchmark for Formal Software Verification. arXiv:2406.08467 [cs.SE] https://arxiv.org/abs/2406.08467

[16] Microsoft. 2024. Microsoft Copilot. https://copilot.microsoft.com

[17] Md Rakib Hossain Misu, Cristina V. Lopes, Iris Ma, and James Noble. 2024. Towards AI-Assisted Synthesis of Verified Dafny Methods. *Proceedings of the ACM on Software Engineering* 1, FSE (July 2024), 812–835. doi:10.1145/3643763

[18] Eric Mugnier, Emmanuel Anaya Gonzalez, Ranjit Jhala, Nadia Polikarpova, and Yuanyuan Zhou. 2025. Laurel: Unblocking Automated Verification with Large Language Models. arXiv:2405.16792 [cs.LO] https://arxiv.org/abs/2405.16792

[19] Manley Roberts, Himanshu Thakur, Christine Herlihy, Colin White, and Samuel Dooley. 2024. To the Cutoff... and Beyond? A Longitudinal Perspective on LLM Data Contamination. In *International Conference on Representation Learning*, B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (Eds.), Vol. 2024. 10795–10862. https://proceedings.iclr.cc/paper_files/paper/2024/file/2d04d97593c8c33d415337f408ed0e1b-Paper-Conference.pdf

[20] Oscar Sainz, Jon Campos, Iker García-Ferrero, Julen Etxaniz, Oier Lopez de Lacalle, and Eneko Agirre. 2023. NLP Evaluation in trouble: On the Need to Measure LLM Data Contamination for each Benchmark. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 10776–10787. doi:10.18653/v1/2023.findings-emnlp.722

[21] Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. 2024. Clover: Closed-Loop Verifiable Code Generation. arXiv:2310.17807 [cs.AI] https://arxiv.org/abs/2310.17807

[22] Changjie Wang, Mariano Scazzariello, Alireza Farshin, Simone Ferlin, Dejan Kostić, and Marco Chiesa. 2024. NetConfEval: Can LLMs Facilitate Network Configuration? *Proc. ACM Netw.* 2, CoNEXT2, Article 7 (June 2024), 25 pages. doi:10.1145/3656296

[23] Shenao Yan, Shen Wang, Yue Duan, Hanbin Hong, Kiho Lee, Doowon Kim, and Yuan Hong. 2024. An LLM-Assisted Easy-to-Trigger Backdoor Attack on Code Completion Models: Injecting Disguised Vulnerabilities against Strong Detection. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 1795–1812. https://www.usenix.org/conference/usenixsecurity24/presentation/yan

[24] Quanjun Zhang, Weifeng Sun, Chunrong Fang, Bowen Yu, Hongyan Li, Meng Yan, Jianyi Zhou, and Zhenyu Chen. 2025. Exploring Automated Assertion Generation via Large Language Models. *ACM Trans. Softw. Eng. Methodol.* 34, 3, Article 81 (Feb. 2025), 25 pages. doi:10.1145/3699598

[25] 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).