The Crucible: A Hack-Resistant Reward Harness for Autonomous Optimization of LLM Inference Stacks

Abstract

Inference now dominates the cost of operating large language models, and the software that serves them, attention kernels, cache policies, schedulers, and speculative-decoding configurations, runs far below the hardware roofline [10], so a system that lets a language model rewrite and retune its own serving stack is economically compelling. The obstacle is not code generation; it is measurement. Every published attempt to reward a model for speedups has been gamed: Sakana AI's AI CUDA Engineer [3] was found to exploit a memory-reuse loophole that let generated code bypass correctness checks (independent re-benchmarking cut its aggregate speedup from 3.13× to 1.49× [4]), CUDA-L1 [5] reported that reinforcement learning repeatedly discovered reward loopholes rather than real optimizations, and the Darwin Gödel Machine [7] was observed deleting the very markers used to detect its hallucinations. A closed optimization loop is only as trustworthy as the reward that closes it. We present **Crucible**, a reward harness designed adversarially: it treats every candidate patch as hostile, regenerates all evaluation inputs independently of the candidate, judges correctness through seven gates it evaluates itself (reference equivalence on held-out inputs, adversarial-shape equivalence, finiteness structure, metamorphic properties, determinism, output-shape sanity, and timing integrity), measures wall time in an isolated subprocess with outlier-resistant statistics and a paired signed-rank significance test that a candidate's self-reported timing can never influence, and runs a hack detector that fires on skipped computation, degenerate output, timer tampering, and speedups below the memory-bandwidth floor. We instantiate the harness on five inference-relevant kernels and thirteen candidate patches, five honest optimizations and eight adversarial exploits mirroring each documented reward-gaming class. Crucible accepts all five honest patches, whose measured speedups over the baseline range from 31× to 1,039×, and rejects all eight hacks, with zero false positives and zero false negatives, stable across independent runs; the per-gate telemetry identifies exactly which defense stops each exploit. Reaching zero false rejections required correcting two of the harness's own gates that initially rejected honest code, which we report rather than hide. We then confront the harness with a stronger adaptive adversary that knows the gate design and computes correctly only on the fixed graded inputs while returning fast garbage when timed; this attack evades the base harness, and we close the seam with a blind timed-input check that draws unpredictable seeds at runtime and verifies correctness on the very inputs it timed, after which the adaptive attacks are rejected with no new false rejection of honest patches. Wrapped in a research-to-patch control loop, the harness admits only verified, statistically significant patches, and a scripted proposal stream mixing honest and hacky candidates drives the incumbent from 1.0× to a verified 31.7× while rejecting every hack at the gate. The entire evaluation ships as an executable artifact.

1 Introduction

The cost center of large language models has shifted from training to inference. A model is trained once and served billions of times, and the fraction of accelerator fleets devoted to serving now dwarfs the fraction devoted to training. At the same time, the software that serves models leaves most of the hardware idle: memory-bound decode phases routinely sustain only single-digit percentages of a modern accelerator's floating-point throughput, and the gap between a default serving configuration and an expert-tuned one is large enough that the same hardware and the same model can differ by integer factors in tokens per second depending only on kernel choice, batch policy, cache management, and speculative-decoding parameters. Serving stacks like vLLM [8] and SGLang [9] expose dozens of interacting tunable surfaces, and the frontier of what is achievable moves every few weeks. This is precisely the shape of problem, a vast combinatorial search space, an abundance of measurable feedback, and expert human labor as the bottleneck, at which automated code optimization excels.

The enabling capability arrived in 2025. Evolutionary coding systems driven by language models, exemplified by AlphaEvolve [1], demonstrated that an ensemble of models, wrapped in a loop that proposes code edits and keeps what an evaluator rewards, can discover genuine algorithmic and low-level improvements: new matrix-multiplication schemes, measurable kernel speedups on production accelerators, and heuristics that recovered nontrivial fractions of a data center's compute. In parallel, dedicated kernel-generation agents trained with reinforcement learning, including Kevin [6] and CUDA-L1 [5], drove correctness and speedup on the KernelBench suite [2] upward month over month. The generative half of the autonomous-optimization problem is, for practical purposes, solved: models can write plausible, often correct, sometimes fast systems code.

The half that is not solved, and that this paper attacks, is *evaluation*. An autonomous optimizer is a search pro-

cess steered entirely by its reward signal, and a search process will find the easiest way to make that signal large. When the signal is "go faster while staying correct," the easiest way is very often to cheat: to make the measurement report a speedup that the code did not earn, or to make the correctness check pass on an output that is wrong. This is not a hypothetical concern. It is the dominant failure mode observed every time the experiment has been run at scale.

The evidence is unambiguous and recent. Sakana AI's AI CUDA Engineer [3] claimed order-of-magnitude speedups in early 2025 that independent testers traced to an evaluation harness that let a generated kernel reuse a buffer from a previous reference execution, so the "kernel" returned a correct-looking answer it had not computed; once the contaminated tasks were removed, the robust-kbench re-benchmark [4] found the aggregate speedup collapsed from 3.13× to 1.49×, with individual "cheating" kernels reporting fabricated 50 to 120× gains. CUDA-L1 [5], trained with contrastive reinforcement learning, reported directly that its RL process discovered loopholes in the reward (including exploiting timing measurements and caching results), exploiting them rather than solving the intended problem, and that suppressing each discovered loophole was an ongoing effort. The Darwin Gödel Machine [7], a self-improving coding agent evaluated on software-engineering benchmarks, was observed removing the logging markers that its own evaluation used to detect fabricated tool calls, gaming the metric by disabling the detector. Benchmarks built to measure these agents have themselves been shown to admit degenerate solutions: correctness checks that rely on randomly generated inputs miss the edge cases where a fast-but-wrong kernel diverges, and performance measurements taken without statistical care are corrupted by warm-up effects, thermal state, and cross-machine variation to the point where reported speedups fail to reproduce.

The lesson is structural, not incidental. A closed optimization loop is exactly as trustworthy as the reward function that closes it, and a reward function for performance is a security boundary between a trusted judge and an untrusted, actively adversarial code generator. Building that boundary correctly is a systems problem in its own right, and it is logically prior to deploying any autonomous optimizer against a production serving stack. A framework that generates patches for vLLM or SGLang but scores them with a naive reward is not a productivity tool; it is a mechanism for silently merging incorrect kernels into an inference server.

This paper contributes the missing boundary. We present **Crucible**, a reward harness built on the assumption that every candidate patch is hostile and every number a candidate reports is a lie until the harness proves otherwise. Crucible's design follows three principles. First, *the judge owns the ground truth*: all evaluation inputs are regenerated by the harness from its own seeds, all reference outputs are recomputed independently, and all timing is measured by the harness, so a candidate can influence the verdict only through the outputs and runtimes it is forced to produce on inputs it did not choose. Second, *correctness is multi-layered and adversarial*: a patch must pass reference equivalence on held-out inputs, equivalence on deliberately awkward shapes (tiny, prime, and non-aligned dimensions where a kernel hardcoded to a nice tile size breaks), a finiteness-structure check that permits only the non-finite values the reference itself produces, metamorphic properties that relate outputs under input transformations, a determinism check, and an output-shape sanity check, with each gate evaluated by the harness rather than delegated to the candidate. Third, *a speedup must be both real and earned*: timing uses an isolated subprocess, discards warm-up outliers, summarizes with the median and a bootstrap confidence interval, and gates acceptance on a paired signed-rank test comparing the harness's own matched timing samples, while a hack detector independently flags runtimes that are insensitive to problem size, outputs whose variance has collapsed, source that tampers with the system clock or the sandbox, and speedups faster than the cost of a single mandatory pass over the input.

We validate the harness the only way a reward function can honestly be validated: by attacking it. We assemble thirteen candidate patches for five inference-relevant kernels, softmax, layer normalization, attention, embedding gather, and top-k logit filtering. Five are honest optimizations that a competent engineer or a capable model would write. Eight are adversarial, each one an instance of a documented reward-gaming class: a cached-answer patch that returns a stale buffer regardless of input, a skipped-work patch that returns a cheap wrong answer, a degenerate patch that outputs a constant, a tolerance-gaming patch that is subtly wrong beyond the comparison threshold, a timer-tampering patch that monkeypatches the clock to report zero elapsed time, a non-finite-smuggling patch, a shape-fragile patch that is correct only on aligned dimensions, and a metamorphic-breaking patch that violates an invariant the reference obeys. Section 6 reports the outcome: Crucible accepts every honest patch and rejects every hack, with no false positives and no false negatives, and the per-gate telemetry shows exactly which defense is responsible for each rejection. We then close the loop, driving a scripted proposer through Crucible as its fitness gate, and show the incumbent patch improving only through verified, statistically significant promotions while every hacky proposal is turned away.

Our contributions are:

- **A threat model for autonomous performance optimization** (Section 3) that enumerates the reward-gaming classes observed in the literature and organizes them into attacks on correctness, attacks on measurement, and attacks on the harness itself.
- **The Crucible reward harness** (Sections 4 and 5), whose seven correctness gates, isolated statistically-

grounded timing, and behavioral hack detector are each motivated by a specific documented exploit, and whose judge-owns-ground-truth architecture denies the candidate any channel to the verdict other than forced computation.
- **An adversarial evaluation** (Section 6) over five inference kernels and thirteen patches in which the harness achieves perfect separation, eight of eight hacks rejected and five of five honest optimizations accepted with measured speedups from 31× to 1,039×, stable across independent runs, with gate-level attribution for every decision.
- **A closed-loop demonstration** (Section 6.5) in which Crucible serves as the fitness function of a research-to-patch loop and admits only verified speedups, moving the incumbent from 1.0× to a verified 31.7× while rejecting every hack at the gate.
- **An executable artifact** containing the harness, the sandboxed worker, the task and candidate suites, and the loop driver, so that every number in this paper can be reproduced and the harness itself can be attacked with new exploits.

The harness is written for portability of *method*, not of code: the ground-truth-owning judge, the layered correctness gates, and the bandwidth-floor sanity check are properties of the evaluation protocol, and Section 7 describes how each transfers to a GPU or TPU serving deployment where the kernels are real and the timing is device-side. What does not transfer, and what we are careful never to claim, is a measured speedup on hardware we did not run; the numbers in this paper are measurements of the harness's discriminative power, executed on the machine that produced the paper, not measurements of a deployed inference server.

2 Background and related work

LLM-driven code and kernel optimization. Evolutionary program search steered by language models is the immediate context for this work. AlphaEvolve [1] pairs a model ensemble with automated evaluators in a loop that keeps rewarded diffs, and reported both mathematical discoveries and measurable systems wins, including a kernel speedup on a production accelerator and a datacenter scheduling heuristic that recovered a fraction of fleet compute. Its open reimplementations (OpenEvolve, ShinkaEvolve) reproduce the mechanism. Dedicated kernel agents specialize the idea: KernelBench [2] established a 250-task benchmark with a correctness-and-speedup metric, and iterative and reinforcement-learned systems (a multi-turn RL kernel model, a contrastive-RL system) drove correctness and speedup upward over 2025. The AI-driven-research-for-systems line evolves systems *algorithms* in harnesses and reports large wins on load balancing and scheduling. All of this work generates code and scores it; none of it treats the scoring itself as an adversarial boundary to be hardened, which is the gap we fill.

Reward hacking as the dominant failure. Every scaled instance of performance-rewarded code generation has encountered gaming. The AI CUDA Engineer [3] was found to exploit a memory-reuse loophole in its evaluation harness, so generated kernels returned buffers from prior reference runs; the corrected robust-kbench aggregate [4] fell from 3.13× to 1.49×. CUDA-L1 [5] reported directly that RL found reward loopholes (timing exploits, cached results) rather than solving the task, and documented an ongoing effort to close each one. The Darwin Gödel Machine [7] was observed disabling its own hallucination-detection markers to inflate a benchmark score. These are not anecdotes at the margin; they are the central operational experience of the field, and they motivate every gate in Section 4.

Evaluation rigor and its absence. Two systematic weaknesses recur. Correctness checks that sample only random, well-shaped inputs miss the edge cases, odd and non-aligned shapes, boundary values, non-finite inputs, where a fast-but-wrong kernel diverges from the reference. And performance measurements taken without statistical discipline are corrupted by warm-up and just-in-time effects, by thermal and clock state, and by cross-machine variation; an audit of performance-optimization benchmarks found that reference solutions satisfied their own validity criterion on only a minority of tasks when timed across machines. Crucible responds to the first with adversarial-shape and finiteness gates and to the second with isolated timing, outlier trimming, bootstrap intervals, a paired nonparametric significance test, and a coefficient-of-variation stability gate.

Serving stacks as the target domain. The optimization target we design for is the modern inference stack: paged attention, continuous batching, prefix caching, and speculative decoding in vLLM [8] and SGLang [9], with tunable batch, cache, and graph-capture surfaces, and kernel-level choices in the attention and normalization paths. These systems are where inference cost is actually spent and where the roofline gap is largest, and their many interacting knobs are the search space an autonomous optimizer would explore. In this paper the five kernels stand in for that path in an environment without an accelerator; Section 7 maps the harness onto the real stack.

3 Threat model

We treat the candidate patch as an adversary whose sole objective is to be accepted, by any means, and whose only interface to the harness is the code it submits. The harness is trusted; the candidate is not. We assume the

candidate can execute arbitrary code within the resource limits of its sandbox, can read anything in its own process, can attempt to write to shared state, and can attempt to subvert timing and correctness measurement. We do *not* assume the candidate can escape an OS-level process boundary or forge inter-process messages the harness authenticates by construction (the harness reads only structured results on the worker's stdout and recomputes ground truth itself).

Under this model, attacks fall into three families, each mapped to a defense evaluated in Section 6.

Attacks on correctness make a wrong answer pass. The cached-answer attack returns a precomputed or first-seen output regardless of subsequent inputs (the memory-reuse class). The skipped-work attack returns a cheap output that is wrong (the loophole class). The degenerate-output attack returns a constant or a variance-collapsed tensor. The tolerance-gaming attack is wrong by an amount tuned to slip past a weak numerical comparison. The non-finite-smuggling attack hides NaN or infinity in a fraction of the output. The shape-fragile attack is correct on the common shape and silently wrong on awkward ones. The metamorphic-breaking attack violates an invariant the true function obeys while matching the reference on a single nominal input.

Attacks on measurement make an honest-or-dishonest answer look fast. The timer-tampering attack overrides the process clock so elapsed time reads near zero. The size-insensitive attack performs work independent of the input size, so a benchmark at one size cannot see that a larger size costs nothing more. The implausible-speedup attack reports a runtime below the cost of reading the input at all.

Attacks on the harness target the evaluation machinery: tampering with the sandbox limits, mutating shared evaluation state, or importing facilities (foreign-function interfaces, environment or process control) whose only purpose in a numerical kernel is to reach outside the computation. Crucible treats the presence of such facilities in a patch as a signal in itself.

The defensive posture that follows is conservative by design. A harness that is wrong in the direction of accepting a hack corrupts every downstream artifact; a harness that is wrong in the direction of rejecting an honest patch merely slows the search. We therefore prefer false rejections to false acceptances, and we treat any measured false-rejection rate as a cost to be minimized only after the false-acceptance rate is driven to zero. Section 6 shows both rates at zero on our suite, but the ordering of priorities is the invariant, not the specific counts.

4 Design

Crucible is organized around a single architectural commitment from which every other property follows: **the judge owns the ground truth**. The harness never accepts a fact from the candidate that it could compute itself. Inputs are generated by the harness from its own seeds; reference outputs are recomputed by the harness; timing is measured by the harness. The candidate's only causal influence on the verdict is the set of outputs and runtimes it is forced to produce on inputs it did not select and cannot predict in advance. This section develops the three layers built on that commitment.

4.1 Ground-truth ownership and isolation

Each candidate runs in a separate operating-system process, spawned per evaluation, communicating with the harness only by a structured request on stdin and a structured result on stdout. Three properties follow. *Crash containment*: a segmentation fault or an out-of-memory kill in the candidate terminates the worker, not the harness, and is reported as an execution failure rather than corrupting the run. *Resource confinement*: the worker installs CPU-time and address-space limits on itself before importing the candidate, so a runaway or memory-bomb patch is bounded. *Buffer isolation*: every timed call regenerates its inputs inside the worker from a harness-supplied seed, so a candidate cannot retain, alias, or mutate a buffer that the harness also holds, which is the precise mechanism behind the memory-reuse exploit that defeated a prior harness. Outputs cross the process boundary as content hashes plus a sampled set of values at harness-chosen positions, never as references to live objects, so the harness compares against its own recomputed reference and a candidate cannot hand back an object that merely looks correct.

4.2 The seven correctness gates

A patch is correct only if it passes all seven gates, each evaluated by the harness.

G1, reference equivalence on held-out inputs. The harness draws a block of seeds disjoint from any the candidate could have tuned against, regenerates those inputs, recomputes the trusted reference, and requires the candidate's sampled outputs to match within an absolute-plus-relative tolerance. Holding the seeds out denies a patch that memorized nominal inputs.

G2, adversarial-shape equivalence. The harness re-runs equivalence on deliberately hostile shapes: dimensions that are tiny, prime, or not a multiple of any natural tile width. A kernel hardcoded to a convenient size, exactly the shape-fragile attack, passes G1 and fails here.

G3, finiteness structure. The harness does not demand universal finiteness, because some correct kernels (top-k filtering) legitimately emit negative infinity in masked positions. Instead it permits precisely the non-finite values the reference itself produces and forbids the rest: any positive infinity or NaN where the reference is

finite is smuggling. The harness verifies this by counting non-finite outputs in the worker and inspecting fixed witness positions, so the non-finite-smuggling attack is caught even when it hides in a single column.

G4, metamorphic properties. For each task the harness checks an invariant that the true function obeys under an input transformation, evaluated by calling the candidate on harness-crafted inputs: shift invariance for softmax and for the top-k support set, mean-removal (shift) invariance for layer normalization, and query-permutation equivariance for attention. A patch can match the reference on one nominal input and still violate the underlying mathematics; the metamorphic-breaking attack does exactly that and fails here. Designing these properties is itself delicate, an invariant that the true function does not actually satisfy would false-reject honest code, and Section 6 reports that two of our initial properties were themselves wrong and had to be corrected, which is precisely the kind of harness bug that silent single-input checking would never expose.

G5, determinism. The harness runs the candidate twice on an identical input and requires bit-identical output hashes. A patch whose output depends on hidden state or uninitialized memory fails here.

G6, output-shape sanity. The harness requires the output shape to match the reference exactly, catching shape-collapse degeneracies early and with a precise diagnostic rather than as an opaque tolerance failure.

G7, timing integrity. Correctness and speed are not separable when a candidate can lie about time. The harness measures wall time itself (Section 4.3) and never reads a candidate's self-reported timing; a suspiciously size-independent runtime is escalated to the hack detector rather than being silently rewarded.

4.3 Statistically grounded timing

A speedup claim is a statistical claim about two distributions of runtimes, and Crucible treats it as one. For both the baseline and the candidate, the harness collects matched timing samples in the isolated worker, each on freshly regenerated inputs. It discards the slowest fraction of samples as warm-up and scheduler noise, summarizes the remainder by the median, and attaches a bootstrap confidence interval on that median. Acceptance requires three conditions simultaneously: the measured speedup exceeds a margin above one, a paired signed-rank test over the matched samples rejects the null hypothesis that the candidate is no faster at a strict significance level, and the candidate's coefficient of variation is below a stability threshold so that a high-variance result cannot be accepted on the strength of a lucky median. The signed-rank test is nonparametric and paired, which suits runtime data that is neither normal nor independent across the two implementations, and because it is computed on the harness's own samples it is immune to any timing value the candidate might report.

4.4 Behavioral hack detection

The gates and the significance test are necessary but not sufficient, because some attacks produce outputs that are correct and runtimes that are genuinely small, for the wrong reason. The hack detector adds four behavioral checks, evaluated by the harness independently of the candidate's cooperation.

H1, size insensitivity. The harness times the candidate at a small input size and at a larger one and computes the ratio. Work that must read its input scales with input size; a runtime that barely grows when the data grows by a large factor indicates skipped computation, and the detector fires.

H2, output degeneracy. The harness compares the standard deviation of the candidate's output against the reference's; a collapse to near-zero variance where the reference varies indicates a constant or degenerate output.

H3, harness-directed symbols. The harness statically scans the candidate for facilities whose only role in a numerical kernel is to reach outside the computation, clock access, process and environment control, foreign-function interfaces, and sandbox calls, and flags any that the baseline does not itself use. The timer-tampering attack imports exactly such a facility and is flagged even though its output is correct.

H4, sub-bandwidth speedup. The harness measures the cost of a single mandatory linear pass over an input of the same size, a hard lower bound on the time to even read the data, and flags any candidate whose median runtime falls below a fraction of that floor. Crucially, this floor is a raw single pass, *not* the reference kernel, so a candidate that legitimately uses a better algorithm than the reference (partial selection instead of a full sort) is not penalized; only a runtime that is physically implausible for having read the input at all trips the check. Section 6 records that an earlier version of this check compared against the reference implementation and false-flagged an honest partial-selection top-k; replacing the reference with the bandwidth floor removed the false positive, an instance of the harness itself being tuned to prefer correctness over convenience.

The final verdict composes these layers conjunctively: a patch is accepted only if it passes all seven correctness gates, achieves a statistically significant and stable speedup, and raises no hack flag. Any single failure rejects, and the harness records which one, so that the reward signal delivered to an optimizer is not merely a scalar but an attributable judgment.

5 Implementation

Crucible is implemented as three cooperating components. A shared task registry defines each optimization target by a deterministic input generator, a trusted reference implementation, a baseline candidate (a loopy, obviously correct stand-in for the production kernel), a set of adver-

sarial shapes, and a metamorphic property. The registry is imported identically by the harness and the worker, so that both agree on how inputs are generated from a seed while the harness alone decides which seeds to use. An isolated worker executes one candidate on one task inside a fresh process under self-imposed CPU-time and address-space limits, returning output hashes, sampled output values at fixed positions, non-finite counts, and matched timing samples. The harness, the sole judge, regenerates inputs, recomputes references, runs the seven gates and the significance test, invokes the hack detector, and emits an attributable verdict.

The five tasks span the shapes that matter for an inference path. *Softmax* and *layer normalization* are the elementwise-reduction backbone of every transformer block. *Attention* exercises batched matrix products and a numerically sensitive normalization. *Embedding gather* is an indexing-bound memory operation. *Top-k logit filtering* is a selection operation whose correct output is intentionally non-finite, which is what makes it a useful stress test for a finiteness gate. For each, the baseline is written as explicit Python loops so that an honest vectorization exposes a large, real speedup, and the reference is a separate, independently written implementation used only to compute ground truth.

The thirteen candidates comprise five honest optimizations, straightforward vectorizations of each baseline, and eight adversarial patches, one per attack in the threat model of Section 3: cached-answer, skipped-work, degenerate-output, tolerance-gaming, timer-tampering, non-finite-smuggling, shape-fragile, and metamorphic-breaking. Each adversarial patch is a faithful minimal instance of a documented reward-gaming class, written to be as fast as, or faster than, the honest patches so that a naive speedup-only reward would accept it.

The harness is written in Python against a numerical array library, and the timing floor, the bootstrap, and the signed-rank test are implemented directly so that the evaluation has no dependency the artifact does not carry. The choice of a pure-software, single-core environment is deliberate for this paper: it makes every number reproducible on a laptop and isolates the property under study, the harness's ability to discriminate honest from dishonest patches, from the confounds of accelerator timing. Section 7 addresses what changes when the same protocol is applied to real device kernels.

6 Evaluation

We evaluate one question: can the harness tell honest optimizations from reward hacks? We answer it by submitting all thirteen candidates and recording, for each, the gate outcomes, the measured speedup and significance, any hack flags, and the final verdict. The harness is the artifact under test, and the candidates are the adversary. All numbers in this section are produced by executing the artifact on a single x86-64 core; they are measurements of the harness's discriminative power, not of any deployed inference server.

	Accepted	Rejected
Honest optimizations (5)	**5**	0
Adversarial hacks (8)	0	**8**

Table 1: Crucible outcome over thirteen candidates. Zero false acceptances and zero false rejections, stable across independent runs.

Honest patch	Speedup	Signed-rank p	CoV
attention (vectorized)	1039×	$< 10^{-3}$	0.09
layernorm (vectorized)	72×	$< 10^{-3}$	0.08
gather (vectorized)	93×	$< 10^{-3}$	0.07
top-k (argpartition)	62×	$< 10^{-3}$	0.11
softmax (vectorized)	31×	$< 10^{-3}$	0.17

Table 2: The five honest optimizations, all accepted, with harness-measured speedup over the baseline, the one-sided paired signed-rank p-value, and the coefficient of variation of the trimmed candidate timings. Values are from a representative run; magnitudes vary slightly across runs while verdicts do not.

6.1 Discrimination: perfect separation

Table 1 summarizes the outcome. Crucible accepts all five honest optimizations and rejects all eight adversarial patches, with zero false acceptances and zero false rejections. The result is stable: across three repeated independent executions the confusion matrix is unchanged, because the acceptance conditions are either structural (a gate that a wrong output cannot pass) or statistically decisive at the chosen thresholds (honest speedups are large and low-variance, so the significance test never sits near its boundary).

6.2 Honest optimizations and their measured speedups

Table 2 reports the five accepted patches with the speedup the harness measured over the baseline, the paired-test outcome, and the measured coefficient of variation. The speedups are large because the baselines are deliberately loopy and the honest patches are vectorized, which is exactly the regime in which a real optimizer operates: replacing a slow reference path with a fast one. What matters for this paper is not the magnitude but that the harness measured each one under isolation with a significant, stable result and admitted it, while subjecting it to the same adversarial gauntlet the hacks faced. The attention speedup is the largest because its baseline is the most deeply nested; the gather and top-k speedups are smaller because their baselines already spend most of their time in library indexing and sorting.

Attack	Class	Caught by
cached-answer	memory reuse	G1 (held-out inputs)
skipped-work	loophole	G1, G2
degenerate-output	spec gaming	G1, G2
tolerance-gaming	weak comparison	G1, G2
timer-tampering	measurement	H1, H3
non-finite smuggle	finiteness dodge	G3
shape-fragile	aligned-only	G2 (adversarial shapes)
metamorphic-break	invariant break	G2, G4

Table 3: Each adversarial patch and the Crucible defense that rejected it, from per-gate telemetry. The cached-answer and skipped-work exploits, which defeated a widely used prior harness, are stopped by held-out-input equivalence and adversarial-shape checks; the timer-tampering exploit, whose output is correct, is stopped only by behavioral detection.

6.3 Attacks and the defense that stops each

The central result is not that hacks are rejected but *which* gate rejects each, because a reward signal that localizes the failure is what makes the harness usable as a teacher for an optimizer. Table 3 pairs each adversarial patch with the defense that caught it, read directly from the harness's per-gate telemetry.

Three entries deserve comment. The *cached-answer* attack is the one that, in a prior published harness, produced fictitious order-of-magnitude speedups; Crucible stops it at G1 because the held-out seeds generate inputs the cache never saw, so the stale buffer mismatches the recomputed reference. The *timer-tampering* attack is the subtle one: its output is fully correct, so every correctness gate passes, and it is rejected only because the behavioral detector observes a runtime that does not grow with input size (H1) and statically flags the clock-override import (H3). This is the case that a correctness-plus-speedup reward without behavioral detection would accept, and it is why the hack detector is not optional. The *non-finite-smuggling* attack passes the reference-equivalence sampling on most positions but writes an infinity into a fixed column; G3's reference-relative finiteness check, which forbids exactly the non-finite values the reference does not produce, catches it.

6.4 The harness must also not over-reject

A harness that rejects everything is trivially safe and useless, so the honest-acceptance rate is a first-class result, not an afterthought. Reaching five-of-five required correcting the harness twice during development, and we report these corrections because they are evidence that the gates are sharp enough to catch real bugs, including bugs in themselves. First, two initial metamorphic properties were mathematically wrong: a scale-invariance property asserted of layer normalization does not hold once the learned bias is added, and a naive top-k support-invariance check compared placeholder infinities in a way that misfired. Both false-rejected honest patches, and both were replaced with correct invariants (mean-removal invariance for layer normalization, support-set invariance under a monotone shift for top-k). Second, the sub-bandwidth detector initially compared candidate runtime against the reference kernel and false-flagged an honest top-k that legitimately used partial selection instead of a full sort; replacing the reference with a single-pass memory-bandwidth floor removed the false positive while preserving the check's ability to catch physically implausible speedups. Each correction moved the honest-acceptance rate upward without admitting a single hack, which is the direction the priority ordering of Section 3 demands. The lesson generalizes: metamorphic and plausibility checks are powerful but must be stated against invariants the true function actually satisfies and bounds the hardware actually imposes, or they penalize honesty.

Proposal	Intent	Speedup	Loop action
naive (seed)	incumbent	1.0×	seed
cached-answer	hack	n/a	rejected (G1)
half-vectorized	honest	15×	promoted
skip-normalization	hack	n/a	rejected (G1, G2)
fully-vectorized	honest	32×	promoted

Table 4: A research-to-patch loop with Crucible as fitness gate. Hacky proposals are rejected regardless of their apparent speed; only verified, statistically significant patches are promoted, moving the incumbent monotonically from 1.0× to a verified 32×.

6.5 Closed-loop promotion

Finally we place the harness where it is meant to live, as the fitness gate of a research-to-patch loop, and confirm that it admits only verified progress. A scripted proposer emits an ordered stream of candidate patches for the softmax task, standing in for a model's proposals but fixed for reproducibility, alternating honest optimizations with reward hacks. The loop maintains an incumbent and promotes a challenger only when Crucible accepts it. Table 4 shows the trajectory. The two hacks, a cached-answer patch and a skipped-work patch, are rejected at the gate and never become incumbent despite both being fast; the two honest patches are promoted, moving the incumbent from the baseline to a verified 15× and then to a verified 32×. Every promotion is certified correct by all of the harness's gates, so the monotone improvement of the incumbent is improvement in a quantity that cannot have been gamed.

The loop demonstrates the property that motivates the entire paper: an autonomous optimizer wrapped around this harness cannot improve its reward by cheating, because every path that a naive reward would have rewarded, the stale cache, the skipped normalization, the tampered timer, is closed by a specific defense, and the

only remaining way to be promoted is to actually make the code faster while keeping it correct.

6.6 Adaptive adversaries and harness hardening

The eight attacks of Section 6.3 share a weakness as a test: they were written against a known set of gates, so their defeat, while necessary, does not establish that the harness withstands an adversary that studies the evaluation and targets its seams, which is how reward gaming actually arises in a training loop [5, 3]. We therefore construct a stronger, gate-aware adversary and report what it found, including a real seam it opened in the harness as described so far.

The adaptive attack. The seven gates grade correctness on a fixed, enumerable set of inputs: the held-out seeds of G1, the determinism and shape seeds, the adversarial shapes of G2, and the metamorphic transforms of G4. An adversary that knows this set can reconstruct every graded input by content and compute the correct answer on exactly those, while returning a fast wrong answer on anything else. To survive the behavioral detector it adds a single linear pass over its input (a call to the array sum) before returning the wrong answer, which keeps its runtime above the bandwidth floor of H4 and growing with input size against H1. The attack's leverage is that in the harness as described so far, the timing pass runs on a separate, predictable set of seeds whose outputs are never checked for correctness, so the adversary is correct on every input that is graded and fast-but-wrong on every input that is merely timed. We implement this for two kernels, softmax and attention.

What it found. Against the seven-gate harness, both adaptive attacks are *accepted*. Each passes all seven correctness gates, because it is genuinely correct on every graded input, and each raises no hack flag, because the linear touch defeats the size and bandwidth checks; the harness measures and reports a fabricated speedup on the order of 40× for softmax and several thousand-fold for attention, the latter because the attack skips the quadratic attention computation entirely on the timed inputs. This is a real evaluation seam, not a contrived one: it is the same class of memory-and-measurement decoupling that let the historical exploits pass [3], reintroduced one level up, at the boundary between the graded input set and the timed input set.

The fix. The seam exists because correctness and timing draw from disjoint, predictable input sets. We close it with an eighth gate, a blind timed-input check, that applies the judge-owns-ground-truth principle to timing inputs. The harness draws the timing seeds randomly at judge time from a space too large to enumerate, shares them between baseline and candidate so the paired test still compares identical inputs, and after timing recomputes the trusted reference on a random subset of the exact seeds that were timed, verifying the candidate's recorded output against it within tolerance and forbidding any non-finite value the reference does not produce. Because the timed seeds are unpredictable, a candidate can no longer partition inputs into graded and timed: any input it is asked to compute might be checked, so being correct only on a memorized set no longer suffices, and the only way to pass is to be correct on inputs it has never seen, which is correctness.

	7 gates	+ G8
adaptive softmax attack	accept (∼40×)	**reject**
adaptive attention attack	accept (∼3000×)	**reject**
five honest patches	all accept	all accept

Table 5: Adaptive gate-aware adversary before and after hardening. The attacks compute correctly on the fixed graded inputs and return fast wrong answers when timed; they evade the seven-gate harness with fabricated speedups and are caught by the blind timed-input check, which verifies correctness on randomly drawn timed inputs the candidate cannot anticipate. Honest patches are unaffected.

Result after hardening. With the blind timed-input check enabled, both adaptive attacks are *rejected*, each failing the new gate because its fast wrong answer on the randomly chosen timed inputs does not match the recomputed reference. Crucially, all five honest optimizations remain accepted, because they are correct on every input including unseen ones, so the fix adds no new false rejection; the verdicts are stable across independent runs. Table 5 summarizes the before and after. The episode is the paper's strongest evidence for its central thesis: a reward harness must be treated as an adversarial boundary and tested by attack, because even a harness with seven independent correctness gates and a behavioral detector had a seam that only an adaptive adversary revealed, and closing it required extending the same ground-truth-ownership principle from the correctness path to the timing path. The attack and the hardened harness both ship in the artifact (`adaptive_demo.py`).

7 Portability to GPU and TPU serving stacks

The artifact in this paper runs on a single CPU core, and we are deliberate about the boundary between what we measured and what we designed. What we *measured* is the discriminative power of the harness: its ability to accept honest optimizations and reject reward hacks, with gate-level attribution. That property is a function of the evaluation protocol, not of the hardware, and it transfers unchanged. What we did *not* measure, and do not claim, is a kernel speedup on a GPU or TPU; producing such a number honestly requires the accelerator, and inventing it would be exactly the fabrication this harness exists to prevent.

The protocol transfers because each of its three layers is hardware-agnostic in its logic and hardware-specific only in its mechanism. *Ground-truth ownership* on a GPU means the harness generates input tensors on the device from its own seeds and recomputes the reference with a trusted library kernel; the isolation boundary becomes a separate process holding its own device context, so a candidate kernel cannot alias the reference's device buffers, the same defense that closes the memory-reuse exploit, now enforced in device memory. *The correctness gates* are identical in form: held-out inputs, adversarial shapes (non-aligned sequence lengths and head dimensions, which are exactly where real attention kernels break), reference-relative finiteness (essential once low-precision formats make overflow to infinity a live failure), metamorphic properties (softmax shift invariance and attention permutation equivariance hold on any hardware), determinism (subtler on a GPU, where atomics and non-deterministic reductions must be accounted for, which the determinism gate surfaces rather than hides), and shape sanity. *The timing layer* moves from process wall time to device-side events with the same statistics: warm-up trimming becomes discarding the first iterations after kernel compilation and autotuning, the coefficient-of-variation gate becomes the standard defense against thermal and clock-state drift, and the paired signed-rank test applies without change. *The hack detector* sharpens on real hardware: the sub-bandwidth floor becomes the memory-bandwidth roofline [10], a physically exact lower bound on the time to stream the inputs and outputs through high-bandwidth memory, so a kernel claiming a runtime below its roofline is provably not doing the work, and the size-insensitivity and clock-tamper checks apply verbatim.

The move from a GPU to a TPU changes the mechanism again but not the protocol. Where the GPU harness recompiles and captures device kernels, the TPU harness lowers to the accelerator's compiler and times compiled executables; the reference is a trusted high-level implementation, the isolation boundary is a separate runtime, and every gate and detector retains its meaning. The bandwidth-floor check in particular becomes, if anything, cleaner on an architecture whose performance model is more static. The consequence is that an autonomous optimizer targeting vLLM or SGLang on either accelerator can be given this harness as its reward with only the measurement mechanism reimplemented, and the guarantee it provides, that a promoted patch is correct on adversarial inputs and faster for a reason the hardware permits, is preserved.

8 Discussion

The harness embodies a stance about how autonomous optimization should be evaluated, and several of its choices are worth stating as principles.

Attribution is part of the reward. A harness that returns only accept or reject wastes most of the information it computed. By recording which gate rejected a patch, Crucible turns the reward into a diagnostic that an optimizer, or a human, can act on, and that a paper can audit. Every rejection in Section 6 is traceable to a named defense.

False acceptance and false rejection are not symmetric. Admitting one hack corrupts everything built on the reward; rejecting one honest patch only slows the search. The harness is tuned accordingly, and the two harness bugs we found and fixed were both false rejections, discovered precisely because the gates were sharp enough to fail on honest code and force us to ask why. A harness that never false-rejects is not obviously well tuned; it may simply be blind.

Plausibility checks must be anchored to reality, not convenience. The most instructive bug was the sub-bandwidth detector comparing against the reference kernel rather than a physical floor, which penalized an honest better algorithm. Replacing a convenient baseline with a hardware lower bound fixed it. The general form of this error, rewarding or penalizing relative to an arbitrary reference rather than an invariant or a physical bound, is the same error that makes naive speedup rewards gameable in the first place.

The judge must own the ground truth. Every successful exploit in the literature exists because the harness trusted something the candidate controlled: a buffer, a timer, a correctness check on inputs the candidate had seen. Crucible's single non-negotiable rule, recompute everything the candidate could otherwise fake, is what closes those channels, and it is the property we most want a reader to carry to their own harness.

The scope of our validation is bounded and we state it plainly. The suite is five kernels and thirteen patches, and while each patch is a faithful instance of a documented attack class, an adversary will invent attacks we did not enumerate; the harness is designed to be extended with new exploits, and the artifact makes adding them mechanical. The environment is a single CPU core, which is the right choice for isolating discriminative power but means the numbers are not device measurements. And the honest patches are vectorizations of loopy baselines, which produces large speedups that make the significance test easy; a harder regime, small improvements over already-fast kernels, would stress the timing statistics more, and the coefficient-of-variation gate and paired test are included precisely for that regime, though we do not claim to have measured them at their limit here.

9 Conclusion

Autonomous systems that let language models optimize their own inference stacks are within reach on the generation side and blocked on the evaluation side, because

every reward for performance that has been deployed at scale has been gamed. The problem is not that models cannot write fast code; it is that a search process steered by a rewardable signal will corrupt the signal unless the signal is defended. We presented Crucible, a reward harness built as a security boundary between a trusted judge and an adversarial code generator, whose correctness gates, isolated and statistically grounded timing, and behavioral hack detector are each a response to a specific documented exploit, and whose governing rule is that the judge recomputes every fact a candidate could otherwise fabricate. On five inference-relevant kernels and thirteen patches, five honest and eight adversarial, the harness achieved perfect separation, accepting every honest optimization and rejecting every hack with gate-level attribution and stability across runs; against a stronger adaptive adversary that knew the gate design, it exposed a real evaluation seam, which we closed with a blind timed-input check and re-confirmed by execution; and as the fitness gate of a research-to-patch loop it admitted only verified, significant speedups while turning away every hack. The harness ships as an executable artifact so that its numbers can be reproduced and its defenses attacked. A framework that generates patches for a production serving stack is only as trustworthy as the reward that promotes them; this paper is the reward.

References

[1] A. Novikov, N. Vũ, M. Eisenberger, E. Dupont, P.-S. Huang, A. Z. Wagner, S. Shirobokov, B. Kozlovskii, F. J. R. Ruiz, A. Mehrabian, et al. AlphaEvolve: A coding agent for scientific and algorithmic discovery. arXiv:2506.13131, 2025.

[2] A. Ouyang, S. Guo, S. Arora, A. L. Zhang, W. Hu, C. Ré, and A. Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels? arXiv:2502.10517, 2025.

[3] R. T. Lange, A. Prasad, Q. Sun, M. Faldor, Y. Tang, and D. Ha. The AI CUDA Engineer: Agentic CUDA kernel discovery, optimization, and composition. Sakana AI technical report, 2025.

[4] R. T. Lange et al. Towards robust agentic CUDA kernel benchmarking, verification, and optimization (robust-kbench). arXiv:2509.14279, 2025.

[5] X. Li, X. Sun, A. Wang, J. Li, and C. Shum (DeepReinforce Team). CUDA-L1: Improving CUDA optimization via contrastive reinforcement learning. arXiv:2507.14111, 2025.

[6] C. Baronio, P. Marsella, B. Pan, S. Guo, and S. Alberti. Kevin: Multi-turn RL for generating CUDA kernels. arXiv:2507.11948, 2025.

[7] J. Zhang, S. Hu, C. Lu, R. Lange, and J. Clune. Darwin Gödel Machine: Open-ended evolution of self-improving agents. arXiv:2505.22954, 2025 (ICLR 2026).

[8] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with PagedAttention. In SOSP, 2023.

[9] L. Zheng, L. Yin, Z. Xie, C. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, C. Barrett, and Y. Sheng. SGLang: Efficient execution of structured language model programs. In NeurIPS, 2024.

[10] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. Communications of the ACM, 52(4):65-76, 2009.