
VeriTool: Safer Agents through Safer Tools

Saurav Panigrahi

Research Engineer,
Zoho Labs

With
Apart Research

Abstract

Agent safety is often discussed at the level of the whole agent, but many practical failures happen at a lower level: the tools the agent is allowed to use. A file reader, SQL wrapper, API caller, or evaluator can each create a very different failure mode. VeriTool explores a simple alternative: instead of treating these tools as trusted infrastructure, synthesize them under explicit contracts and reject unsafe candidates with counterexamples.

The system implements one shared counterexample-guided synthesis loop across six micro-tool families, each with a bounded verifier tailored to its contract. We ran the full suite live on four models under the same six-tool benchmark and the same four-attempt budget per tool. Across those runs, 15 of 24 tool instances converged, with the best model reaching 5 of 6 tools. The main value of the artifact is not just the final pass count. It is the recorded repair process: first failure, concrete counterexample, next attempt, and final acceptance or budget exhaustion. That gives a more inspectable picture of how models behave when capabilities have to be earned through verifier feedback rather than granted up front. Code: <https://github.com/expmode/veritool>

1. Introduction

A lot of agent safety discussion happens at too high a level. We ask whether an agent is aligned, whether it can be trusted, or whether its reasoning is safe in general. But in real systems, the sharpest failures often come from the interfaces the model is given. A model that can read arbitrary files, issue network requests, run SQL, or evaluate code has a much larger hazard surface than a model that can only emit text. In practice, the question is often less “is the agent safe?” and more “what exactly are we letting it do?”

This project starts from that narrower question. Instead of assuming the tool layer is fixed and trusted, VeriTool treats tools themselves as the thing to synthesize and check. The goal is not to prove that a whole agent is safe. The goal is to make capability admission more disciplined: if a model wants a file reader, API caller, or evaluator, that capability should enter through a small interface with a stated contract and a verifier that can reject bad implementations.

That framing is useful for two reasons. First, tool contracts are much narrower than general behavioral claims, so they are easier to specify honestly. Second, failures at the tool boundary are usually easier to turn into concrete counterexamples. A path traversal bug, a forbidden HTTP method, or a bad SQL clause can often be demonstrated with a small witness. That makes the repair loop legible in a way that broader agent failures usually are not.

VeriTool implements this as a counterexample-guided synthesis loop. For each micro-tool, the model is prompted with a contract and asked to produce an implementation. A verifier then checks that implementation against bounded policy or semantic conditions. If it fails, the failure is turned into a structured counterexample and fed into the next attempt. A tool is accepted only if the verifier clears it within budget.

Our main contributions are:

1. We frame safe tool admission, rather than whole-agent behavior, as the primary object of verification.
2. We implement a shared CEGIS-style runtime across six different agent-tool classes, each with its own bounded verifier.
3. We run a live multi-model benchmark under one common loop and one common four-attempt budget per tool.
4. We release full repair traces, including counterexamples and accepted implementations, so the artifact captures not just outcomes but how those outcomes were reached.

One source of inspiration for this project was recent work on Recursive Language Models (RLMs), which argues that some problems become easier to handle when the model is allowed to interact with an external environment instead of being forced to solve everything inside one opaque model call. VeriTool borrows that general intuition, but applies it to tool synthesis rather than

long-context reasoning. Here the environment is not a notebook over a long prompt, but a verifier-guided repair loop over candidate tools. The shared idea is that model behavior becomes more useful when it is structured by an external system that can expose failure and guide the next step.

2. Related Work

VeriTool draws on several lines of work: counterexample-guided inductive synthesis, spec-driven program synthesis, executable-oracle evaluation, and safer tool design for agent systems. In standard CEGIS, a candidate program is tested against a specification, and any failure is turned into a concrete witness that guides the next attempt. That pattern fits tool synthesis surprisingly well, because many unsafe behaviors at the tool boundary can be shown with compact examples rather than broad natural-language judgments.

Compared with vericoding-style synthesis benchmarks, the main difference here is the target. We are not trying to synthesize arbitrary programs from specs. We are trying to synthesize small agent-facing tools under explicit safety contracts. Compared with ordinary coding benchmarks, the evaluation is less about whether code “works” in a generic sense and more about whether it obeys a policy oracle under bounded adversarial cases. Compared with broader agent-safety work, the emphasis here is on shrinking the capability surface instead of trying to certify an unconstrained agent end to end.

It is also important to be clear about what VeriTool is not. It is not a general proof system for arbitrary Python. It is not a full agent architecture. It is not a substitute for sandboxing, monitoring, or defense in depth. It is a small capability-admission scaffold: a way to ask whether a concrete tool can be admitted into an agent environment under a contract that is narrow enough to check honestly.

The gap we are trying to address is practical. There is a lot of interest in giving agents safer interfaces, but less evidence about what happens when the tools themselves have to be synthesized under explicit verifier pressure. VeriTool is meant as a bounded artifact for that setting.

3. Methods

At a high level, this setup is inspired by the same external-environment intuition seen in Recursive Language Models, except that the environment here is a synthesis-and-verification loop rather than a REPL over long context. VeriTool uses one shared runtime across six tool classes. Each run follows the same basic loop. We define a micro-tool and its safety invariant, construct a prompt with the contract and any prior counterexamples, ask the model for an implementation, run AST

guards plus a tool-specific bounded verifier, and either accept the candidate or feed the failure back into the next iteration. The first candidate that clears the verifier is accepted.

The key design decision was to treat tools, not downstream tasks, as the object of synthesis. In a normal agent setup, tools are already there and the model is expected to use them safely. We flipped that around. Before granting a capability, we ask whether that capability can be distilled into a narrow tool with a contract simple enough to verify. In that sense, the verifier is not just an evaluator. It is the gatekeeper for capability admission.

The six tool tracks are Local Context Reader, Bounded SQL Querier, Safe API Caller, Append-Only Logger, PII Masker, and Compute-Bounded Evaluator. These were chosen because they cover several common agent capabilities while requiring different kinds of checks. The file reader uses path and extension policy. The SQL tool uses a small grammar and reference behavior. The API caller enforces transport, host, and method restrictions. The logger checks an append-only state transition. The masker checks for residual sensitive patterns. The evaluator checks a small arithmetic AST and bounded semantic equivalence.

Each tool is intentionally narrow. That is not just for presentation. Narrow tools make the contracts easier to state and the failures easier to localize. A broad agent policy usually fails in messy ways. A tool contract usually fails in a way that can be pointed to directly: a bad path, a disallowed AST node, an unsafe URL, or a mutating SQL statement. That is what makes counterexample-guided repair workable here.

We do not claim a universal verifier for arbitrary Python. Each tool has a bounded signature, a bounded input or state space, an explicit policy oracle, and a verifier that converts unsafe behavior into a structured counterexample. That keeps the evidence honest, even if it is necessarily limited.

The repository supports both replay mode for deterministic regression testing and live OpenAI-compatible mode for real runs. The completed live comparison set uses the same six-tool suite and the same four-attempt budget per tool for ``anthropic/claude-sonnet-4``, ``openai/gpt-4o``, ``deepseek/deepseek-v3.2-exp``, and ``google/gemini-2.5-pro``.

One detail that matters in practice is the role of the counterexample. In this system it is not just a failed test. It is the compact piece of evidence that gets shown back to the model on the next attempt. That is what makes the repair traces readable: we can see not just that a candidate failed, but what evidence was enough to force the next revision.

4. Results

We evaluated the same six-tool suite under the same four-attempt budget per tool for four completed live runs. Across those runs, the system recorded 24 total tool instances and 15

converged ones, for an aggregate convergence rate of 62.5%. The mean number of iterations per tool was 3.33.

Per-model outcomes were:

- `anthropic/claude-sonnet-4`: 5 of 6 tools converged in 17 total iterations
- `openai/gpt-4o`: 4 of 6 tools converged in 21 total iterations
- `deepseek/deepseek-v3.2-exp`: 4 of 6 tools converged in 20 total iterations
- `google/gemini-2.5-pro`: 2 of 6 tools converged in 22 total iterations

The headline result is that the same contracts and the same repair loop produce noticeably different behavior across models. Claude Sonnet 4 gave the strongest completed run. GPT-4o and DeepSeek were in the middle. Gemini 2.5 Pro completed the benchmark but used up more of its budget on AST-guard failures and incomplete repairs. Since the benchmark, prompts, and budget were held fixed, these differences mostly reflect how well each model responds to verifier feedback.

The more interesting result, though, is in the traces rather than the summary table. VeriTool records the path from first failure to accepted repair. In the Local Context Reader, an early candidate failed on a valid nested path because it did not canonicalize correctly. In the SQL tool, an early candidate ignored the WHERE clause and effectively returned all rows. In the PII masker, an early candidate handled emails but left phone and card patterns behind. In the evaluator, an early candidate accepted syntax outside the allowed grammar. These are all small failures, but they are exactly the kind that matter if such tools are going to sit under an agent.

A few repair patterns came up repeatedly. Sometimes a model saw one concrete witness and fixed the issue immediately. Sometimes it fixed the surface behavior but still violated a structural restriction such as an AST guard. In a few cases, models repeated the same kind of mistake even after being shown a counterexample. That last case is important because it shows the limit of prompt-level repair: some errors are easy to patch once named, while others reflect a deeper mismatch between the task and the model’s default coding habits.

Overall, the results suggest that explicit tool contracts are a useful way to probe model behavior. A flat pass rate would miss most of what is interesting here. The trajectory view shows which failures are easy to repair, which are sticky, and where the difficulty really lies across tool classes.

5. Discussion and Limitations

What this benchmark gives us is not just a pass rate, but a clearer view of how models behave when access to a capability is conditional on verifier feedback. The traces show whether a model

can use a concrete counterexample, whether it keeps making the same structural mistake, and whether certain kinds of tools are consistently harder than others. In our runs, bounded logging, local file reads, and small SQL fragments were usually easier to repair than outbound API policy enforcement and arithmetic evaluation under a restricted grammar.

That points to a broader systems lesson. If we expect future agents to accumulate more capabilities over time, it may be better to make those capabilities pass through small, inspectable interfaces than to hand the model broad power and hope later layers catch misuse. VeriTool is not a full safety solution, but it does make the admission step more legible. Instead of asking whether a powerful tool can be used safely in the abstract, we ask whether a narrower version of that tool can be synthesized under a contract we are actually willing to check.

I think of this less as a replacement for alignment or sandboxing, and more as an additional control layer. The contribution is not that the verifier is perfect. It is that the interface is narrow enough for verification to be meaningful at all. In practice, that is already an improvement over giving an agent a general-purpose primitive and relying on prompt instructions alone.

Limitations:

The current artifact is deliberately bounded and micro-scoped. It does not provide a symbolic semantics for arbitrary Python, and the evidence here should not be read as a universal proof of safety. The verifiers are a mix of AST guards, policy fixtures, regex checks, and bounded executable oracles. That is enough to make the benchmark concrete, but not enough to claim completeness. The live evaluation also depends on provider behavior and endpoint stability. Some requested runs stalled and were excluded from the completed comparison set rather than treated as real outcomes.

There are also important limits to the framing itself. VeriTool does not solve arbitrary tool synthesis, replace sandboxing, or remove the need for monitoring and defense in depth. A weak contract can still admit a bad tool, and a bad verifier can still approve the wrong thing. For that reason, I would describe the system as a defensive prototype for capability admission, not as a general verified-programming solution for agent systems.

Future Work:

The next steps are fairly clear. Some of the bounded executable checks could be replaced with stronger symbolic or solver-backed reasoning. The SQL and evaluator tracks could be broadened while still keeping the contracts tractable. The system could also support richer stateful tools and better handling of latency, retries, and stalled model calls. A more ambitious direction would be to train models explicitly for this style of verifier-mediated repair, instead of relying on generic coding behavior plus prompt feedback.

6. Conclusion

VeriTool treats tool synthesis as a policy-and-counterexample loop. Instead of trusting an unconstrained agent with a broad capability, it asks whether a narrower version of that capability can be synthesized under a contract and admitted only after verification. Across four completed live runs, that setup exposed meaningful differences in how models respond to repair pressure under the same constraints.

I do not think this solves agent safety in any broad sense. What it does offer is a more workable unit of control. If future agent systems are built out of many small operational capabilities, then one practical safety question is how each of those capabilities is allowed into the system in the first place. VeriTool is a small attempt at making that admission step more explicit, more inspectable, and easier to reason about.

Code and Data

- **Code repository:** <https://github.com/expmode/veritool>
- **Other artifacts (optional): Demo website:** <https://demo-site-blond-two.vercel.app/>

Author Contributions

Saurav Panigrahi conceived the project, implemented the artifact, ran the experiments, wrote the report, and reviewed the final submission.

References

1. Zhang, A. L., Kraska, T., and Khattab, O. 2026. Recursive Language Models. arXiv:2512.24601. <https://doi.org/10.48550/arXiv.2512.24601>
2. Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., and Veith, H. 2000. Counterexample-Guided Abstraction Refinement. In Computer Aided Verification (CAV).
3. Solar-Lezama, A. 2013. Program Sketching. International Journal on Software Tools for Technology Transfer, 15(5-6).
4. Regehr, J. 2026. Zero-Degree-of-Freedom LLM Coding using Executable Oracles. Blog essay.
5. Melcer, D. et al. 2024. Approximately Aligned Decoding. arXiv:2410.01103.
6. Vericoding: A Benchmark for Formally Verified Program Synthesis. 2025. arXiv:2509.22908.

Appendix

Supplementary materials for the project include a leaderboard for completed live runs and a trajectory explorer showing prompts, counterexamples, raw responses, and accepted code for each tool. The benchmark uses a shared four-call budget per tool, so non-convergence means that the verifier never accepted a candidate before the budget was exhausted.

The repository also includes machine-readable run artifacts, markdown summaries, and per-model trajectory bundles. These materials are intended to make the synthesis-and-verification process auditable rather than only reporting final outcomes.

LLM Usage Statement

I used an LLM-based coding assistant to help with implementation and editing. All experimental results, quantitative claims, and final report text were manually reviewed before inclusion.