
Does Oracle Quality Matter? Adversarial Feedback for Formally Verified Code Synthesis¹

Prana
v
Kasett
y

Indep
enden
t
Resea
rcher

With
Apart Research

Abstract

Counterexample-Guided Inductive Synthesis (CEGIS) is the standard approach for repairing LLM-generated code that fails formal verification. We investigate whether augmenting the verifier's error feedback with adversarial LLM-generated counterexamples accelerates convergence. Using 50 Dafny tasks from the Vericoding benchmark, we find that adding a same-capability oracle (Sonnet) does not change pass rates but improves early convergence: pass@2 increases from 66% to 72%, with 9 tasks converging faster vs 6 slower among 42 jointly-solved tasks. The more surprising finding emerges when we upgrade to a stronger oracle (Opus): it unlocks a task that failed under all prior conditions by providing specific proof-repair guidance (identifying a missing lemma call) where the weaker oracle could only report "it times out." This suggests that for secure program synthesis pipelines, oracle

¹ Research conducted at the [The Secure Program Synthesis Hackathon](#), May 2026

quality, the ability to provide actionable proof-engineering feedback, matters more than oracle presence.

1. Introduction

Formal verification provides the strongest available guarantee that code meets its specification, particularly important for verified cryptographic libraries, smart contracts, and safety-critical systems. As LLMs are increasingly used to generate code, the ability to produce *formally verified* implementations becomes essential for secure program synthesis.

Regehr (2026) argues that LLM-generated code achieves "zero degrees of freedom" only when constrained by a formal verifier: the output must pass a proof checker, leaving no room for subtle bugs. The Vericoding benchmark (Bursuc et al., 2025) operationalizes this by evaluating LLMs on 12,504 formal specifications. Even frontier models fail 33-40% of Dafny tasks on the first attempt. The standard recovery is a CEGIS loop: feed verifier errors back for repair, iterating up to a budget.

However, verifier errors are often cryptic: line numbers and failed assertions without explaining *why* the proof fails. Regehr's vision of multi-oracle systems suggests that additional feedback sources could help. We test this by adding an adversarial LLM oracle that generates concrete counterexamples alongside the verifier.

Contributions:

1. An experimental framework evaluating adversarial oracle feedback in CEGIS loops for formal verification, replicating the Vericoding methodology on 50 Dafny tasks.
2. Evidence that same-capability oracle feedback accelerates early convergence (+6% at pass@2) without changing pass rates.
3. A demonstration that oracle *quality* is the key variable: a stronger oracle provides specific proof-repair guidance that unlocks a previously unsolvable task.
4. A taxonomy of irreducible CEGIS failure modes based on the 7 tasks that resist all repair conditions.

2. Related Work

- (a) **Vericoding (Bursuc et al., 2025):** The benchmark and methodology we build on. They evaluate 9 LLMs on formal specifications using a CEGIS loop with up to 5 repair iterations. Claude Opus achieves 66.6% on Dafny APPStest; Sonnet achieves 60.3%. They do not use adversarial oracles. We extend their framework by adding an oracle feedback channel and measuring its effect on convergence.
- (b) **Zero-degree-of-freedom coding (Regehr, 2025):** Argues that formal verification constrains LLM code generation to provably correct outputs, eliminating the "gray area"

of subtle bugs. Proposes multi-oracle architectures where different verification tools complement each other. Our adversarial oracle is a direct instantiation of this vision, using an LLM as a semantic oracle alongside the formal verifier.

- (c) **CEGIS (Solar-Lezama et al., 2006)**: The original CEGIS framework uses a formal verifier to produce counterexamples that constrain synthesis. Our work replaces the formal counterexample generator with an LLM oracle, trading soundness for richer semantic feedback.
- (d) **Verified code generation (Clover, Sun et al., 2024; Laurel, Ren et al., 2024)**: Recent systems for generating verified code in Dafny use multi-step pipelines with proof decomposition. ExVerus (Yang et al., 2026) targets Verus/Rust verification with proof-aware prompting. Unlike these systems, we focus on the feedback signal in the repair loop rather than the generation pipeline, making our approach complementary.
- (e) **LLM-based program repair (Xia et al., 2023)**: Prior work uses test failure feedback for repair. Our setting uses formal verification (mathematical proof failures rather than test failures) and adds a natural-language explanation layer via the oracle.

3. Methods

3.1 CEGIS Loop

We replicate the Vericoding paper’s CEGIS methodology: parse a Dafny spec file with tagged placeholder sections (`<vc-helpers>`, `<vc-code>`), call the LLM to fill placeholders as a JSON array, verify with Dafny (120s timeout), and iterate up to 5 times on failure. Each repair is a fresh single-turn API call using the paper’s exact prompt templates. We scan for `assume/{:axiom}` cheats in editable sections (detected in 2 tasks).

3.2 Adversarial Oracle (Condition B)

In Condition B, before each repair call, we query a separate oracle LLM with the specification, broken implementation, and verifier error. The oracle generates a concrete counterexample: specific input values violating postconditions and a brief explanation of which spec clause is violated. This counterexample is appended to the generator’s repair prompt.

3.3 Experimental Design

Condition	Generator	Oracle	Tasks
A (baseline)	Claude Sonnet 4.6	None	50 DA tasks
B (same-tier)	Claude Sonnet 4.6	Claude Sonnet 4.6	50 DA tasks

Condition	Generator	Oracle	Tasks
B (stronger)	Claude Sonnet 4.6	Claude Opus 4.6	17 hard tasks

We evaluate on 50 Dafny APPStest tasks (DA0000-DA0049) from the Vericoding benchmark, with 1 trial per task-condition. The "hard tasks" subset consists of tasks requiring >2 iterations in the baseline.

4. Results

- 4.1 Pass Rates and Convergence

Metric	Cond. A (Verifier)	Cond. B (Verifier+Oracle)
Pass@5	43/50 (86%)	42/50 (84%)
Pass@1	24/50 (48%)	24/50 (48%)
Avg iters (passing, >1 iter)	3.00	2.56

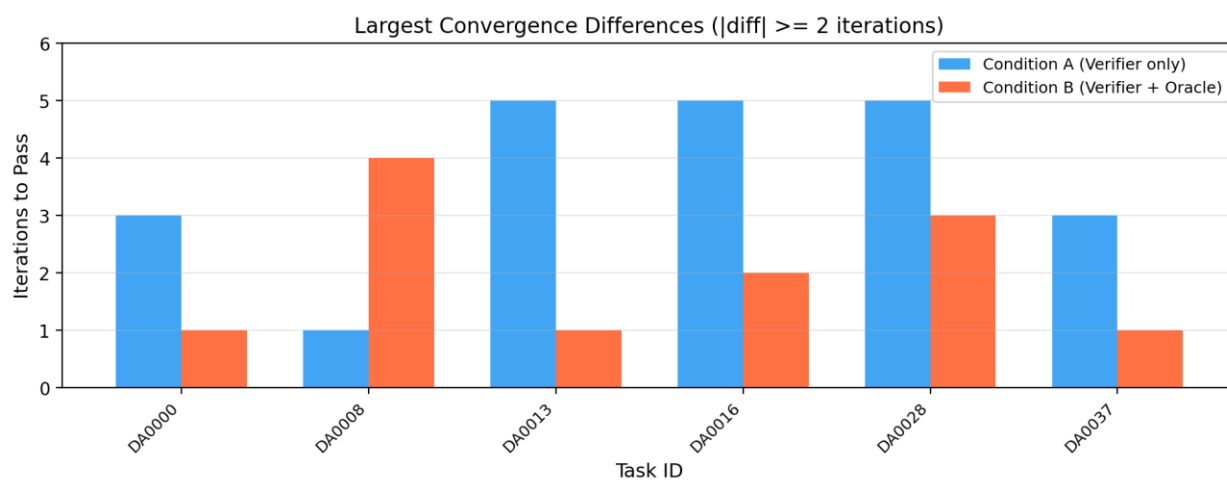
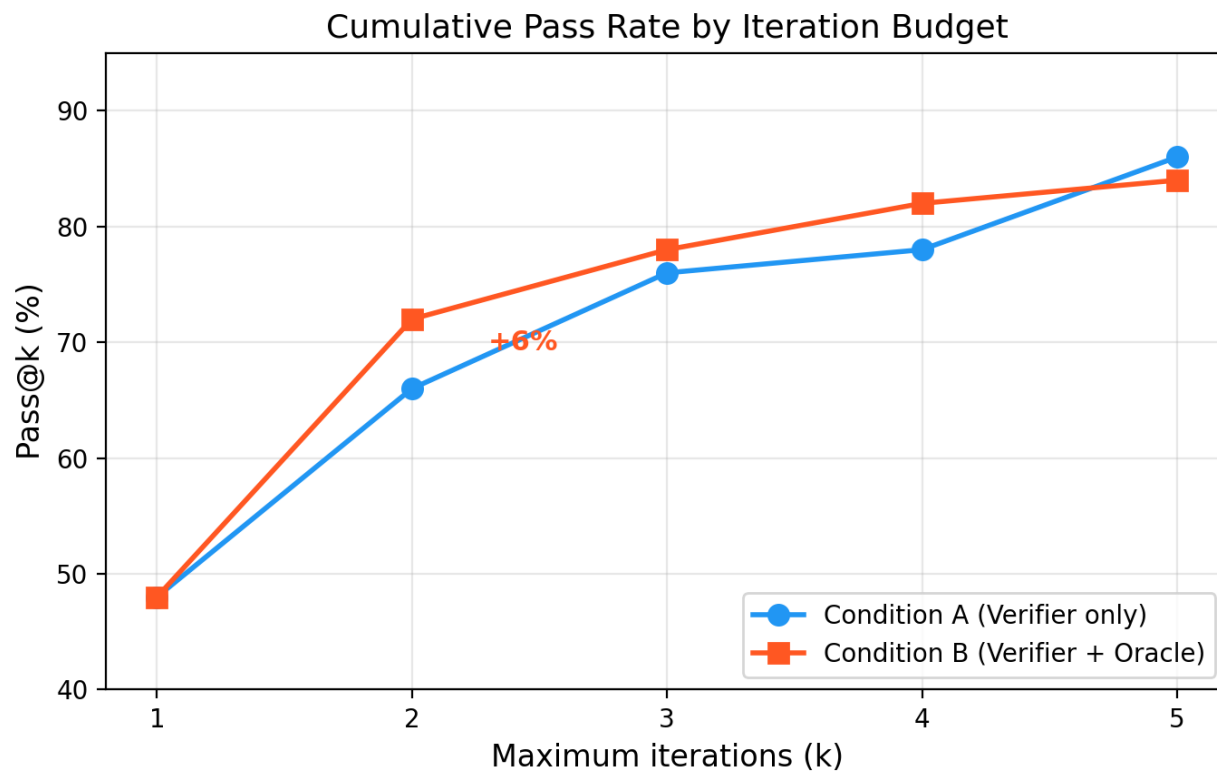
-
- Pass rates are statistically equivalent (the 2% gap is due to one remaining infrastructure failure, a JSON parsing error on DA0019, not oracle-caused harm). The oracle's value appears in *convergence speed*. Among 42 tasks solved by both conditions, B converges faster on 9 tasks vs A on 6 (mean paired difference: +0.21 iters favoring B, Wilcoxon signed-rank $p=0.21$, not significant at conventional thresholds). The effect is concentrated on hard tasks: DA0013 drops from 5 to 1 iteration, DA0016 from 5 to 2, and DA0037 from 3 to 1 (see Figure 2). Not all tasks benefit: DA0008 converges in 1 iteration under Condition A but 4 under Condition B, likely reflecting LLM sampling variance rather than systematic oracle harm (see Limitations).
-

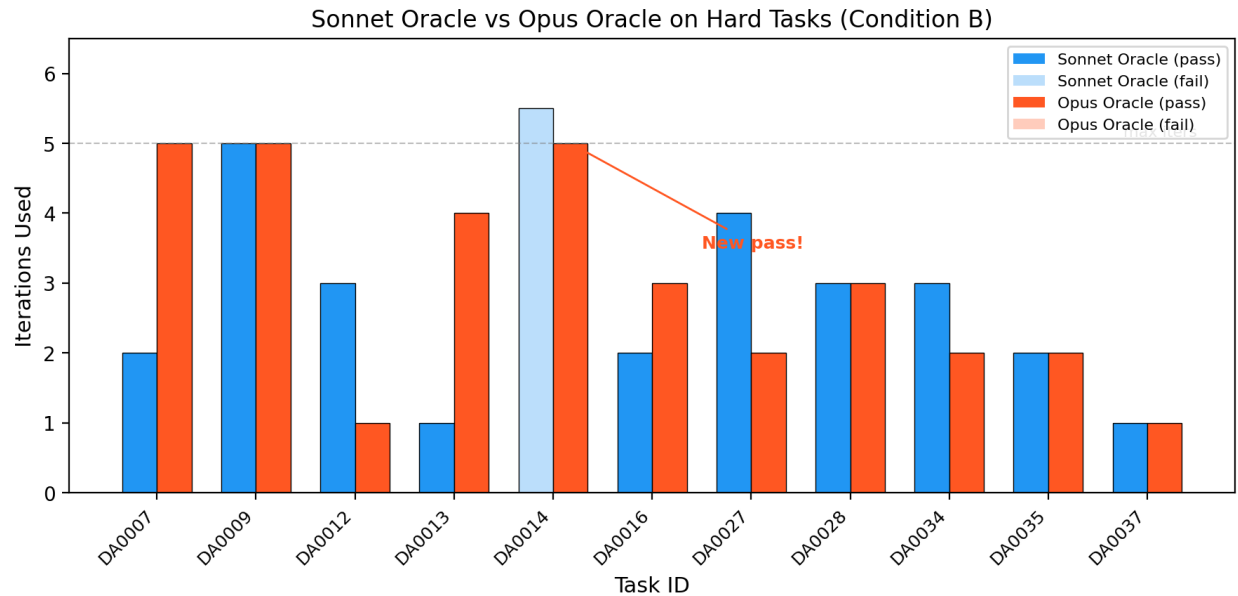
- **Table 2: Pass@k, cumulative pass rate by iteration budget (see Figure 1)**

k	Condition A	Condition B	B advantage
1	48%	48%	0%
2	66%	72%	+6%
3	76%	78%	+2%
4	78%	82%	+4%

k	Condition A	Condition B	B advantage
5	86%	84%	-2%

-
- The peak advantage at pass@2 (+6%) indicates the oracle's primary mechanism: converting 2-iteration solves into 1-iteration solves by providing actionable feedback on the first failure. B maintains a +4% advantage at pass@4. The small reversal at k=5 is due to one remaining infrastructure failure (JSON parsing on DA0019) in Condition B, not oracle feedback causing harm.





4.2 Oracle Quality: Opus vs Sonnet

- We re-ran 17 hard tasks with Claude Opus as the oracle (generator remains Sonnet).
-

Metric	Sonnet Oracle	Opus Oracle
Pass rate	10/17 (59%)	11/17 (65%)
Tasks with faster convergence	—	3 (DA0012, DA0027, DA0034)
Previously unsolvable tasks now solved	—	1 (DA0014)

-
- DA0014, which requires proving that GCD-based fraction reduction produces irreducible fractions, failed under *all prior conditions*: Condition A, Condition B with Sonnet oracle, across multiple runs (see Figure 3).
- 4.3 Case Study: Why Opus Succeeds on DA0014
- The qualitative difference in feedback explains the mechanism:
-
- **Sonnet oracle** (iterations 2-4, task fails):
-
- *"The GcdDivides lemma times out... the postcondition IsIrreducibleFraction cannot be verified."*
-
- Describes the *symptom* (a timeout) without explaining what proof step is missing.

-
- **Opus oracle** (iteration 2, task eventually passes):
-
- *"The GcdDividesNumerator lemma's else branch fails because knowing $(a-b) \% g == 0$ is insufficient to derive $a \% g == 0$. The missing call to `GcdDividesDenominator(a-b, b)` in that branch is needed."*
-
- Identifies the *exact missing proof step*: which lemma to call, in which branch, and why. This transforms repair from blind exploration into targeted surgery.
- **4.4 Taxonomy of Irreducible CEGIS Failures**
- We analyze the 7 tasks that failed under both conditions to identify distinct failure modes:
-

Failure Mode	Tasks	Description
Verification timeout	DA0003, DA0014, DA0023, DA0038	Solver times out on the <code>solve</code> method or supporting lemmas. Generated code may be logically correct but the proof obligations overwhelm the SMT solver within the time limit. DA0014 was eventually solved by the Opus oracle providing proof-engineering hints.
Nonlinear arithmetic	DA0003, DA0024	Specs involve GCD, modular arithmetic, or exponentiation. The LLM generates lemmas with incorrect preconditions (e.g., calling <code>GcdSymmetric(b, a%b)</code> when <code>a%b</code> might be 0, violating <code>requires b > 0</code>) and cannot self-correct the proof structure across iterations.
Complex quantifier reasoning	DA0006	Spec involves nested existential quantifiers over 6-digit ticket permutations. Dafny warns "could not find trigger for this quantifier," and

Failure Mode	Tasks	Description
		the LLM generates loop invariants with untriggerable quantifiers that cause brittle verification.
Specification complexity	DA0038, DA0043	Specs require reasoning about multiple interacting concepts (string traversal with step constraints in DA0038; floor division monotonicity across positive/negative domains in DA0043). The LLM generates helper lemmas that individually time out, preventing the main proof from assembling.
Syntax errors on final attempt	DA0024	In iteration 5, the LLM produces invalid Dafny syntax (<code>ensures gcd_positive(a, b); a % gcd(a, b) == 0, a</code> semicolon in the ensures clause), indicating repair oscillation where the model cycles between proof strategies without converging.

Note: tasks may exhibit multiple failure modes. DA0003 shows both timeout and nonlinear arithmetic issues.

These failure modes suggest different mitigation strategies: timeouts may respond to longer verification budgets or proof decomposition; nonlinear arithmetic requires specialized lemma libraries; quantifier issues need trigger annotations that current LLMs rarely generate. Notably, timeout failures may not be fundamentally irreducible: DA0014 demonstrates that higher-quality proof-engineering guidance from a stronger oracle can overcome them (Section 4.3).

5. Discussion and Limitations

Implications for Secure Program Synthesis

Our results support Regehr's (2025) multi-oracle thesis with an important caveat: adding oracles is necessary but not sufficient. Oracle *quality* determines effectiveness. A same-tier oracle provides marginal convergence benefits, but a stronger oracle can provide the proof-engineering specificity needed to unlock genuinely stuck tasks.

This points to a practical architecture: pair a cost-efficient generator (Sonnet) with a more capable oracle (Opus) invoked *only when repair fails*. The oracle's role is not to generate code but to diagnose proof failures with enough specificity to guide targeted repairs, which is cheaper than using the expensive model for all generation.

For AI safety, faster convergence has a concrete benefit: fewer repair iterations mean fewer opportunities for the generator to introduce specification-bypassing behaviors. Our cheat detection caught 2 `assume`-based bypass attempts in our experiments.

Limitations

Statistical power. 50 tasks with 1 trial each. The convergence trend (Wilcoxon $p=0.21$) is not statistically significant. Our single-trial design is appropriate for a proof-of-concept identifying the mechanism (oracle quality), but insufficient for precise effect-size estimation. The `pass@k` curves and the DA0014 case study motivate the multi-trial follow-up described below.

Task difficulty. Our 50-task subset achieves 86% baseline, higher than the paper's 60-67% on the full 677 tasks. Oracle effects may be larger on harder tasks.

Single case for Opus uplift. The DA0014 result is qualitatively compelling but is a single task.

Future Work

1. Multiple trials (3-5 per condition) to achieve statistical significance.
2. Scaling to the full 677-task Dafny APPStest set, where baseline pass rates are 60-67%, substantially harder than our 50-task subset (86%) and likely to amplify oracle effects.
3. Harder benchmarks (the Verified-Cogen subset of Vericoding, where Sonnet achieves 84%) where oracle effects may be larger.
4. Structured oracle prompts explicitly requesting proof-engineering guidance, mirroring the pattern observed in DA0014.
5. Investigating whether timeout-class failures (4 of 7 in our taxonomy) respond to increased verification budgets or proof decomposition strategies.

6. Conclusion

We investigated adversarial LLM oracles as a complement to formal verifier feedback in CEGIS loops. A same-capability oracle shows early convergence benefits (+6% at pass@2, +4% at pass@4) but does not expand the set of solvable tasks, and the overall convergence difference is not statistically significant with our sample size. The key finding is that oracle *quality* matters: a stronger oracle provides specific proof-engineering guidance, identifying missing lemma calls rather than reporting timeouts, enabling it to solve tasks that no amount of verifier feedback alone can fix. Our taxonomy of irreducible failures reveals that verification timeouts, nonlinear arithmetic, and complex quantifier reasoning are the primary barriers to CEGIS convergence, suggesting targeted mitigation strategies for future work.

Code and Data

- Code repository: <https://github.com/PranavKasetty/vericoding-adversarial>
- Benchmark: Vericoding (Bursuc et al., 2025), <https://github.com/Beneficial-AI-Foundation/vericoding>
- Experimental artifacts: `experiment_results/` in the repository

References

1. Bursuc, S., et al. (2025). "A Benchmark for Vericoding: Formally Verified Program Synthesis." arXiv:2509.22908.
2. Regehr, J. (2026). "Zero-Degree-of-Freedom LLM Coding using Executable Oracles." Blog post, March 2026.
3. Solar-Lezama, A., et al. (2006). "Combinatorial Sketching for Finite Programs." ASPLOS.
4. Sun, C., et al. (2024). "Clover: Closed-Loop Verifiable Code Generation." arXiv:2310.17807.
5. Ren, S., et al. (2024). "Laurel: Generating Dafny Assertions Using Large Language Models." arXiv:2405.16792.
6. Yang, Y., et al. (2026). "ExVerus: Leveraging LLMs for Verified Rust Code Generation." arXiv:2603.25810.
7. Xia, C. S., et al. (2023). "Automated Program Repair in the Era of Large Pre-trained Language Models." ICSE.
8. de Moura, L. and Bjorner, N. (2008). "Z3: An Efficient SMT Solver." TACAS.

LLM Usage Statement

Claude Code was used for implementing the experimental framework, running analysis scripts, and drafting portions of this report. All experimental results were generated by running scripts independently, and all claims were verified against raw data.