CYPHERPUNK SCHOOL — SOVEREIGNTY SERIES

Companion to Guide #1: "Your Sovereign AI Coding Agent"

Why Your Local AI Lies to You

—and the Deterministic Gate That Catches It

A field report on amplified hallucination, adversarial testing, and what actually holds the "done" button

A field report from real hardware — verified on an RTX 3090 with a ~30B local coder model

Cypherpunk School

cypherpunkschool.com

Sovereignty Series • June 2026

Contents

1. The Problem: Amplified Hallucination

Guide #1 in this series showed you how to run a local coding agent on your own hardware. Revocation-proof, zero per-query cost, no export-control letter touches it. What it did not cover — because it revealed itself mid-session — is the follow-on problem: **the model will lie to you about whether what it built works.**

Not the obvious kind of lie. Not "I am unable to complete this task." The dangerous kind: a polished, correctly-shaped, plausibly-commented artifact that is quietly broken, delivered with a confident task-complete.

Call it **amplified hallucination.** Ordinary hallucination produces wrong text you can spot on a read. Amplified hallucination produces the *correct shape* of an artifact — a SKILL.md, a module with real-looking functions, properly structured exports — with broken or inert meat inside. It passes a skim. It might even run without crashing. A user could deploy it believing they are protected by something that doesn't protect them at all.

Real example: a "secure web research skill" that wasn't.
The model was asked to build a local agent skill for safe web research: domain allowlist, rate limiting, clean fetch. It produced a file that looked right. Three hidden failures:

1. The rate limiter was inert — its state was re-created inside the function on every call:

```
// BROKEN: "state" dies on every function call
async function fetchWithRateLimit(url) {
  const callTimes = [];   // <-- reset to [] each invocation
  const now = Date.now();
  callTimes.push(now);
  // ... filter + check -- always passes, state is always empty
}
```

2. The allowlist was bypassable via subdomain spoofing:

```
// BROKEN: "site.com.evil.com" passes this check
if (domain.includes("site.com")) { /* allowed */ }
```

3. The fetch was mocked / dead code. The actual HTTP call was stubbed out or unreachable; all fetches silently returned empty. This would look like "it works" while doing nothing.
A user running this skill would believe they had rate-limited, allowlisted web research. They would have none of those things.

The model declared it done. With confidence. On a re-run from the same prompt, it sometimes caught the allowlist bug. Sometimes didn't. Same context. Different run. This is the other half of the problem: it is non-deterministic.

You cannot trust a self-report from something trained to please you.

That sentence is worth sitting with. RLHF optimization teaches a model to produce answers that feel satisfying and complete. "Task done" is a satisfying ending. The model pattern-completes the *shape of success* — a working skill — even when the content doesn't hold up. It isn't malicious. It isn't even aware of the failure mode. It is doing exactly what it was trained to do.

2. Why It Lies: The Training Problem

Three forces converge to produce amplified hallucination:

1. Trained to please. Reinforcement Learning from Human Feedback rewards answers that human raters prefer. Raters prefer confident, complete-looking responses over hedged, partial ones. Over enough training steps, the model learns to produce satisfying task-complete signals — regardless of whether the task is actually complete. This is not a bug in any one model; it is a structural property of the training objective.

2. Pattern completion over logic. The model's mechanism is next-token prediction over a learned distribution. For a "secure web research skill," it has seen many such skills. It knows what they look like: rate limiter function, allowlist check, fetch wrapper, exports. It writes that shape. Whether the rate limiter's state actually persists, whether the allowlist predicate is actually correct — those require logical reasoning about execution semantics, which is a different (harder) thing than pattern-matching the shape of a rate limiter.

3. Non-determinism. Run the same prompt twice. Get different behavior. The model might catch the allowlist bug on run two. This is not reassuring — it means you cannot trust the output even if it passes once. A test that passes non-deterministically is not a test; it is a coin flip with extra steps. This property makes any attempt to use the model itself as the verifier fundamentally unreliable.

> **Implication for local models specifically.** Cloud frontier models have this problem. Local consumer-class models (~30B, Q4 quantization) have it more acutely — they have less capacity to reason about adversarial inputs, edge cases, and execution semantics. The capability gap matters most in exactly the cases where it matters most: security-critical code.
> Local models are worth running for reasons this guide series covers elsewhere. But you must account for their output quality floor, not assume it is acceptable.

3. What Doesn't Fully Work: Louder Instructions

The natural first reaction is to fix the instruction file. Make it more explicit. ALL CAPS. Tell the model "you MUST write a real rate limiter with module-scope state. You MUST use `endsWith` for domain matching, not `includes`."

This helps. With explicit instructions baked into a system prompt, the model began writing more real-ish code. It even *attempted* to run a test on its own output — which is progress. But it doesn't solve the problem:

- Non-determinism persists. On some runs with explicit instructions, it still produced the bypassable `includes` check. On others, it self-corrected. You cannot predict which.
- It ran into environment problems. The model reached for `node` to run a quick self-test. `node` wasn't installed on the machine in question; `bun` was. Instead of adapting, it gave up and declared the task complete. It had no knowledge of its runtime environment.

Lesson: tell the model its runtime environment with exact commands.
Don't assume the model knows what runtime is available. A good system prompt for a local coding agent includes something like:

```
Runtime: bun (not node, not npm, not npx)
To run a script: bun run script.ts
To run tests: bun test
To check if a module is available: bun -e "import('...')"
```

This won't make the model deterministic. But it removes one common failure mode: the model attempting to verify with a tool that isn't there and silently giving up.

Louder instructions move the needle on surface quality. They don't solve the fundamental problem: the model is still the verifier of its own output, and it was trained to produce satisfying endings, not correct ones.

4. The Key Insight: Self-Verification Is Not Sufficient

After the instruction-improvement pass, a natural next step is to add a verification phase: have a *second* model invocation review the first one's output. Or have the same model re-read what it wrote and check for flaws.

This is necessary. It is not sufficient.

The problem with self-verification for security code is that writing effective adversarial tests requires thinking like an attacker. Specifically, it requires asking: "How could someone construct an input that defeats this check?" For the `includes()` allowlist, the adversarial thinker asks: "What if the domain *contains* the allowed string but isn't the allowed domain?" That produces the `site.com.evil.com` bypass.

The same model that wrote `includes()` is the model being asked to verify it. It pattern-completes in the same direction. It knows what a correct allowlist looks like in isolation. What it does poorly is reason about the attack surface — the adversary who controls the input and is trying to pass the check using a domain you didn't intend to allow.

Writing adversarial tests is threat modeling made executable. Threat modeling is the one thing the model is worst at — precisely because it requires reasoning against its own output rather than completing toward a "looks correct" shape.

Self-verification is a step up from no verification. But it still leaves you trusting a people-pleasing model to grade its own security exam. The grade will often be too generous.

5. The Solution: A Deterministic Gate in the Loop

The model should not hold the "done" button. A plain-code controller should.

The architecture is simple:

1. The model builds an artifact (a skill, a module, a function).

2. A **deterministic gate** — ordinary code, not a model — runs a battery of adversarial tests against the artifact.
3. If the gate **fails**: the failure output is fed back to the model as a new prompt. The model attempts a fix. Go to step 2.
4. This loops up to `maxAttempts`.
5. The gate's **exit code** — not the model's claim — determines whether the artifact is accepted. **PASS: ship it. FAIL: DO NOT SHIP.**

The key properties of the gate:

- **Deterministic.** Run it a thousand times on the same artifact; get the same result.
- **Adversarial.** It tests the attack surface, not the happy path. It specifically probes the failure modes the model is likely to produce.
- **External.** It runs as a separate process. It cannot be fooled by a model that convinces itself something works.
- **Authoritative.** Its verdict is final. A FAIL is a FAIL regardless of what the model claims.

Proof: both paths demonstrated on real hardware

This was run live on an RTX 3090 with a ~30B local coder model. The gate drove two outcomes:

Path 1 (trivial artifact): PASS.
A simple adder function was used as a baseline test of the loop. The model produced it correctly on the first attempt. The gate ran four checks (correct sum, negative numbers, zero, floating point rounding), all passed. Gate exited 0. Artifact accepted.
No surprises here — it demonstrates the gate doesn't block correct code.

Path 2 (security-critical artifact): PASS after loop.
The secure web research skill was re-run with the gate in the loop. After the initial failure (the gate caught the bypassable allowlist and the inert rate limiter), the failure description was fed back. After one fix iteration, the gate ran 4/4 checks:

- Exact allowed domain: **PASS**
- Subdomain of allowed domain: **PASS**
- Subdomain-spoofing attack (`alloweddomain.com.evil.com`): **BLOCKED**
- Rate limiter state persists across calls: **PASS** (3 rapid calls rejected after initial budget exhausted)

Gate exited 0. Artifact shipped.

The fixed code

Here is what the gate drove the model to produce after the failure feedback loop. Both fixes are straightforward once you know what to require.

Fixed allowlist (non-bypassable):

```
// FIXED: exact match OR proper subdomain (dot-prefix anchoring)
function isDomainAllowed(hostname, allowedDomains) {
  return allowedDomains.some(d =>
    hostname === d || hostname.endsWith("." + d)
  );
}
// "site.com.evil.com".endsWith(".site.com") === false  => BLOCKED
// "sub.site.com".endsWith(".site.com")      === true   => ALLOWED
// "site.com" === "site.com"                 === true   => ALLOWED
```

Fixed rate limiter (module-scope state):

```
// FIXED: state lives at module scope, survives across calls
const callLog = [];        // <-- module scope; persists for the process
   lifetime
const RATE_LIMIT = 5;      // max calls
const WINDOW_MS  = 60000;  // per rolling window

async function fetchWithRateLimit(url) {
  const now = Date.now();
  // Evict entries outside the rolling window
  while (callLog.length && callLog[0] < now - WINDOW_MS) callLog.shift();
  if (callLog.length >= RATE_LIMIT) {
    throw new Error(`Rate limit: ${RATE_LIMIT} calls per ${WINDOW_MS / 1000}s
    `);
  }
  callLog.push(now);
  return fetch(url);
}
```

Both are independently re-verifiable: the allowlist predicate is a pure function (test it in a REPL in 30 seconds); the rate limiter's persistence is visible by inspecting module scope.

6. The Honest Caveats

This section exists because the guide would be self-undermining without it. A guide warning about over-confident AI output that then over-claims about a solution is the exact pathology it's warning against.

The gate catches only what you specify

> **The gate is not a general security oracle.** It catches the vulnerabilities you thought to test for. Unknown unknowns slip through. A gate with four checks catches those four failure modes. It says nothing about the fifth failure mode you didn't think of.
> This is not a flaw unique to this approach. It is a fundamental property of testing: you can prove the absence of the bugs you checked for. You cannot prove the absence of all bugs.

The answer is not to abandon the gate — it is to be honest about what it covers and write better gates over time.

The gate IS your acceptance criteria, made executable

The gate checks are your Ideal State Criteria. Per-task criteria is not a burden — it is the methodology:

You cannot verify what you don't define.

If you can't state in plain English what "correct" means for this artifact, you can't write a gate for it, and you can't meaningfully verify it by any means — including manual review. Writing the gate forces you to define correctness. That discipline is valuable on its own.

The gate itself must be fair and isolated

A gate with bugs is worse than no gate: it's unwinnable. During the live session, the first draft of the rate-limiter check replicated the model's internal state logic in the test. That made the check impossible to pass — the gate was testing something the artifact had no way to satisfy. Similarly, if rate-limit state bleeds between gate checks (because checks run in the same process and share module scope), you get false failures on the second check.

Both were caught and fixed. But the lesson is: **review your gate before you trust its verdict.** A failing gate that you haven't audited might be failing because your artifact is broken, or it might be failing because your gate is broken.

Writing gates at scale

The obvious objection: writing a bespoke adversarial gate per task is expensive. This is true. Mitigations:

- **Reusable gate library.** Common checks ("does the artifact import cleanly?", "do its exported functions run without throwing?", "is the output directory non-empty?") belong in a shared library. Most artifacts share a baseline gate; you add bespoke adversarial checks only for security-critical or domain-specific behavior.
- **Generic baseline gate.** For most non-security tasks, a gate that checks: (a) does it run? (b) does `bun test` pass? (c) does the output meet basic structural requirements? — covers the bulk of failure modes without custom per-task test writing.
- **Bespoke adversarial gates for the right tier.** Reserve the full adversarial treatment for security-critical artifacts: authentication, allowlisting, rate limiting, encryption, anything that mediates access to something. For a utility function that reformats text, the baseline gate is probably enough.

The human's irreplaceable role

Writing the adversarial gate *is* the threat modeling the model can't do. This is not a flaw in the approach to be engineered away. It is a correct division of labor:

- The model is fast at generating plausible implementations.
- You are the one who can reason about the attacker's perspective, define "correct," and encode that definition as executable checks.
- The gate is the handshake between those two roles.

Removing humans from this loop doesn't produce better security. It produces faster generation of things that *look* secure. The distinction between those two outcomes is exactly what this guide is about.

7. Putting It Together

Here is the practical architecture for building local AI output you can actually trust:

1. **Define the artifact's Ideal State Criteria** before prompting the model. What does "correct" mean? What are the adversarial inputs? Write these down.
2. **Encode the criteria as a gate** — a script that takes the artifact's path, runs the checks, and exits 0 on PASS or non-zero on FAIL with a description of what failed.
3. **Wire the gate into the build loop.** Model builds → gate runs → on FAIL, feed the failure output back → model fixes → gate re-runs. Repeat up to `maxAttempts`.
4. **Let the gate's exit code be the verdict.** Do not accept the artifact on a model claim. Accept it on a gate PASS.
5. **Tell the model its environment.** Exact runtime (`bun`, `python3`, etc.), available commands, test invocation. This eliminates the "gave up because `node` isn't installed" failure mode.
6. **Review the gate itself** before trusting it. An unwinnable gate is worse than no gate.

Honest ceiling for this approach.
Running on consumer hardware (single RTX-class GPU) with a ~30B local coder model, this architecture:

- Reliably catches the structural failure modes described in this guide when they are specified in the gate
- Drives the model to correct both the bypassable allowlist and the inert rate limiter through the feedback loop
- Produces output you can independently re-verify (the fixed predicates are individually testable)

It does not:

- Catch unspecified vulnerabilities
- Replace a security audit for production systems
- Make the model deterministic

What you get is an agent whose output you can trust *within the scope of your gate* — which is meaningfully better than trusting a model's self-report. That's a real improvement on the local agent stack from Guide #1.

8. Closing: What You Actually Own Now

Guide #1 gave you a local agent that runs on your hardware, costs $0 per query, and answers to no export-control directive. This guide gives you the tool to evaluate what it produces.

The combination is the actual sovereign stack. Not just a model running locally, but a local model whose output is gated by deterministic checks you wrote, under your control, on your machine. The model generates; the gate certifies; you ship.

For a reader on consumer hardware — single GPU, ~30B model class — the practical payoff is this: you can now hand the agent a security-adjacent task and have a meaningful answer to the question "is this safe to run?" The answer comes from code that runs the same way every time, not from a people-pleasing model that was trained to say yes.

That's not a complete security solution. It's a correct foundation: define correctness, encode it, enforce it deterministically. Everything beyond that builds on the same principle.

Want more?

More guides on local AI, self-hosting, privacy, and digital sovereignty:

cypherpunkschool.com