DafnyProver: Agentic Verified Code Generation in Dafny

Benjamin Breen [1] **Austin Letson** [1] **Borja Requena Pozo** [1] **Leopoldo Sarra** [1]

Abstract

We study agentic code generation in Dafny, where a model must generate both executable code and the proof artifacts for verification. We present AxDafny, a verifier-guided repair framework that iteratively generates implementations, invariants, assertions, and termination arguments. We also introduce LiveCodeBench-Pro-Dafny (LCB-Pro-Dafny), a benchmark of 250 competition-style programming problems translated into Dafny with formal specifications and a verifier-based evaluation harness. On LCB-Pro-Dafny, AxDafny substantially improves verification success over baseline GPT-5.5 performance. On DafnyBench, AxDafny verifies 725/782 instances (92.7%), outperforming the strongest previously reported proof-hint baseline by 6.5 percentage points. Lastly, we show that verification success and runtime test performance measure different aspects of generated code.

1. Introduction

Inference scaling for coding agents often takes the form of an iterative generate–check–repair loop: a model proposes code, receives feedback from an environment, and then refines the code based on the feedback (Yang et al., 2024; Requena et al., 2026). Formal methods provide a rich feedback signal: rather than checking a fixed set of test cases, a verifier can identify when a program violates its specification, when additional proof annotations are needed to verify logical steps, and when edge cases are not handled. This makes verified code generation a practical setting for studying the effects of inference scaling in agentic coding. Despite this promise, applications of formal methods in mainstream AI tasks remain limited due to the substantial expertise they require.

We study how verification feedback can improve agentic code generation, using Dafny as an experimental testbed because it provides built-in support for formal verification. We make two main contributions. First, we introduce AxDafny, an agentic framework for program and proof synthesis that generates verified implementations through iterative verifier-guided repair. Second, we introduce LiveCodeBench-Pro-Dafny (LCB-Pro-Dafny), a benchmark that translates competition-style coding problems into Dafny to evaluate both program synthesis and proof synthesis.

Empirically, AxDafny with Gemini-3.1-Pro verifies 92.7% on the established DafnyBench proof-hint task, outperforming the strongest previously reported baseline by 6.5 percentage points. On the newly introduced LCB-Pro-Dafny benchmark, AxDafny verifies 56.4% of tasks, improving over the 11.6% success rate of direct GPT-5.5 generation. A secondary evaluation compiles verified LCB-Pro-Dafny solutions to Python and executes them under the original LiveCodeBench-Pro test harness. We observe that most executable failures result from resource limits, because the Dafny specifications enforce functional correctness rather than asymptotic complexity. Together, these results demonstrate the efficacy of verifier-guided repair for functional verification, while exposing the remaining challenge of generating verified code that meets competitive-programming resource constraints.

1.1. Dafny and Formal Verification

Dafny is a programming language with built-in support for formal verification (Leino, 2010). A Dafny method can state assumptions about its inputs using `requires` and guarantees about its outputs using `ensures`; these are known as preconditions and postconditions, respectively. Dafny checks that the implementations satisfy these specifications using an SMT-based verifier, reporting failed proof obligations with source locations and diagnostic messages.

Verification in Dafny often requires additional annotations beyond the executable code (Figure 1). Loop invariants specify conditions that remain true across every iteration of a `for` or `while` loop; for example, an invariant in a sorting loop might state that the processed prefix is sorted. Assertions ask Dafny to prove a local fact at a specific point in a program; for example, `assert a[j] <= a[j+1]`

[1]Axiomatic AI, Boston, MA, USA. Correspondence to: Benjamin Breen <ben@axiomatic-ai.com>.

The 3rd *AI for Math Workshop* at the 43rd International Conference on Machine Learning (ICML), Seoul, South Korea, 2026.
Copyright 2026 by the author(s).

```
method Sum(n: int) returns (s: int)
  requires n >= 0
  ensures s == n * (n + 1) / 2
{
  var i := 0;
  s := 0;

  while i < n
    invariant 0 <= i <= n
    invariant s == i * (i + 1) / 2
  {
    i := i + 1;
    s := s + i;
  }

  assert i == n;
}
```

Figure 1. A verified Dafny method for computing an arithmetic sum from 0 to n. Executable statements, highlighted in blue, implement the main loop. Method specifications, highlighted in green, use `requires` and `ensures` to specify input assumptions and output guarantees. Proof annotations, highlighted in orange, use `invariant` and `assert` to supply intermediate facts to the verifier in order to prove the postcondition.

can expose an ordering fact needed to preserve the loop invariant or prove the final postcondition. Finally, termination arguments show that recursive calls or loops make progress.

Dafny provides two additional properties useful for code generation. First, it can produce concrete counterexamples for certain failed verification conditions. Second, verified programs can be compiled to standard programming languages such as Python, Java, and C++ for execution.

These properties make Dafny a strong setting to study verifier-guided LLM code generation: verifier failures identify both program errors and missing proof obligations, while successful outputs remain executable. Unlike ordinary code generation, Dafny generation requires the model to produce both the implementation and the proof structure needed for verification. We therefore distinguish *program synthesis*, which generates executable Dafny code, from *proof synthesis*, which generates invariants, assertions, and other annotations needed to establish verification.

2. Related Work

2.1. Agentic Architectures for Dafny

Several recent Dafny systems study LLM-assisted generation with verifier feedback. DafnyBench evaluates LLM proof-hint generation with a basic iterative loop: the model proposes annotations, Dafny checks the file, and verifier errors are returned for repair (Loughridge et al., 2024). DafnyPro improves this setting by editing existing files instead of regenerating whole proofs, pruning unused invariants, and retrieving from a manually curated Dafny proof-hint library (Banerjee et al., 2026).

These systems target proof generation, whereas our work addresses the joint synthesis of programs and proofs. Moreover, our method does not rely on proof-hint retrieval to improve proof-synthesis performance.

2.2. Dafny Benchmarks

Most public Dafny benchmarks target proof synthesis rather than program synthesis. DafnyBench removes `assert` and `invariant` annotations from existing verified programs and evaluates whether a model can restore sufficient proof hints for verification (Loughridge et al., 2024). DafnyComp evaluates generation of specifications and proof annotations across multiple Dafny programs (Xu et al., 2025). VeriEquivBench evaluates two related proof tasks: refining proof annotations for a fixed specification, and generating the strongest specification for a fixed implementation (Zeng et al., 2026).

Public benchmarks for Dafny program synthesis are less common. Dafny-synthesis evaluates verified Dafny method generation from natural-language prompts (Misu et al., 2024); its 178 tasks are derived from the Mostly Basic Python Problems (MBPP) benchmark (Austin et al., 2021). HumanEval-Dafny translates 129 tasks from the original HumanEval benchmark into Dafny (Chen et al., 2021; JetBrains Research, 2024). These benchmarks are limited for comparisons with mainstream programming languages by their scale, simplicity, potential data contamination, and lack of runtime constraints. We therefore introduce LCB-Pro-Dafny, a larger competition-style benchmark for verifier-guided Dafny program synthesis.

3. Methods

3.1. Agent Architecture

For AxDafny, we adapt the agentic architecture proposed in Requena et al. (2026) used for Lean proving to Dafny code synthesis and verification. It consists of an iterative refinement loop with a proposer agent, a review system, and a memory module. The proposer writes Dafny code, the review system checks it and returns concrete feedback, and the memory module preserves information across iterations.

Proposer. The proposer's goal is to produce a Dafny file that passes `dafny verify`, without modifying the code and `ensures` statements that define the problem. It receives the original file content, the name of the primary target, and any context from previous attempts maintained by the memory module. With this information it returns the complete annotated file.

Reviewer system. The reviewer has two stages: deterministic checks and LLM-based review. The deterministic stage consists of two static checks on the proposed file, followed by verifier diagnostics from Dafny.

First, we extract the `requires` and `ensures` clauses from the original and proposed files, and require the original clauses to be a subset of the proposed clauses. This prevents specification weakening while still allowing new specifications for helper methods and lemmas.

Second, a regex filter rejects proof-bypass constructs: `{:axiom}` and `{:verify false}` for specifications, `assume` for asserting verification facts without proof, and `{:extern}` for declarations whose implementations are assumed to be verified externally. If either check fails, the offending lines are returned to the proposer.

We then run `dafny verify` on the generated file. When verification fails, diagnostics identify the failed proof obligations and their source locations, including unmet preconditions, postconditions, invariants, or termination checks. When enabled, the `--counterexamples` flag can also provide concrete counterexamples for invalid verification conditions. These verification messages are fed back to the proposer as repair feedback.

Finally, a reviewer LLM checks the file for cheating patterns which aren't covered by the deterministic checks, for example rewriting a predicate to be trivially true. A candidate is accepted only if it passes the preservation checks, contains no rejected proof-bypass constructs, verifies with Dafny, and is not rejected by the LLM reviewer.

Context management. We use a self-managed, reflection memory variant as in (Requena et al., 2026), adapted to Dafny verifier output. On every iteration the proposer receives two forms of historical context. First, the immediately previous attempt is included verbatim: the proposer's prior reasoning, the proposed file, and the feedback returned by either `dafny verify` or the reviewer agent. Second, a separate scratchpad call maintains a rolling memory that condenses lessons accumulated across all earlier attempts. After each iteration, this memory is regenerated by an LLM given the current attempt's reasoning, code, and feedback together with the previous notebook contents, with explicit instructions to preserve earlier lessons while folding in new ones.

3.2. Agent Configuration

Experiments use either Gemini-3.1-Pro or GPT-5.5 as the base language model. GPT-5.5 is evaluated with either the low or medium reasoning setting. In small-scale ablations, Gemini-3.1-Pro and GPT-5.5 provided the best performance-cost tradeoffs among the models tested. For the full evaluation, AxDafny was allowed a budget of 20 iterative attempts per benchmark instance.

3.3. Benchmarks

We evaluate AxDafny on two complementary verified-code tasks. DafnyBench measures proof-hint synthesis: the implementation and specification are given, and the agent must restore missing invariants and assertions so that Dafny verifies the file (Loughridge et al., 2024). LCB-Pro-Dafny, introduced in this work, measures program synthesis: the agent receives both a natural-language problem description and Dafny specification, and must generate the implementation and proof artifacts needed for verification.

LCB-Pro-Dafny Construction LCB-Pro-Dafny contains 250 LiveCodeBench-Pro (Zheng et al., 2025) competition problems translated into Dafny: 100 easy, 100 medium, and 50 hard instances. Each instance contains the original natural-language problem statement, a Dafny method signature, and a formal specification. The benchmark is released with an evaluation harness; acceptance criteria are defined in Section 3.4.

Benchmark validation. Specification quality is a known bottleneck for verifiable-code benchmarks (Zeng et al., 2026). We therefore treat LCB-Pro-Dafny construction as a curation process rather than a fully automatic translation step. Each specification was checked by LLM-as-judge review against the original problem statement, verifier-based sanity checks, and manual review of flagged cases. All 250 specifications parse and type-resolve under Dafny 4.11.0. LLM-as-judge review flagged 14/250 specifications (5.6%) for semantic fixes; all flagged specifications were revised before inclusion. Appendix A reports the validation checks and counts.

3.4. Metrics

For proof synthesis on DafnyBench, we follow the benchmark's original evaluation protocol (Loughridge et al., 2024). A solution is accepted only when it verifies in Dafny and does not alter the `requires` or `ensures` clauses or rely on vacuous proofs such as `assume false` or `{:verify false}`.

For program synthesis on LCB-Pro-Dafny, we use an analogous evaluation protocol. A solution is accepted only when (1) the generated implementation verifies in Dafny, (2) the provided specification, definitions, and functions are not modified, and (3) the solution does not rely on vacuous proof shortcuts, including `assume` or `{:extern}`.

Additionally, we report a secondary functional-correctness metric using the original LCB-Pro test cases (Zheng et al., 2025). For this evaluation, the generated Dafny code is

compiled to Python and executed against the corresponding test suite. This metric is separate from verifier acceptance and measures executable correctness under the original competitive-programming harness.

3.5. Baselines

For DafnyBench, we compare with pass@1 model results and published proof-hint results using the original evaluation protocol, including DafnyBench, dafny-annotator, and DafnyPro (Loughridge et al., 2024; Poesia et al., 2024; Banerjee et al., 2026). These baselines provide context for the current frontier on the established proof synthesis task.

For LCB-Pro-Dafny, our primary baseline is GPT-5.5 pass@1. The model is given the same problem statement and Dafny specification as AxDafny, but is evaluated using a single non-iterative generation attempt. This baseline measures the verification success of direct model output without agentic verifier-guided repair.

As context for cross-language baselines, Appendix B reports standard LCB-Pro leaderboard results under the original Python test-based evaluation. These results are not directly comparable to LCB-Pro-Dafny verification rates, but they contextualize the difficulty of the programming tasks under the original runtime-test criterion.

4. Results

We report verification success on DafnyBench and LCB-Pro-Dafny. The LCB-Pro-Dafny results are stratified by difficulty to separate performance on easy, medium, and hard instances.

On DafnyBench, Table 1 shows that AxDafny with Gemini-3.1-Pro verifies 725 / 782 benchmark instances (92.7%), improving over the strongest previously reported proof-hint baseline by 6.5 percentage points. The comparison with DafnyPro requires some additional context: DafnyPro reports 86.2% for its full system, which includes a retrieval component from a manually curated proof-hint library. Its ablation attributes 9.97 percentage points of DafnyBench performance to hint augmentation; without this component, DafnyPro reports 76.2% success. AxDafny does not use a task-specific proof-hint library.

While Table 1 reports the standard evaluation setting for the DafnyBench proof-hint task, we also test a more permissive setting in which models may introduce helper lemmas and propositions in addition to proof annotations. Under this evaluation setting, AxDafny with GPT-5.5 medium verifies 92.0% of DafnyBench instances.

Figure 2 shows that verifier-feedback gains on DafnyBench are concentrated in the early iterations. The Gemini-3.1-Pro setting achieves the highest verification success, reaching

Table 1. DafnyBench proof-hint results.

Method	Verified (%)
AxDafny (Gemini-3.1-Pro)	**92.7**
AxDafny (GPT-5.5 medium)	**88.9**
DafnyPro (Banerjee et al., 2026)	86.2
AxDafny (GPT-5.5 low)	**85.3**
DafnyPro w/o hints	76.2
Gemini-3.1-Pro pass@1	68.5
DafnyBench (Loughridge et al., 2024)	68.0
dafny-annotator (Poesia et al., 2024)	50.6
GPT-5.5 (medium) pass@1	54.6
GPT-5.5 (low) pass@1	54.1

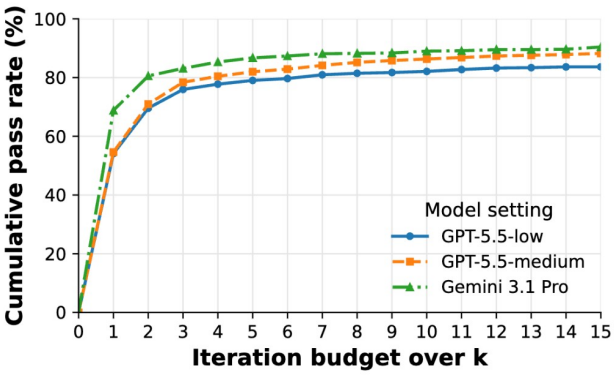

Figure 2. Cumulative AxDafny pass rate on DafnyBench vs. iteration budget across model settings. The primary feedback gains are heavily concentrated within the first 5 iterations.

Table 2. Results on LCB-Pro-Dafny by difficulty split. Each entry is the verification success rate (%).

Method	Easy n=100	Medium n=100	Hard n=50	Overall n=250
AxDafny	**75.0%**	**52.0%**	**28.0%**	**56.4%**
GPT-5.5 pass@1	13.0%	14.0%	4.0%	11.6%

92.7% (725/782), compared to the 88.9% (695/782) of GPT-5.5 medium and 85.3% of GPT-5.5 low. At a fixed budget of 15 iterations, GPT-5.5 medium exceeds GPT-5.5 low by 3.6 percentage points, while Gemini Pro exceeds GPT-5.5 low by 7.4 percentage points; by iteration 5, Gemini Pro already surpasses the final GPT-5.5 low pass rate. The models demonstrated gains between 25% and 33% over the course of the 15 iterations.

Table 2 and Figure 3 show that AxDafny substantially improves over the GPT-5.5 pass@1 baseline on LCB-Pro-Dafny. However, the hard split remains far from saturated: AxDafny verifies only 28.0% of hard instances. This confirms that LCB-Pro-Dafny is a challenging benchmark for verified program synthesis.

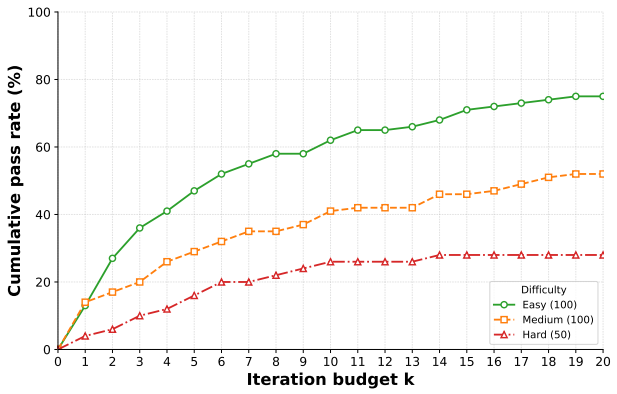

Figure 3. Cumulative AxDafny pass rate on LCB-Pro-Dafny across difficulty splits. Accuracy scales steadily with an increased iteration budget across all levels. Compared to proof-hint filling on DafnyBench, end-to-end program synthesis derives more sustained benefits from more iterations.

Executable correctness. We also compile verified Dafny outputs to Python and evaluate them with the original LCB-Pro test harness. This secondary evaluation is only defined for verified solutions: unverified Dafny files do not build. On the easy split, AxDafny verifies 75/100 tasks; among these verified outputs, 32 pass the original executable tests, 39 fail by time limit exceeded (TLE), and 4 fail by memory limit exceeded (MLE). On the medium split, AxDafny verifies 52/100 tasks; among these verified outputs, 6 pass the executable tests, 44 fail by TLE, and 2 fail by MLE. For context, Appendix B reports standard LCB-Pro leaderboard results under the original executable-code evaluation.

5. Discussion

The verification results show two complementary outcomes. On DafnyBench, AxDafny verifies 92.7%, improving proof-hint synthesis over published baselines and indicating that verifier-guided repair is effective for generating proof annotations when the implementation and specification are fixed. On LCB-Pro-Dafny, AxDafny improves substantially over direct GPT-5.5 pass@1 generation, but the hard split remains far from saturated. This suggests that LCB-Pro-Dafny is a challenging testbed for end-to-end verified program synthesis and a useful benchmark for future work on verifier-guided coding agents.

The secondary executable evaluation measures a different property: whether verified Dafny programs also satisfy the runtime and input-output constraints of the original LCB-Pro test harness. Among verified outputs that fail this secondary evaluation, the observed failures are resource-limit failures (TLE or MLE), rather than wrong-answer failures or failures to compile unverified code. These failures have two main sources. First, the Dafny specifications constrain functional correctness but generally do not constrain asymptotic complexity or memory usage. Agents can therefore produce implementations that are easier to verify but inefficient at runtime. Second, Dafny compilation to Python relies on runtime implementations for language constructs such as sets and maps, which can be slower than native Python or C++ implementations. We partially mitigate this second source with a lightweight Python compatibility layer for selected Dafny runtime constructs.

These results show that verifier acceptance and competitive-programming performance remain distinct objectives. A second limitation is the relative scarcity of Dafny training data compared with mainstream programming languages. As a rough proxy for this gap, GitHub code search returns 3.5×10^7 Python files and 1.5×10^4 Dafny files,[1] a difference of more than three orders of magnitude.

Software and Data

We release the code, benchmark data, and evaluation harness in a public GitHub repository. For LCB-Pro-Dafny, we release the problem statements, Dafny specifications, and evaluation harness. The harness checks Dafny verification, compiles verified Dafny solutions to Python, and evaluates the resulting programs against the LiveCodeBench-Pro test harness (Zheng, 2026) under fixed time limits. We also provide a lightweight Python compatibility layer for selected Dafny runtime constructs to reduce Time Limit Exceeded (TLE) failures during evaluation.

Acknowledgments

We thank Krystian Nowakowski and Matteo Cipollina for helpful discussions and feedback during the development of this work.

Impact Statement

Formal verification can make everyday coding tasks more reliable by checking program behavior against explicit specifications rather than only against finite test suites. If tools for verified code generation become more usable, developers could apply stronger correctness checks to routine library functions, data-processing code, and infrastructure components where subtle edge cases are difficult to test exhaustively.

At the same time, verified code generation can create misplaced confidence when specifications are incomplete, in-

[1] GitHub code search: https://github.com/search?type=code&q=language:Python and https://github.com/search?type=code&q=language:Dafny, accessed 2026-05-07.

correct, or misaligned with user intent. Our evaluation therefore treats the specification as a trusted object and explicitly checks for specification modification and vacuous proof shortcuts. These safeguards are necessary for using formal verification in practical coding workflows.

References

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., and Sutton, C. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Banerjee, D., Bouissou, O., and Zetzsche, S. Dafnypro: Llm-assisted automated verification for dafny programs, 2026.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

JetBrains Research. Humaneval-dafny: Translating humaneval to dafny. https://github.com/JetBrains-Research/HumanEval-Dafny, 2024.

Leino, K. R. M. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pp. 348–370. Springer, 2010.

LiveCodeBench Pro. Livecodebench pro live leaderboard. https://livecodebenchpro.com/projects/livecodebench-pro/leaderboard, 2026. Accessed 2026-05-23.

Loughridge, C., Sun, Q., Ahrenbach, S., Cassano, F., Sun, C., Sheng, Y., Mudide, A., Misu, M. R. H., Amin, N., and Tegmark, M. Dafnybench: A benchmark for formal software verification. *arXiv preprint arXiv:2406.08467*, 2024.

Misu, M. R. H., Lopes, C. V., Ma, I., and Noble, J. Towards ai-assisted synthesis of verified dafny methods. *Proc. ACM Softw. Eng.*, 1(FSE), 2024. doi: 10.1145/3643763. URL https://doi.org/10.1145/3643763.

Poesia, G., Loughridge, C., and Amin, N. dafny-annotator: Ai-assisted verification of dafny programs, 2024.

Requena, B., Letson, A., Nowakowski, K., Beltran-Ferreiro, I., and Sarra, L. A minimal agent for automated theorem proving, 2026. URL https://arxiv.org/abs/2602.24273.

Xu, X., Li, X., Qu, X., Fu, J., and Yuan, B. Local success does not compose: Benchmarking large language models for compositional formal verification. *arXiv preprint arXiv:2509.23061*, 2025.

Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., and Press, O. SWE-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024.

Zeng, L., Che, F., Huang, X., Ye, F., Xu, X., Yuan, B., and Fu, J. Veriequivbench: An equivalence score for ground-truth-free evaluation of formally verifiable code. In *International Conference on Learning Representations*, 2026.

Zheng, G. Livecodebench-pro: Llm benchmarking toolkit. https://github.com/GavinZhengOI/LiveCodeBench-Pro, 2026.

Zheng, Z., Cheng, Z., Shen, Z., Zhou, S., Liu, K., He, H., Li, D., Wei, S., Hao, H., Yao, J., Sheng, P., Wang, Z., Chai, W., Korolova, A., Henderson, P., Arora, S., Viswanath, P., Shang, J., and Xie, S. Livecodebench pro: How do olympiad medalists judge llms in competitive programming? *arXiv preprint arXiv:2506.11928*, 2025.

Table 3. Published LCB-Pro live leaderboard results from the project website. Easy, medium, and hard report current live success rates; overall is computed by aggregating reported pass counts across the live difficulty splits rather than using the website's Elo rating (LiveCodeBench Pro, 2026).

Method	Easy	Medium	Hard	Overall
GPT-5.2-high	90.1	52.1	15.9	53.1
GPT-5-high	90.1	40.8	4.2	45.1
o4-mini-high	85.7	29.2	1.4	37.4
Qwen3 Next 80B A3B Thinking	69.0	7.0	0.0	25.4
Claude 4.5 Sonnet Thinking	53.5	0.0	0.0	17.8
DeepSeek R1	50.7	1.4	0.0	17.4

A. LCB-Pro-Dafny Construction

LCB-Pro-Dafny was constructed from LiveCodeBench-Pro competition problems (Zheng et al., 2025). We retained the original difficulty labels and selected 250 instances: 100 easy, 100 medium, and 50 hard. Each instance was converted into a Dafny task containing the natural-language problem statement, a Dafny method signature, and a formal specification.

The initial Dafny specifications were produced with model assistance and then manually reviewed. During review, we removed or revised tasks with inconsistent, underspecified, or vacuous specifications, and checked that the remaining specifications captured the intended functional behavior of the original problem. The evaluation harness treats these specifications as fixed: submitted solutions may add helper definitions and proof artifacts, but may not weaken or rewrite the provided method signature, definitions, or specifications.

The harness runs Dafny verification as the primary acceptance criterion and applies additional checks for proof-bypass constructs such as `assume`, {`:extern`}, and {`:verify false`}. For the secondary executable evaluation, verified Dafny solutions are compiled to Python and run against the corresponding LiveCodeBench-Pro tests under the original time limits.

Validation checks. We report the validation outcomes used during benchmark curation. All 250 released specifications parse and type-resolve with `dafny resolve --allow-warnings` under Dafny 4.11.0. Static checks found no missing `ensures` clauses and no trivially true postconditions. Under Dafny's stricter default warning policy, 49 specifications emitted quantifier-trigger warnings, but otherwise parsed and type-resolved without errors.

LLM-as-judge review against the original problem statements flagged 14/250 specifications (5.6%) as semantic mismatches requiring fixes. In one representative medium instance, an earlier specification draft represented a set of chosen objects as a sequence but did not forbid duplicate entries; we fixed this by requiring the sequence to be distinct. All flagged specifications were revised and reviewed before inclusion.

As an additional vacuity check, no placeholder implementation verified, and no automatically generated constant-or-empty-output mutation verified in the 234 cases where the mutation harness applied.

B. Standard LCB-Pro Results

Table 3 reports standard LCB-Pro leaderboard results under the benchmark's original executable-code evaluation. These numbers provide context for the difficulty of the underlying programming tasks, but they are not directly comparable to LCB-Pro-Dafny verification rates: standard LCB-Pro measures whether generated Python code passes runtime tests, while LCB-Pro-Dafny measures whether generated Dafny code verifies against formal specifications. The comparison helps interpret our secondary executable evaluation: strong coding models still struggle on the hard split under ordinary runtime testing, while LCB-Pro-Dafny shifts the evaluation target to producing machine-checked implementations.

C. Model Ablation

Figure 4 reports a small model ablation on the LCB-Pro-Dafny easy split. The results compare verified pass rates for the same verifier-guided synthesis setting across four frontier model settings.

The ablation suggests that performance on LCB-Pro-Dafny is sensitive to the choice of base model and reasoning setting.

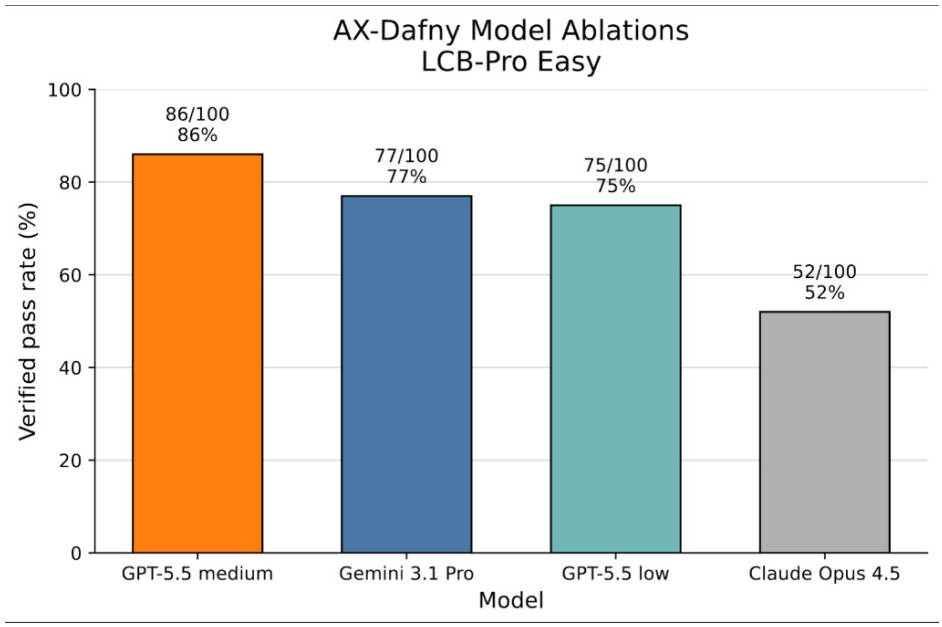

Figure 4. Model ablation on the LCB-Pro-Dafny easy split. Bars report verified pass rate over 100 easy instances.

On the easy split, GPT-5.5 medium verifies 86/100 instances, Gemini 3.1 Pro verifies 77/100, GPT-5.5 low verifies 75/100, and Claude Opus 4.5 verifies 52/100. We include this result to document model sensitivity for the verifier-guided synthesis loop; broader model comparisons require evaluation across all difficulty splits.