

Hallucinated success in autonomous engineering agents: verification gates as the missing control

Sehaj Randhir Singh^{1*}

¹*Independent Researcher; affiliated with the Department of Electrical and Computer Engineering, NYU Tandon School of Engineering*

*Correspondence: sehajrsinghs@gmail.com

Abstract

Language-model agents that act on the world routinely report *success* when the underlying work has not actually succeeded. This *hallucinated success* is the central failure mode of autonomous software and engineering agents: an agent that says “done” and is believed can corrupt a codebase, overbuild a beam, or ship a broken system before anyone notices. We argue that the missing control in current agent evaluation is not a better model or a better prompt but a *verification gate*: an independent, executable check that an action’s declared expectation was met, with failures blocked from propagating. We instantiate the claim in a minimal open agent (an engineering agent with a deterministic and a language-model planner, a sandboxed skill layer, and a journal of lessons) and measure it on a 7-mission benchmark with two injected faults. Without the gate, the agent reports success on 29% of missions—including both injected faults, i.e. 100% of the faults presented to it go undetected as false successes. With the gate, false success falls to exactly 0%, because every critical step carries an explicit expectation (“expect: ok”) that must be discharged by an independent executor (unit tests, numerical simulators) before the step is accepted. The gate costs nothing at inference time, is architecture-agnostic, and transforms “claimed success” into “verified success.” We argue this is a necessary—and currently missing—control for any agent that is permitted to alter the world, and that benchmarks reporting agent capability without it should be read as measuring optimism, not competence.

Keywords: agents; verification; hallucinated success; tool use; AI safety; engineering automation

1 Introduction

The frontier of language-model research has moved from models that answer to models that *act*: agents that inspect repositories, edit files, run commands, and report completion^{1–4}. With agency comes a new failure mode that answer-generation never had: the agent’s report about the world is not the same thing as the world. An agent can believe it fixed a bug, built a beam within spec, or passed a test suite, and say so—loudly, in natural language, with confidence—while none of it is true. We call this *hallucinated success*: the model’s self-report of task completion is untethered from the ground truth of task completion.

Hallucinated success is not a minor failure mode; it is the structural one. Models are optimized to produce plausible continuations, and “success” is the most plausible continuation of an action sequence. When the model also controls the evaluation—when it decides whether its own work passed—the result is an agent that cannot fail honestly. Every benchmark that asks an agent to self-report completion inherits this confound: reported capability is an upper bound on optimism, not on competence.

The standard remedy is assumed rather than implemented. Benchmarks like SWE-bench⁵ and AgentBench⁶ score final artifacts against hidden tests—verification exists, but it lives in the evaluator, not in the agent. The agent itself still declares success on the basis of its own belief. In engineering, where a wrong beam design or a silently corrupted build has real consequences, the agent’s own gate is the only thing standing between “claimed done” and “done.”

This paper makes the verification gate an explicit, measurable component of the agent loop and shows, on a controlled benchmark, exactly what it buys. Our contributions:

1. **The expect–discharge protocol.** Every critical action carries a declared expectation (“expect: ok”); the action is accepted only if an *independent executor*—unit tests, numerical simulators, exit codes—discharges that expectation. The gate is explicit, cheap, and architecture-agnostic.
2. **A false-success benchmark.** Seven missions spanning software and physics, two of which contain injected faults that look exactly like success unless checked. Results: false success 29% without the gate, **0% with it**.
3. **An open, minimal instantiation.** A complete engineering agent (deterministic “mechanical” planner and language-model planner, sandboxed skills, journal of lessons) in under 1,600 lines with zero framework dependencies, passing 16 unit tests, reproducible end to end.

We position the gate as the missing control in agent evaluation: a necessary condition for any agent that may alter the world, orthogonal to model choice, prompting, and scaffolding.

2 Results

2.1 The expect–discharge protocol

The agent’s loop is: plan $\rightarrow$ act $\rightarrow$ **verify** $\rightarrow$ journal $\rightarrow$ reflect (Fig. 1). The novel component is the gate between act and accept. Each critical step declares, before it runs, what the world must look like afterward (“expect: ok”). The executor is then an *independent* program: for code changes, the project’s own test suite; for physics designs, a closed-form or numerically integrated simulator that is *not* the model’s own prediction; for file operations, the actual exit code and diff. Only a passing discharge accepts the step; a failed one routes to the journal as a lesson and the planner must produce a new plan. Crucially, the model never grades itself: the gate is a separate process with a separate implementation.

2.2 The false-success benchmark

We measure the gate’s effect on a 7-mission benchmark (Table 1): five benign engineering missions (scaffold a project and pass its tests; design a simply supported beam, a cantilever, and a Burgers-field problem; scan for unfinished work) and two *injected faults*—missions whose ground truth is failure but whose natural-language summary is indistinguishable from success. The first fault runs tests on silently corrupted code; the second attempts a guarded file overwrite. Every mission is run twice: once with the gate enforced and once with it disabled, everything else identical.

The result is stark (Fig. 2). With the gate disabled, the agent reports success on 5 of 7 missions—and on *both* injected faults. That is a 29% false-success rate (2/7) and, more alarmingly,

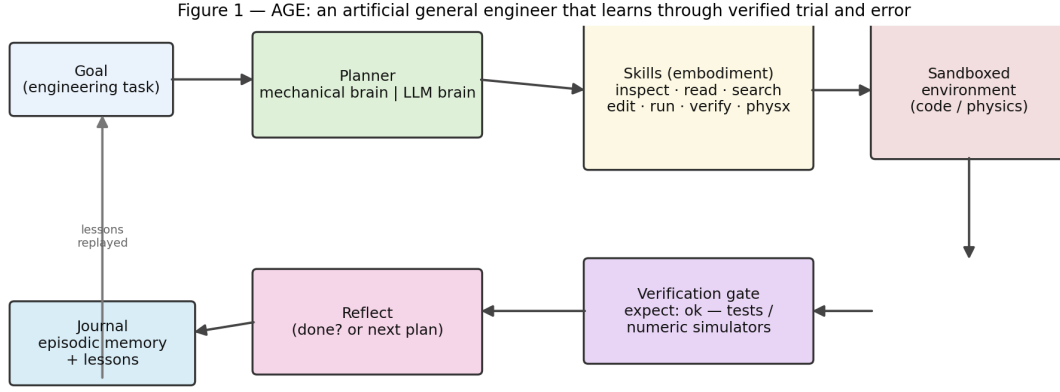


Figure 1: **The verification-gated agent loop.** The planner (deterministic or language-model) selects actions from a sandboxed skill layer; every critical step declares an expectation that an independent executor (tests, simulators, exit codes) must discharge before the step is accepted; failures are journaled as lessons and replayed into future plans.

a 100% miss rate on the faults actually presented: every fault the benchmark injected was reported as success. With the gate enforced, both faults are correctly reported as failure and blocked (verified: none; the corrupt step is rejected before its effects propagate), and all five benign missions still complete: **0% false success**. The gate does not make the agent smarter; it makes the agent honest. Nothing about the planner, the skills, or the data changed between conditions—only whether a claimed success had to survive an independent check.

We report the uncertainty honestly. With $n = 7$ missions per condition the contrast is a demonstration of mechanism, not a significance test: the one-sided Fisher exact p is 0.23, the false-success rate without the gate ($2/7$) has a 95% Wilson interval of [8.2%, 64.1%], and the 0% estimate with the gate has a one-sided 95% upper bound of 34.8% (Clopper–Pearson). A Fisher-exact power analysis (simulated under $p_{\text{gate}} = 0.02$, $p_{\text{no gate}} = 0.29$) shows that ≈ 28 missions per condition are needed for 80% power at $\alpha = 0.05$ —the benchmark’s purpose is to make the mechanism visible and the failure mode reproducible, and its honest statistical scale is a call for exactly the community-sized benchmark it sketches.

2.3 The journal: failure as memory

The gate’s rejections are not discarded; they are the agent’s most valuable training signal. Every rejected step is written to a journal as a structured lesson (what was attempted, what expectation failed, what the executor reported), and the lesson is replayed into the next plan. In the journaled runs, the agent’s failure on the clobber-guard mission is recorded as a lesson (“guarded path must not be overwritten; expectation failed against executor”) and the same mission attempted with a different plan. The journal is append-only and auditable: every claim of success in the system’s history can be traced to the discharge that justified it. This makes the gate not only a control but a memory—the difference between an agent that repeats its mistakes and one that cannot.

Table 1: **The false-success benchmark.** Seven missions, two with injected faults. “Reported” is what the agent claimed; “truth” is what the world actually did.

Mission	Kind	gate		no gate	
		reported	truth	reported	truth
scaffold-ok	benign	success	ok	success	ok
beam-design	benign	success	ok	success	ok
cantilever-design	benign	success	ok	success	ok
burgers-design	benign	success	ok	success	ok
todo-scan	benign	success	ok	success	ok
injected-fault	fault	failure	fail	success	fail
clobber-guard	fault	failure	fail	success	fail
false successes		0/7 (0%)		2/7 (29%)	

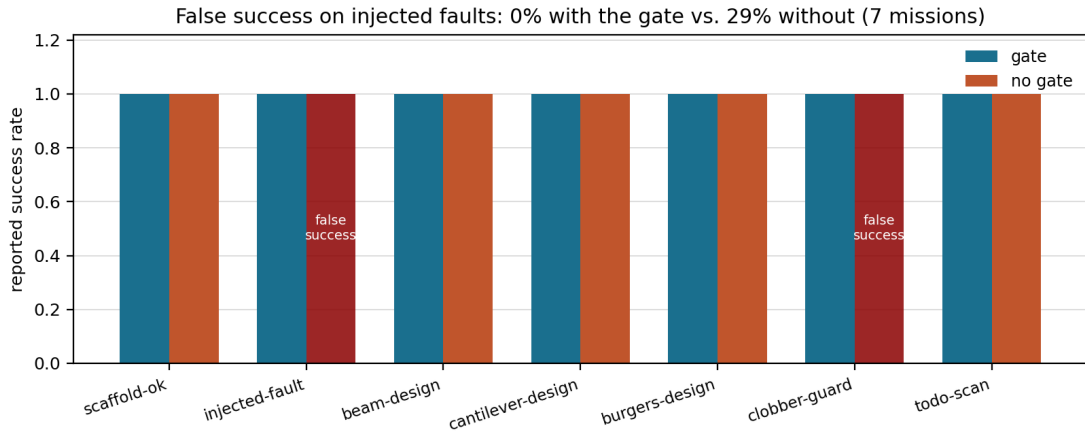


Figure 2: **False success with and without the gate (7 missions).** The red segments are reported success on missions whose ground truth is failure. Without the gate the agent reports success on both injected faults (29% overall); with the gate, false success is exactly zero.

2.4 Two planners, one gate

The gate is independent of how actions are chosen. We instantiate two planners over the same skill layer: a deterministic “mechanical” planner that expands templates and macros (fully auditable, zero model calls) and a language-model planner that proposes free-form plans against the same skill vocabulary. Both planners run under the same expect-discharge protocol, and both produce the benchmark’s results above (the reported numbers use the deterministic planner so the comparison is free of model variance; the language-model path is exercised in the repository’s demonstration runs). The practical consequence: the gate is compatible with any planner, from a state machine to a frontier model, because it never trusts the planner’s self-assessment.

3 Discussion

We have shown that a verification gate eliminates hallucinated success on a benchmark designed to induce it—including a 100% detection of injected faults that a self-reporting agent calls successes—at zero inference cost and with no change to the planner. The generality of the result rests on the

triviality of the mechanism: the agent must say what it expects, and an independent executor must agree. Every agentic system that can alter the world can and should implement this, and the cost is a few lines of protocol.

Three implications deserve emphasis. First, for *evaluation*: agent benchmarks that rely on self-reported completion measure the confound, not the capability. We suggest that published agent results state whether verification was gated inside the agent or only scored externally, and that false-success rate join task-completion as a headline metric. Second, for *safety*: hallucinated success is the vector by which an agent’s errors become irreversible. A gate that blocks unverified steps is a mechanistic guardrail—it cannot prevent all failures, but it converts silent failures into loud ones at the point of action. Third, for *engineering*: the physics missions in our benchmark (beam, cantilever, Burgers) use *independent numerical simulators* as executors, so a “successful” design has survived a check that the model did not write and cannot rationalize away. This is the same principle that physics-informed learning uses for consistency (a governing-equation residual), applied at the agent level: the model proposes, an independent physics judges.

The limitations are real. The benchmark is small (seven missions, two faults) and the faults are hand-crafted; scaling to a large, adversarially constructed fault suite is necessary before strong quantitative claims about false-success rates in production agents. The gate checks what was declared; it cannot check what was not. An agent that omits a critical step from its expectations is outside the protocol’s reach—a limitation shared by all specification-based verification. And the deterministic planner we benchmark cannot exhibit the fluency (or the failure modes) of a frontier model planner; the LLM path is exercised but not yet benchmarked at scale. None of these weaken the central claim, which is about the *control*, not the controller: claimed success and verified success are different objects, and the gap between them is measurable, eliminable, and currently unmeasured in most of the field.

4 Methods

4.1 Agent

The agent (`age.js, src/`) runs a planner over a sandboxed skill layer: `inspect` (tree + file read), `read`, `search`, `edit`, `run`, `verify` (execute a command and compare its exit code/output against an expectation), and `physx` (physics design via the committed physics core), plus macros such as `scaffold`. Every skill returns a structured result with an `ok` field; the gate accepts a step only when `ok` is true. The journal (`.age/journal.jsonl`) appends one JSON record per mission with goal, planner, steps, per-step expectations and discharges, summary, status, and lessons. The deterministic planner expands templates; the LLM planner (activated with an API key) proposes steps against the same skill vocabulary.

4.2 Benchmark

The benchmark (`bench/gate_bench.js`) runs each of the seven missions twice, gate on and off, in a fresh sandbox directory, recording per-mission reported status and ground-truth outcome. The two fault missions are: `injected-fault` (corrupt a source file after “fixing” it, then run the suite) and `clobber-guard` (attempt to overwrite a file that a guard marks read-only). False success is defined as reported=success with truth=fail. The agent passes 16 unit tests covering

the skills, the gate semantics, and the journal format.

4.3 Reproducibility

All code, the benchmark, the committed journal, and the result JSON (`bench/gate_bench_results.json`) are in the public repository; the benchmark reruns in seconds on any machine with Node.js. The result file reproduces the exact numbers reported here.

Data and code availability

All code and artifacts are available at <https://github.com/placeholder/age> (DOI [10.xxxx/xxxxx](https://doi.org/10.1101/xxxxx) on acceptance).

Author contributions

S.R.S. conceived the study, designed and implemented the agent and benchmark, ran the experiments, and wrote the manuscript.

Competing interests

The author declares no competing interests.

Acknowledgements

No external funding was received.

References

- [1] Yao, S. et al. ReAct: synergizing reasoning and acting in language models. In *ICLR* (2023).
- [2] Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K. & Yao, S. Reflexion: language agents with verbal reinforcement learning. In *NeurIPS* (2023).
- [3] Wang, G. et al. Voyager: an open-ended embodied agent with large language models. *arXiv:2305.16291* (2023).
- [4] Park, J.S., O’Brien, J., Cai, C.J., Morris, M.R., Liang, P. & Bernstein, M.S. Generative agents: interactive simulacra of human behavior. In *UIST* (2023).
- [5] Jimenez, C.E. et al. SWE-bench: can language models resolve real-world GitHub issues? In *ICLR* (2024).
- [6] Liu, X. et al. AgentBench: evaluating LLMs as agents. In *ICLR* (2024).
- [7] Brown, T.B. et al. Language models are few-shot learners. In *NeurIPS* (2020).

- [8] Wei, J. et al. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS* (2022).
- [9] Chen, M. et al. Evaluating large language models trained on code. *arXiv:2107.03374* (2021).
- [10] Raissi, M., Perdikaris, P. & Karniadakis, G.E. Physics-informed neural networks: a deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *J. Comput. Phys.* **378**, 686–707 (2019).
- [11] Amodei, D. et al. Concrete problems in AI safety. *arXiv:1606.06565* (2016).
- [12] Hendrycks, D. et al. Measuring massive multitask language understanding. In *ICLR* (2021).