CRAFT: Counterexample-guided Repair And Falsification of Theorems in ML and Optimization

Yirui Liu [1] **Jiayu Qin** [2] **Penghang Liu** [3] **Jian Chen** [4]

Abstract

AI systems are increasingly used as research assistants, and mathematical reasoning is a central capability for research across mathematics, engineering, AI, and the sciences. Recent LLMs and reasoning-oriented systems have made rapid progress on mathematical problem solving, but plausible natural-language reasoning is not reliable verification. Models can produce confident explanations for false claims, and checking those explanations by hand may simply move the bottleneck from generation to verification. We present **CRAFT** (**C**ounterexample-guided **R**epair **A**nd **F**alsification of **T**heorems), a counterexample-first workflow for mathematical claims that admit executable falsification.[1] CRAFT first asks an LLM to analyze a natural-language claim and produce a structured true/false/uncertain judgment with a verbal rationale. A verifier then searches for concrete counterexamples: found counterexamples are fed back to expose errors and support repair, while calibrated no-counterexample signals can prompt reconsideration of false alarms. Repaired statements are re-checked through the same executable interface. On a 50-claim benchmark spanning 25 ML/optimization families, verifier feedback improves or preserves accuracy across seven models. For `qwen2.5-7b-instruct`, accuracy improves from 78% to 94%. Search quality determines verifier reliability, verifier feedback corrects errors in the round-1 LLM-only baseline, and all 25 canonical repair targets pass re-checking.

[1]Institute of Artificial Intelligence (TeleAI), China Telecom [2]University at Buffalo [3]JPMorganChase [4]Dolby Laboratories. Correspondence to: Jian Chen <Jian.Chen@dolby.com>.

Proceedings of the 43rd International Conference on Machine Learning, Seoul, South Korea. PMLR 306, 2026. Copyright 2026 by the author(s).

[1]Code, benchmark, and results: https://github.com/yiruiliu/Craft_CounterExample

1. Introduction

AI systems are increasingly used as research assistants, from literature search and code generation to hypothesis generation and mathematical problem solving (Lu et al., 2024). Among these capabilities, mathematical reasoning is especially important: across mathematics, engineering, AI, and the sciences, researchers routinely need to solve technical problems, check assumptions, and repair faulty arguments. This has made LLM-based mathematical problem solving a prominent testbed for AI-assisted research.

Recent LLMs and reasoning-oriented systems have made rapid progress on mathematical reasoning and problem solving. This progress is reflected in stronger performance on benchmarks such as MATH and FrontierMath, as well as in methods that use intermediate reasoning to improve mathematical answers (Wei et al., 2022; Hendrycks et al., 2021; Glazer et al., 2024; Zheng et al., 2022; Yang et al., 2023; Kommuru & Khanvilkar, 2024; Zhang et al., 2025; Li et al., 2025b). For research use, however, producing a plausible answer is not enough. Models can produce confident natural-language explanations for false claims, and checking such explanations by hand may require finding the very missing assumption, loose constant, or invalid regime that the model overlooked. In this setting, LLM assistance does not necessarily accelerate research: it can simply move the bottleneck from generation to verification.

To address this problem, we propose **CRAFT**, a counterexample first workflow for mathematical claims. The key idea is that many mathematical claims in ML and optimization admit executable falsification: while proving such claims true may be difficult, a false claim can often be exposed by a concrete counterexample whose violation is mechanically checkable. CRAFT first asks an LLM to analyze the natural-language claim and produce a structured true/false/uncertain judgment with a verbal rationale. A verifier then searches for concrete counterexamples; when one is found, it is fed back to the LLM to expose the error, correct the judgment, and propose a repair. Repaired statements are re-checked through the same executable interface.

We evaluate this workflow on a 50-claim benchmark spanning 25 mathematical families in ML and optimization. The

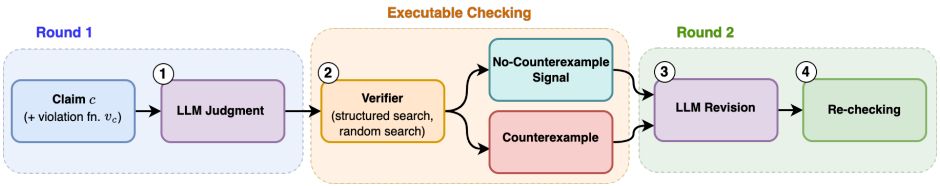

Figure 1. **Overview of CRAFT.** CRAFT obtains a round-1 LLM judgment for a claim c, performs executable checking with v_c, feeds either a counterexample or no-counterexample signal back to the LLM, and re-checks candidate repairs through the same executable interface.

benchmark is designed around realistic draft-stage mutations: each false claim is obtained by weakening a corresponding true claim through a dropped assumption, loosened constant, or overstated condition. Across seven models, verifier feedback improves or preserves round-1 accuracy, with the largest gains on smaller and math-specialized models. For example, on `qwen2.5-7b-instruct`, the loop improves accuracy from 78% to 94%, while all 25 canonical repair targets pass repair validation under re-checking. Our contributions are:

- We introduce a reproducible 50-claim benchmark of executable mathematical claims in ML and optimization, together with violation functions that make counterexample search mechanically testable.
- We present a verifier-in-the-loop workflow that uses found counterexamples to correct false acceptances, calibrated no-counterexample signals to reconsider false alarms, and the same executable interface to validate repairs.
- We show empirically that verifier search quality matters, that verifier feedback corrects errors in the round-1 LLM-only baseline, and that all 25 canonical repair targets pass re-checking.

Related work is reviewed in Appendix A owing to the page limit.

2. Method

Our method targets a restricted but practically important class of research claims: mathematical claims arising in ML and optimization that admit executable falsification. Each claim is paired with a violation function whose positive value certifies a concrete counterexample. This executable structure creates an opportunity for tight coupling between a verifier and an LLM. CRAFT uses the verifier to search for counterexamples and generate external checking signals, while the LLM uses those signals to correct its initial judgments and propose candidate repairs, yielding a stronger claim-checking workflow than LLM-only judgment.

2.1. Problem Formulation

We study draft-stage mathematical claims that arise in ML and optimization and that are sufficiently precise to support executable checking. Each claim is presented in natural language, but is assumed to admit an executable representation through a violation function that can be evaluated on concrete inputs. This is the interface that lets the workflow move from verbal judgment to mechanical checking.

Formally, let c denote a claim. We assume access to a non-negative violation function $v_c(z) \geq 0$ over the executable input domain of claim c, where z denotes a concrete input instance. The key property is that $v_c(z) > 0$ if and only if z is a counterexample to c; candidate instances are drawn only from the premise-satisfying domain, so any returned certificate satisfies the claim's assumptions by construction. The internal structure of z may vary across claim families, but the verifier relies only on this common contract: positive violation means certified failure. (Appendix C provides a broader scope discussion together with representative claims and their corresponding violation functions.)

2.2. The CRAFT Loop

As shown in Figure 1, CRAFT organizes one claim-checking attempt into two LLM rounds separated by executable checking. Round 1 obtains the model's initial judgment; the verifier searches for mechanically checkable evidence of failure; and round 2 feeds the verifier signal back to support correction, candidate repair, and re-checking.

1. **Initial LLM judgment.** The LLM reads the natural-language claim and produces a first-pass structured judgment, including a verdict, a rationale, and, when appropriate, a candidate repair.
2. **Verifier search.** The verifier searches the executable input domain of c for an instance z such that $v_c(z) > 0$. It returns one of two signals: a mechanically checkable counterexample certificate if a violating instance is found, or a no-counterexample signal if the search budget is exhausted without finding one.
3. **LLM revision under external signal.** The verifier signal is fed back to the LLM. A counterexample is

the primary correction signal: it can overturn an incorrect initial judgment and support a revised explanation or candidate repair. A no-counterexample signal is weaker; when search is sufficiently strong, it may prompt the LLM to reconsider an overconfident false judgment, but it does not certify correctness.

4. **Re-checking repaired statements.** When the LLM proposes a repaired statement, the verifier is applied again to the repaired claim through the same executable interface. In our benchmark setting, this re-checking step is evaluated against a canonical repair target.

Each attempt is bounded by construction—one initial judgment, T verifier evaluations, at most one revision round, one repair re-check—so the loop cannot iterate indefinitely.

2.3. Verifier

The verifier is the component that gives CRAFT its external checking signal. Once a candidate instance z is available, deciding whether it violates a claim is mechanical through $v_c(z) > 0$. The core design problem is therefore not checking—which is trivial given a candidate—but *generation*: how to find inputs that expose failure when a claim is false.

The difficulty is that some false claims have violation regions that are sparse or low-measure. Step-size claims may fail only just above a threshold; matrix claims may fail only near the positive semidefinite boundary; structural claims may have exactly one or a handful of discrete witnesses. Uniform random sampling can require many trials to reach these regions, even though any instance found is immediately verifiable.

Our verifier addresses this with two complementary search layers, paralleling the generator design of property-based testing (Claessen & Hughes, 2000). The first is *random search*, which samples directly from the claim's executable domain and provides a claim-agnostic baseline. The second is *structured search*, which applies family-specific constructions targeting known hard regions—for example, evaluating near a claimed numeric threshold or constructing matrices near the positive semidefinite boundary. Structured search runs first when a matching handler is available; random search supplies the remaining coverage. Both layers share the same interface: they propose candidate instances, evaluate them through v_c, and return a certificate only when positive violation is observed. Both layers run for a fixed evaluation budget; a returned certificate is mechanically checkable, but failure to find one does not certify correctness.

3. Experiments

3.1. Benchmark and Setup

We evaluate on `formal_eval`, a 50-claim benchmark of executable mathematical claims in ML and optimization. The benchmark contains 25 claim families, each with one true claim and one mutated false claim, yielding 25 true and 25 false claims in total. False claims are constructed by dropping assumptions, loosening tight constants, or replacing stronger structural conditions with weaker ones. All false claims are marked `false_repairable` and paired with a canonical repair target.

All models are queried through the Aliyun OpenAI-compatible endpoint with temperature 0. Round 1 asks for a verdict ('true', 'false', or 'uncertain'), a brief rationale, and a repair when appropriate. Round 2 appends either a concrete counterexample or a no-counterexample message from the verifier. For math-specialized models that answer in chain-of-thought style rather than JSON, we use a fallback verdict parser; unparsed responses are counted as errors.

3.2. Finding 1: Verifier Feedback Improves Model Judgments

Table 1 is the main result of the paper. Across all reported runs, the loop improves or preserves round-1 accuracy, and the gains are largest on weaker models. For `qwen2.5-7b-instruct`, the loop improves accuracy from 78% to 94%, correcting 8 errors. For the math-specialized `qwen2.5-math-7b-instruct`, the gain is 52% to 66%. At larger scales, the loop still helps: `qwen2.5-72b-instruct` and `MiniMax-M2.5` both reach 100%, and `deepseek-v4-flash` improves from 90% to 98%.

The dominant failure mode corrected by the loop is the false negative: the model initially accepts a subtly false claim, and a concrete counterexample flips the decision. This pattern is especially strong for the 7B and math-specialized models, which tend to over-trust theorem-like statements. By contrast, models such as `qwen-plus` make relatively more false alarms on true claims, so their absolute gain depends more on whether verifier silence can be used safely.

3.3. Finding 2: Search Quality Determines Verifier Effectiveness

Executable checking is already strong with random search alone, but its effectiveness depends heavily on counterexample generation. At 10 trials, random search finds counterexamples for 84% of false claims with no false alarms on true claims. At 50 trials, it reaches 100% discovery. Structured search closes this gap much earlier: a structured-

Table 1. Main loop results on the 50-claim benchmark. 'R1 Acc.' is the LLM-only round-1 accuracy. 'Loop Acc.' is the final accuracy after verifier feedback and optional round-2 revision. 'R2 corrected' counts how many second-round calls changed the final verdict from wrong to correct.

Model	Type	R1 Acc.	Loop Acc.	FN (R1)	FA (R1)	R2 corrected
`qwen2.5-7b-instruct`	General-7B	39/50 (78%)	47/50 (94%)	8	3	8
`qwen2.5-math-7b-instruct`	Math-7B	26/50 (52%)	33/50 (66%)	18	6	7
`qwen2.5-72b-instruct`	General-72B	45/50 (90%)	50/50 (100%)	5	0	5
`qwen2.5-math-72b-instruct`	Math-72B	38/50 (76%)	47/50 (94%)	3	2	8
`deepseek-v4-flash`	Frontier	45/50 (90%)	49/50 (98%)	3	2	4
`qwen-plus`	Frontier	46/50 (92%)	47/50 (94%)	0	4	1
`MiniMax-M2.5`	Frontier	48/50 (96%)	50/50 (100%)	1	1	2

plus-random verifier already reaches 100% discovery at 20 trials.

The missed cases at low random-search budgets are not generic failures, but claims with thin violation regions. These include near-boundary step-size statements, near-PSD matrix claims, and structural violations that are effectively discrete. Once a candidate counterexample is proposed, checking is trivial through $v_c(z)$; the challenge is finding the counterexample. This is why the verifier design in §2.3 is centered on search rather than on more elaborate checking logic.

3.4. Finding 3: Correction Signals Are Asymmetric

The two correction signals in CRAFT are not equally reliable. A counterexample is a strong signal because it directly certifies failure of the current claim statement. Verifier silence is weaker: it only says that no counterexample was found under the current search budget. This asymmetry appears clearly in the per-model breakdown. For `qwen2.5-72b-instruct`, all five round-1 errors are false negatives, and all five are corrected by counterexample feedback alone. For `qwen-plus`, the model makes only false alarms, and the loop gains just one point because silence is a much weaker corrective signal than a counterexample.

We verified this directly with a safety ablation. When the 7B loop is rerun with only 10 random-search trials, the verifier discovery rate drops below the high-coverage regime, and silence begins to cause harm by weakening correct 'false' decisions. Thus false-alarm correction should be enabled only when discovery is calibrated to near-saturation on held-out false claims.

3.5. Finding 4: Counterexamples Improve Repair Localization

Table 2 isolates the role of counterexample values in round 2. Simply telling the model that a counterexample exists is already sufficient to flip the verdict on all six tested false-negative claims. However, this weak form of feedback only yields 2/6 correct repairs. Providing the full counterexample improves repair quality to 4/6, and using a stronger round-2 model with the same counterexample reaches 6/6.

Table 2. Repair localization ablation on six false-negative claims. Merely notifying the model that a counterexample exists is enough to flip the verdict, but full counterexample values substantially improve repair quality.

Feedback to R2	Verdict corrected	Repair matched
No feedback	0/6	0/6
Notify only	6/6	2/6
Full CE	6/6	4/6
Strong R2 + CE	6/6	6/6

This result clarifies what the counterexample contributes. Its main value is not persuasion, but localization: the counterexample identifies the parameter regime in which the claim fails, which helps the model repair the right assumption or constant. Without that anchor, the model often proposes vague or mislocalized repairs even when it correctly changes the verdict. Repair validation is the most stable result across all models: in every reported loop run, all 25 canonical repair targets pass re-checking, confirming that the same executable interface used for falsification can validate corrections mechanically.

4. Discussion

Executable counterexamples do more than correct verdicts: they localize failure, anchoring repair to the failing parameter regime (Table 2).

The weakest gains occur on the math-specialized 7B model, possibly because it treats claims as theorems to confirm or because our benchmark differs from olympiad-style training data.

CRAFT applies only to claims with executable violation functions; encoding each family took 20–40 lines of Python (sampler, violation function, witness serializer); automating this is a natural next step. Verifier silence is evidence about the search budget, not proof of truth.

References

Azerbayev, Z., Piotrowski, B., Schoelkopf, H., Ayers, E. W., Radev, D., and Avigad, J. Proofnet: Autoformalizing and formally proving undergraduate-level mathematics. *arXiv preprint arXiv:2302.12433*, 2023.

Claessen, K. and Hughes, J. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pp. 268–279. ACM, 2000.

First, E., Rabe, M. N., Ringer, T., and Brun, Y. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1229–1241, 2023.

Glazer, E., Erdil, E., Besiroglu, T., Chicharro, D., Chen, E., Gunning, A., Bluemke, C., Heffernan, L., Denain, J.-S., and Markou, H. FrontierMath: A benchmark for evaluating advanced mathematical reasoning in AI. *arXiv preprint arXiv:2411.04872*, 2024.

Gundersen, O. E., Kjensmo, S. F., Nygaard, V., et al. The need for verification in AI-driven scientific discovery. *arXiv preprint*, 2025.

Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D., and Steinhardt, J. Measuring mathematical problem solving with the MATH dataset. In *Advances in Neural Information Processing Systems, Datasets and Benchmarks Track*, 2021.

Huang, J., Chen, X., Mishra, S., Zheng, H. S., Yu, A. W., Song, X., and Zhou, D. Large language models cannot self-correct reasoning yet. In *International Conference on Learning Representations*, 2024.

Jiang, A. Q., Welleck, S., Zhou, J. P., Li, W., Liu, J., Jamnik, M., Lacroix, T., Wu, Y., and Lample, G. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *International Conference on Learning Representations*, 2023.

Kommuru, K. and Khanvilkar, K. ProofSketcher: Hybrid LLM + lightweight proof checker for reliable math/logic reasoning. *IEEE Conference Proceedings*, 2024.

Li, Y., Kuang, J., Huang, H., et al. One example shown, many concepts known! counterexample-driven conceptual reasoning in mathematical LLMs. In *International Conference on Machine Learning*, 2025a.

Li, Z., Li, Z., Yang, K., Ma, X., and Su, Z. Learning to disprove: Formal counterexample generation with large language models. *arXiv preprint arXiv:2502.11849*, 2025b.

Lightman, H., Kosaraju, V., Burda, Y., Edwards, H., Baker, B., Lee, T., Leike, J., Schulman, J., Sutskever, I., and Cobbe, K. Let's verify step by step. In *International Conference on Learning Representations*, 2024.

Lu, C., Lu, C., Lange, R. T., Foerster, J., Clune, J., and Ha, D. The AI scientist: Towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292*, 2024.

Ospanov, A., Feng, Z., Sun, J., Bai, H., Shen, X., and Farnia, F. HERMES: Towards efficient and verifiable mathematical reasoning in LLMs. *arXiv preprint*, 2025.

Polu, S. and Han, J. M. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393*, 2020.

Ren, Z., Guo, D., Shao, Z., et al. DeepSeek-Prover-V2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. *arXiv preprint arXiv:2504.21801*, 2025.

Trinh, T. H., Wu, Y., Le, Q. V., He, H., and Luong, T. Solving olympiad geometry without human demonstrations. *Nature*, 625:476–482, 2024.

Wang, P., Li, L., Shao, Z., Xu, R., Dai, D., Li, Y., Chen, D., Wu, Y., and Sui, Z. Math-shepherd: Verify and reinforce LLMs step-by-step without human annotations. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 2024.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q. V., and Zhou, D. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, 2022.

Welleck, S., Liu, J., Le Bras, R., Hajishirzi, H., Choi, Y., and Cho, K. NaturalProofs: Mathematical theorem proving in natural language. In *Advances in Neural Information Processing Systems, Datasets and Benchmarks Track*, 2021.

Xin, H., Guo, D., Shao, Z., Ren, Z., Zhu, Q., Liu, B., Ruan, C., Li, W., and Liang, X. DeepSeek-Prover: Advancing theorem proving in LLMs through large-scale synthetic data. *arXiv preprint arXiv:2405.14333*, 2024.

Yang, K., Swope, A., Gu, A., Chalamala, R., Song, P., Yu, S., Godil, S., Prenger, R., and Anandkumar, A. Leandojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems*, 2023.

Zhang, H. et al. Step-wise formal verification for LLM-based mathematical problem solving. In *Proceedings of the Association for Computational Linguistics*, 2025.

Zheng, K., Han, J. M., and Polu, S. MiniF2F: A cross-system benchmark for formal olympiad-level mathematics. In *International Conference on Learning Representations*, 2022.

A. Related Work

Formal theorem proving systems and benchmarks. A large body of work uses LLMs to interact with formal proof assistants such as Lean, Isabelle, and Coq, where every proof step is checked by a trusted kernel. Early systems such as GPT-f (Polu & Han, 2020) showed that generative language models can contribute machine-checkable proofs in formal libraries. Subsequent work has used informal proofs to guide formal proof search, as in Draft, Sketch, and Prove (Jiang et al., 2023), and generated and repaired whole formal proofs, as in Baldur (First et al., 2023). LeanDojo (Yang et al., 2023) provides an open toolkit for retrieval-augmented interaction with Lean 4, enabling programmatic proof search at academic scale. AlphaGeometry (Trinh et al., 2024) achieves IMO gold-medal performance on geometry by coupling a language model with a symbolic deduction engine. MiniF2F (Zheng et al., 2022) provides a cross-system benchmark of Olympiad-level problems, and NaturalProofs (Welleck et al., 2021) studies informal mathematical reasoning in natural language. ProofNet (Azerbayev et al., 2023) targets autoformalization of undergraduate-level mathematics. More recent theorem-proving models, including DeepSeek-Prover and DeepSeek-Prover-V2 (Xin et al., 2024; Ren et al., 2025), further improve Lean-based formal reasoning through large-scale synthetic data and reinforcement learning. These systems produce formal proofs—mechanically checked certificates of truth. Our work occupies a different niche: we do not produce formal proofs, and we do not require one. Instead, executable violation functions certify counterexamples numerically. This is less expressive (we cannot certify that a claim is *true*), but far more accessible for the quantitative inequality claims that permeate ML and optimization, where Lean-style formalization would require significant domain-specific engineering.

LLM mathematical reasoning. Chain-of-thought prompting (Wei et al., 2022) demonstrates that intermediate reasoning steps unlock complex multi-step problem solving in LLMs. The MATH benchmark (Hendrycks et al., 2021) and FrontierMath (Glazer et al., 2024) characterize LLM math capabilities at competition and research levels, respectively, with FrontierMath revealing that frontier models remain far below expert performance on genuinely novel problems. Despite these advances, LLM verbal reasoning produces *arguments* rather than *certificates*: a model may give a convincing explanation for a false claim. CounterMATH (Li et al., 2025a) directly surfaces this gap by testing whether LLMs can generate counterexamples to false mathematical statements, finding significant failures even in frontier models. This epistemological gap—between a plausible verbal argument and an independently checkable proof—is the central motivation for CRAFT.

Counterexample generation and falsification. CounterMATH (Li et al., 2025a) is the benchmark most directly related to our task: it evaluates LLMs on counterexample-driven proof across university-level mathematics. Li et al. (2025b) train LLMs via reinforcement learning to produce formal counterexamples verifiable in Lean 4, synthesizing training data by dropping necessary hypotheses from known theorems. Both works focus on counterexample *generation* as an isolated capability. CRAFT closes the verification loop: a numerical verifier produces mechanically checkable certificates that feed back into the LLM to correct initial verdicts and anchor candidate repairs, and the same verifier automatically validates each repair. Our certificates require no formal proof infrastructure and no specialized training—a critical advantage for ML/optimization claims, where the relevant structure is quantitative inequality and the relevant tool is a Python script, not a proof assistant.

LLM self-correction and external feedback. Huang et al. (2024) show that LLMs cannot reliably self-correct reasoning through introspection alone: re-prompting a model to reconsider its answer, absent any oracle signal, does not consistently improve accuracy and can degrade performance. This finding directly motivates the design of CRAFT: rather than asking the LLM to reconsider its reasoning, we supply an *external* mechanically generated signal—a concrete numerical counterexample—that unambiguously certifies failure. The verifier is the external oracle; the LLM's second-round output is conditioned on this oracle signal, not on self-reflection.

LLMs coupled with external verifiers. A growing body of work pairs LLMs with trusted external checkers under the "untrusted producer, trusted checker" architecture. ProofSketcher (Kommuru & Khanvilkar, 2024) uses LLM-generated proof sketches verified by a lightweight trusted kernel. HERMES (Ospanov et al., 2025) interleaves Lean 4 verification at each reasoning step, providing step-level feedback to guide subsequent generation. Step-wise formal verification (Zhang et al., 2025) applies CAS and SMT solvers to verify each step of an LLM math solution, returning formal errors as correction prompts. Pipeline-based verification (Gundersen et al., 2025) proposes multi-agent workflows for automatically checking LLM-generated mathematical solutions against formal proof systems.

On the process supervision side, Lightman et al. (2024)

show that training a verifier to assign step-level correctness labels—process reward modeling—substantially outperforms outcome-only supervision for math problem solving. Math-Shepherd (Wang et al., 2024) extends this with automatically constructed process supervision labels, removing the need for human annotation and enabling RL-based policy improvement.

CRAFT shares the "trusted external signal" architecture with all these works. The key distinction is what the checker produces: formal proof terms or step-level neural scores in prior work; *concrete numerical witness values* in ours. Witnesses are universally verifiable—anyone can run the violation function and inspect the output—without access to a proof assistant or a trained reward model.

AI for research and the verification bottleneck. The AI Scientist (Lu et al., 2024) demonstrates end-to-end research automation in ML but identifies trustworthy claim verification as a persistent open challenge. Gundersen et al. (2025) argue that verification is the fundamental bottleneck in AI-driven scientific discovery: generative AI can produce hypotheses faster than they can be checked, and verbal rationales do not constitute logically derivable verification. CRAFT addresses this bottleneck for a specific high-value subclass of claims—the numerically executable lemmas and inequalities that arise throughout ML and optimization theory—by providing a verification layer that produces independently checkable certificates rather than model-generated assessments.

B. Case Study: Eight False Claims Accepted by the 7B Model

The 7B model (qwen2.5-7b-instruct) produces confident verbal rationales for all eight false claims it accepts. In every case the rationale sounds mathematically plausible, appeals to a real theorem or principle, and contains no flagged uncertainty—yet the claim is false and a concrete counterexample exists. This section lists all eight cases, showing the false claim, the model's R1 reasoning, the numerical counterexample that certifies failure, the structural flaw, and the repair accepted by the verifier.

Premise enforcement and witness audit. Each claim's sampler draws candidate instances only from the premise-satisfying domain, so any returned certificate satisfies the claim's assumptions by construction. We additionally audited every witness reported below by re-evaluating its violation function from the recorded values and checking each claim's premises; all eight witnesses satisfy the premises of their claims and reproduce the recorded violations exactly. In particular, the witness in Case 3 is the *affine* function $f(x) = bx$ with $a = 0$ (the recorded instance is $f(x) = ax^2 + bx$), which is convex, so the premise of the claim holds.

Case 1: Gradient descent with any step size. Claim: *For an L-smooth convex quadratic, gradient descent with any positive step size does not increase the objective in one step.*

R1 rationale (7B): "For a convex quadratic function, the gradient descent update rule will always decrease the objective function for any positive step size if the function is L-smooth."

Counterexample: $f(x) = ax^2$ with $a = 0.51$ ($L = 2a = 1.02$), $x = -3.87$, $\eta = 2.14 > 2/L \approx 1.96$. Objective increases by 3.08.

Flaw: The step-size bound $\eta \leq 1/L$ is required for descent; "any positive" removes it.

Repair (verified): *For an L-smooth convex quadratic, gradient descent with $\eta \leq 1/L$ does not increase the objective in one step.*

Case 2: Step size at the boundary. Claim: *For an L-smooth convex quadratic, gradient descent with $\eta \leq 2.05/L$ does not increase the objective in one step.*

R1 rationale (7B): "For a convex quadratic function, the gradient descent update rule ensures that the objective value decreases or stays the same in each step if the step size is chosen appropriately."

Counterexample: $f(x) = ax^2$ with $a = 2.30$ ($L = 2a = 4.60$), $x = 3.43$, $\eta = 0.441$ (within the claimed range $\eta \leq 2.05/L \approx 0.446$, but above the safe bound $2/L \approx 0.435$). Objective increases by 1.48.

Flaw: The tight bound is $\eta \leq 2/L$; the constant 2.05 is just outside the safe range. The model's rationale is generic and does not notice the constant is wrong.

Repair (verified): *For an L-smooth convex quadratic, gradient descent with $\eta \leq 2/L$ does not increase the objective in one step.*

Case 3: Strong convexity inequality without strong convexity. Claim: *If f is convex, then $f(y) \geq f(x) + \nabla f(x)^\top (y - x) + \frac{\mu}{2}\|y - x\|^2$ for $\mu > 0$.*

R1 rationale (7B): "The inequality represents the quadratic form of a convex function, which is a valid property of convex functions."

Counterexample: the affine function $f(x) = bx$ with $b = -1.84$ (convex, with zero curvature), $x = -0.91$, $y = -2.90$, $\mu = 0.93$. The first-order terms cancel exactly, so the inequality fails by the full quadratic term: violation $\frac{\mu}{2}(y - x)^2 = 1.85$.

Flaw: The μ-quadratic lower bound requires μ-*strong* convexity; a merely convex function can have zero curvature, leaving nothing to absorb the $\frac{\mu}{2}\|y-x\|^2$ term.

Repair (verified): *If f is μ-strongly convex, then $f(y) \geq f(x) + \nabla f(x)^\top (y-x) + \frac{\mu}{2}\|y-x\|^2$.*

Case 4: Unique minimizer of a convex function. Claim: *A convex function has at most one global minimizer.*

R1 rationale (7B): "By definition, a convex function has a unique global minimizer."

Counterexample: $f(x) = 0$ (the zero function, which is convex). Any two distinct points $x_1 = -0.23$ and $x_2 = 0.28$ are both global minimizers. Violation indicator 1.0.

Flaw: Uniqueness requires *strict* or *strong* convexity. The model conflates convex with strictly convex by definition.

Repair (verified): *A strongly convex function has at most one global minimizer.*

Case 5: PSD from nonnegative diagonal entries. Claim: *If A is symmetric and all diagonal entries are nonnegative, then $x^\top Ax \geq 0$ for all x.*

R1 rationale (7B): "This is a property of positive semi-definite matrices. A symmetric matrix with nonnegative diagonal entries is indeed positive semi-definite."

Counterexample:

$$A = \begin{pmatrix} 1.54 & -1.25 \\ -1.25 & 0.98 \end{pmatrix}, \quad x = \begin{pmatrix} 0.74 \\ 0.93 \end{pmatrix}.$$

$x^\top Ax = -0.032 < 0$.

Flaw: Nonnegative diagonal does not imply PSD; large off-diagonal entries can make the matrix indefinite.

Repair (verified): *If A is positive semidefinite, then $x^\top Ax \geq 0$ for all x.*

Case 6: Jensen's inequality without convexity. Claim: *For any differentiable function f, Jensen's inequality holds on convex combinations.*

R1 rationale (7B): "Jensen's inequality states that for a convex function f and any set of weights that sum to 1, the weighted average of f applied to the values is greater than or equal to f applied to the weighted average of the values."

Counterexample: $f(x) = ax^2 + bx$ with $a = -0.008$, $b = -0.81$ (concave-dominant), $x = -1.65$, $y = 1.81$, $\lambda = 0.68$. Violation 0.020.

Flaw: Jensen requires f to be *convex*; "differentiable" is not sufficient. The model's rationale is self-contradictory: it correctly states the convexity requirement in its explanation yet judges the claim true.

Repair (verified): *If f is convex, then Jensen's inequality holds on convex combinations.*

Case 7: Gradient norm bound with wrong constant. Claim: *For an L-smooth convex quadratic f with minimum value f^*, $\|\nabla f(x)\|^2 \leq \frac{L}{2}(f(x) - f^*)$.*

R1 rationale (7B): "For a convex quadratic function, the gradient norm squared can be bounded by a constant times the function value minus the minimum value. This is a known property of such functions."

Counterexample: $f(x) = \frac{a}{2}x^2 + bx$ with $a = 2.66$ ($L = 2.66$), $b = -0.86$, $x = -1.57$. $\|\nabla f\|^2 \approx 25.4$, but $\frac{L}{2}(f - f^*) \approx 6.35$. Violation 19.1.

Flaw: The correct constant is $2L$, not $L/2$ (a factor of 4 too small). The model recognizes the form of the bound but does not check the constant.

Repair (verified): *For an L-smooth convex quadratic f with minimum value f^*, $\|\nabla f(x)\|^2 \leq 2L(f(x) - f^*)$.*

Case 8: Log-sum-exp lower bound with spurious additive term. Claim: *For any real numbers a and b,* $\log(\exp(a) + \exp(b)) \geq \max(a, b) + \log 2$.

R1 rationale (7B): "The inequality holds due to the properties of logarithms and exponentials."

Counterexample: $a = -2.55$, $b = 2.50$. $\log(\exp(a) + \exp(b)) \approx 2.50$, but $\max(a, b) + \log 2 \approx 3.19$. Violation 0.69.

Flaw: $\log(\exp(a) + \exp(b)) = \max(a, b) + \log(1 + \exp(-|a - b|))$. When $|a - b|$ is large, the second term approaches 0, not $\log 2$. The $+\log 2$ bound holds only at $a = b$; it fails when the two arguments are far apart.

Repair (verified): *For any real numbers a and b,* $\log(\exp(a) + \exp(b)) \geq \max(a, b)$.

C. Problem Formulation Details

Scope of executable claims. CRAFT does not target arbitrary mathematical statements. The setting we study is a restricted but practically important subclass of ML and optimization claims whose failure can be exhibited by a finite concrete instance and checked mechanically by a violation function. This includes many inequalities, one-step optimization statements, matrix-condition claims, and local counterexample-certifiable arguments. It excludes statements whose validity depends on long symbolic derivations or quantifier structure that cannot be reduced to executable checking. The benchmark is constructed around this requirement: every claim in the evaluation set is included only if such a violation function can be defined.

Representative claims. The executable structure appears in several recurring claim families in our benchmark. Examples include:

- gradient-descent descent claims, where an input instance specifies a function, a step size, and an iterate;
- quadratic-form claims, where an input instance specifies a matrix and a vector;
- convexity and smoothness inequalities, where an input instance specifies the function parameters and the points at which the inequality is evaluated.

Representative violation functions. For a claim of the form "gradient descent with step size η does not increase the objective," the violation function can be written as the one-step increase

$$v_c(z) = \max\{0, f(x^+) - f(x)\},$$

where z contains the function parameters, the iterate x, and the step size η. For a claim of the form "$x^\top Ax \geq 0$ for all x," the violation function can be written as

$$v_c(z) = \max\{0, -x^\top Ax\},$$

where z contains the matrix A and the vector x. For a Jensen-type convexity claim, the violation function can be written as the amount by which the convexity inequality is exceeded. In each case, the same contract is preserved: $v_c(z) > 0$ if and only if the tested instance exhibits failure.