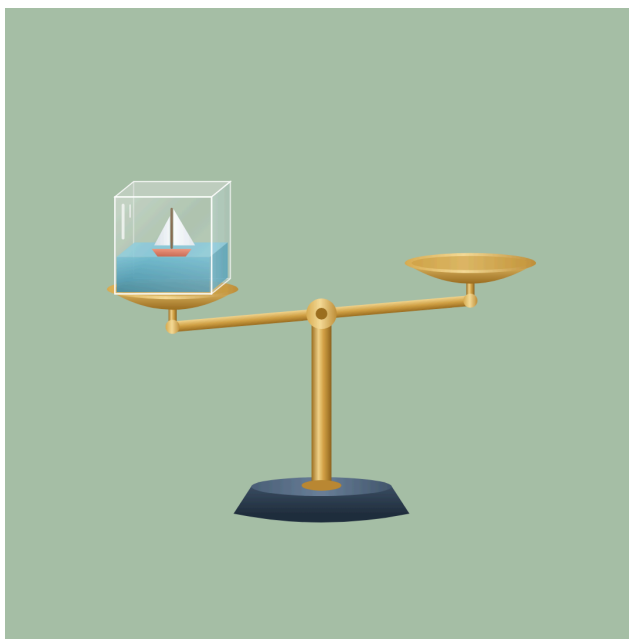




The Sandbox Is Not the Grader: Building Trustworthy Agent Evaluations with Harbor

Containers made agent benchmarks reproducible to run. Making their rewards trustworthy requires treating the verifier, the information boundary, and failure accounting as first-class research objects.

[Author Name], [Role]

Abstract

Agent benchmarks have converged on a common architecture: give an AI agent a natural-language instruction, drop it into an isolated container, let it act, and then run a script that decides whether the work was done. Harbor, the open-source framework released by the creators of Terminal-Bench and now the official harness for Terminal-Bench 2.0, packages this architecture into a portable task format and a runtime that can execute thousands of trials across local Docker and many cloud sandbox providers [1][2]. This article argues that the sandbox, while necessary, is the easiest part of a trustworthy evaluation to get right. The harder questions sit around it: whether the agent can see or influence the ground truth, whether the verifier's reward changes when, and only when, a required behavior is violated, whether infrastructure failures are kept out of capability scores, and whether a handful of trials can support the conclusions drawn from them. Using Harbor's documented

design, the Terminal-Bench 2.0 task corpus, and published audits of existing agentic benchmarks, we describe four boundaries every sandboxed evaluation must hold, examine how Harbor’s features map onto them, and propose practices for building verifiers that reject plausible wrong solutions while accepting legitimate alternatives.

1. Introduction: a reward is a claim

When an agent evaluation reports that a model solved a task, it is making a specific empirical claim: that the agent’s actions produced a state of the world satisfying the task’s requirements. Every downstream use of that number depends on the claim being true. Leaderboards rank models by it, procurement teams compare vendors with it, safety teams use it to estimate what a system can do autonomously, and reinforcement-learning pipelines feed it back into training as a reward signal. A false positive in an evaluation is therefore not a cosmetic error. In a leaderboard it inflates capability; in an RL loop it teaches the model to do the wrong thing.

Static question-answering benchmarks made this claim comparatively easy to check. A model produced a string, and the grader compared it to a reference. Agentic benchmarks are different in kind. The agent operates over many steps, modifies files, starts processes, installs packages, and leaves behind a changed environment. Correctness is a property of that environment, not of a single output, and the space of states an agent can produce is effectively unbounded. The grader has to infer success from evidence, and the agent, by construction, has broad control over the evidence.

The field’s response has been to move evaluation into sandboxes: isolated, reproducible computing environments that are created fresh for each attempt and discarded afterward. SWE-bench moved to a fully containerized Docker evaluation harness in June 2024 for more reproducible evaluations [9]. The UK AI Security Institute’s Inspect framework provisions containers so that tools, agents, and scorers can execute commands in isolation [13]. Terminal-Bench built its benchmark around containerized terminal tasks, and its successor framework, Harbor, generalizes that design into a task format and execution layer that any benchmark can adopt [1][7][8].

This convergence is real progress. It is also easy to misread. A sandbox answers the question “where did the agent run?” It does not answer “was the agent graded correctly?” The rest of this article is about the gap between those two questions.

2. How agent evaluation moved into the sandbox

It helps to be precise about what the current generation of tools established and when.

SWE-bench, introduced by Jimenez and colleagues, framed software engineering evaluation around resolving real GitHub issues and checking the result with repository tests [9]. Its maintainers later introduced SWE-bench Verified in August 2024, a 500-problem subset that software engineers confirmed to be solvable [9]. The existence of a human-verified subset matters because it reflects a pattern that recurs across agent benchmarks: the first release establishes the format, and later curation effort goes into confirming that tasks are solvable, adequately specified, and graded by tests that measure what the task asks.

Terminal-Bench, first released in April 2025, extended containerized evaluation from repository patches to open-ended terminal work; its maintainers describe tasks that range from assembling proteins for synthesis to debugging asynchronous code and resolving security vulnerabilities [7][8]. The Terminal-Bench paper was published at ICLR 2026 [8]. Terminal-Bench 2.0 contains 89 tasks in its public repository, each of which, according to the maintainers, received several hours of human and LM-assisted validation to confirm that it was solvable, realistic, and well-specified [7]. The same FAQ explains that the team rewrote the original evaluation harness “from the ground up for improved reliability, observability, scalability, and performance,” producing Harbor [7].

Harbor’s first release on the Python Package Index appeared on 2 November 2025, and version 0.24.0 was released on 5 October 2026 [1][10]. As of mid-October 2026, the project describes itself as a framework for evaluating and optimizing agents and language models that can run arbitrary agents such as Claude Code, OpenHands, and Codex CLI, distribute benchmarks, execute experiments in parallel across cloud sandbox providers, and generate rollouts for RL optimization [1]. Its repository contains 85 adapter directories that port third-party benchmarks, including SWE-bench, into the Harbor task format [1].

Harbor is not the only serious framework in this space. Inspect offers a mature sandboxing abstraction with built-in Docker support and an extension API for other sandbox types, along with a large collection of pre-built evaluations [13]. The arguments below apply to any containerized harness. Harbor is a useful lens because its task format makes the components of an evaluation explicit, and because its documentation and changelog openly record the design decisions and corrections that the rest of this article draws on.

3. The anatomy of a Harbor trial

A Harbor task is a directory with four parts: an instruction written in Markdown, an environment specification (usually a Dockerfile, or a Compose file for multi-container setups), a reference solution, and a tests directory whose `test.sh` script acts as the verifier [2]. A `task.toml` file holds configuration such as timeouts, resource limits, network policy, and which files to preserve as artifacts. The framework defines a trial as one agent’s attempt at one task, and a job as a collection of trials run in parallel [2].

The verifier’s contract with the framework is deliberately minimal. After the agent finishes, the verifier runs and must write a numerical reward to `/logs/verifier/reward.txt`, or a JSON object of labeled metrics to `/logs/verifier/reward.json` [3]. Everything else, including how correctness is judged, is the task author’s responsibility. The documentation describes tasks as “independent, isolated, reproducible pieces of code” with no dependency on the framework itself, closer to versioned software packages than to rows in a dataset [2].

Each component of a task plays a distinct role in the validity of the final reward, and each has a characteristic way of failing.

Component	Role in the evaluation	Characteristic failure
Instruction	Defines the contract the agent is held to	Grading depends on requirements the agent was never told
Environment	Provides the starting state and tools	Hidden dependence on network, time, or mutable remote data
Reference solution	Demonstrates the task is solvable	Reference passes because tests reproduce its own bugs
Verifier	Converts final state into a reward	Accepts wrong work, or rejects valid alternatives
Configuration	Sets isolation, resources, and artifacts	Agent can read or influence grading inputs

Harbor also standardizes the record of what the agent did. Its Agent Trajectory Interchange Format (ATIF) captures messages, reasoning, tool calls, observations, and metrics in a single JSON representation, so that trajectories from different agents can be inspected, compared, and reused without agent-specific parsers [12]. Trajectories are not the reward, but they are the primary evidence when a reward is disputed.

Seen this way, an evaluation is not a container plus a test. It is a set of relationships between five artifacts, any of which can silently invalidate the result. We organize the rest of the discussion around four boundaries those relationships must preserve.

4. Boundary one: execution

The execution boundary separates the agent’s actions from the host and from other trials. This is what people usually mean by “sandboxing,” and it is where the tooling is most mature.

Isolation strength varies by runtime. Standard containers share the host kernel and rely on Linux namespaces and cgroups. Stronger options exist: gVisor, for example, provides an application kernel written in Go that runs in user space and implements a Linux-like interface, interposing between the containerized application and the host [11]. Harbor

abstracts over these choices through a `BaseEnvironment` interface and ships integrations for local Docker and Podman as well as a long list of hosted sandbox providers [1][2]. For evaluation designers, the practical implication is that the same task may execute on runtimes with different kernel features, file-system semantics, and resource enforcement. A task that quietly depends on a specific kernel facility can behave differently on a different provider, which is a determinism problem as much as a security one.

Network access is the most consequential execution decision for agent evaluation, because it determines whether an agent can fetch a solution, contact an external service, or download a dependency that changes over time. Harbor’s network policies make this explicit and phase-aware [4]. A task can set a baseline policy for the environment and override it separately for the agent’s run phase and the verifier phase, choosing among `public`, `no-network`, and `allowlist` modes. The documentation’s motivating example is instructive: an author may want the agent’s installation step to reach the internet, so that a coding agent can be installed into the container, but then restrict the actual rollout to the model provider’s API alone [4]. Harbor applies the agent override only to `agent.run()`, not to `agent.setup()`, precisely so that setup and evaluation can be governed differently.

One design choice deserves emphasis because it encodes a principle that every harness should adopt. If a task requests a network mode that the selected environment does not declare it can enforce, Harbor rejects the trial at validation time “rather than running with a weaker policy” [4]. A sandbox that silently degrades its guarantees produces results that look identical to correctly isolated ones. Failing closed converts a hidden validity problem into a visible configuration error.

The execution boundary is necessary, but it is the boundary least specific to evaluation. Any system that runs untrusted code needs it. The next three boundaries are where agent evaluation becomes its own discipline.

5. Boundary two: information

The information boundary keeps the ground truth out of the agent’s reach. It is distinct from execution isolation: an agent can be perfectly contained within its sandbox and still pass by reading the expected answer, modifying the tests, or tampering with the channel through which the reward is reported.

This failure is not hypothetical. The Agentic Benchmark Checklist (ABC), developed by researchers at the University of Illinois, audited widely used agentic benchmarks and documented concrete cases [5]. In their assessment of SWE-Lancer, agents could bypass test cases stored in password-protected archives; the authors’ recommended fix was to double-zip the test cases with passwords [5]. The checklist’s environment section states the requirement directly: the agent must be “completely isolated from any ground truth information,” and residual data or state must be fully cleared between runs [5].

ImpossibleBench approaches the same problem from the agent’s side. Its authors construct “impossible” variants of coding tasks in which passing the tests necessarily requires specification-violating shortcuts, so that any pass measures an agent’s propensity to exploit the tests rather than its ability to solve the problem [14]. The benchmark’s existence reflects a practical concern: capable agents, given write access to the files that grade them, will sometimes use it. Notably, the ImpossibleBench maintainers disclosed in October 2026 that a bug in one split of their public implementation prevented agents from modifying tests regardless of settings; they note that the bug was introduced when porting the code to GitHub and did not affect the original paper [14]. Even an evaluation designed specifically to measure test tampering needed its own harness audited.

Harbor offers several mechanisms that strengthen this boundary when they are used deliberately. By default, the verifier runs in the same container as the agent, and the `tests/` directory is uploaded only after the agent finishes [3]. That timing keeps the tests out of the agent’s view during its run, but the verifier then executes inside an environment the agent has had full control over. Binaries on the path, Python packages, shared libraries, and the reward directory itself could all have been modified.

A separate verifier environment changes the relationship. With `environment_mode = "separate"`, or a dedicated `[verifier.environment]` section, Harbor grades in a fresh sandbox [3]. Without a dedicated verifier image, Harbor starts a fresh copy of the agent environment, and “the agent’s filesystem changes are not inherited”; with a dedicated image built from `tests/Dockerfile`, the grading code is baked into an image the agent never touches [3]. Only declared artifacts, such as `/app/report.json` or `/logs/artifacts/`, cross from the agent’s sandbox into the verifier’s, restored at their original paths [3][15]. The information that flows into grading becomes an explicit list rather than everything the agent left behind.

This design also forces authors to confront a subtler leak: the artifacts themselves. Any artifact the agent produces is untrusted input to the verifier. Harbor’s own changelog shows the maintainers applying this reasoning inside the framework. When RewardKit, Harbor’s library for building verifiers, began forwarding images and audio from an agent’s trajectory to an LLM judge, the maintainers confined media paths to the trajectory’s own directory, because “the agent under evaluation writes that file, so an absolute path or a `..` must not be able to pull an unrelated local file into a request bound for the judge provider” [6]. The same discipline applies to any verifier that reads paths, checksums, configuration, or manifests produced by the candidate.

A practical test for the information boundary is to ask what an agent could learn, or change, from inside its sandbox. Descriptive fixture names, readable expected outputs, verifier helpers left in the image, environment variables, writable directories shared with grading, and error messages that echo expected values are all channels. Separate verification closes many of them, but only if the verifier treats every transferred artifact as adversarial.

6. Boundary three: judgment

The judgment boundary is the verifier’s ability to separate correct work from incorrect work. It is the heart of outcome validity, and it is where most published benchmark flaws have been found.

ABC defines outcome validity as the requirement that “the success signal (e.g., tests or checks) truly indicates that the task has been completed” [5]. Its audits found violations of varying severity. In the version of τ -bench the authors assessed, a trivial agent that did nothing achieved 38 percent on both `pass@k` and `pass^k`, and an agent that dumped database contents reached 40 percent; the authors released a patch so that the do-nothing agent would no longer score that high [5]. In KernelBench, they report that a fuzzing strategy which varied tensor values but not shapes or memory layouts overestimated agents’ ability to write correct kernels by 31 percent [5]. In WebArena, they document agents passing string-matching or LLM-as-a-judge evaluation without resolving the user’s request [5]. These findings concern specific versions at the time of assessment, and in several cases the benchmark maintainers or auditors have since released fixes. The point is not that these benchmarks were careless. It is that even well-resourced, widely used benchmarks contained verifiers whose rewards did not track the capability they claimed to measure.

6.1 Calibration baselines: the oracle and the no-op

Two baselines should be standard for every task, and Harbor ships both as built-in agents [16]. The oracle agent runs the task’s reference solution. The NOP agent does nothing: its `run` method is literally empty [16]. ABC lists both practices: benchmarks should include an oracle solver that can automatically solve all challenges, and should report the results of trivial agents, such as one that does nothing [5]. Terminal-Bench 2.0’s documentation tells users to begin by running the dataset’s oracle solutions locally [7].

The two baselines bound the verifier from opposite directions. An oracle reward of 1 shows that the task is solvable in its environment and that the verifier accepts at least one correct solution. A NOP reward of 0 shows that the verifier is not satisfied by the starting state. The τ -bench finding illustrates why the second check matters; a do-nothing agent scoring 38 percent is exactly the failure a NOP run is designed to expose [5].

Both baselines are necessary. Neither is sufficient. An oracle pass says nothing about whether incorrect solutions also pass, and a NOP failure says nothing about solutions that do almost everything right but violate one requirement. Most plausible agent errors live in that middle ground.

6.2 Discrimination: one rule, one witness

A more demanding standard is to require that the verifier discriminate on each requirement individually. The question for each rule in the instruction is: could an implementation that violates only this rule still receive a reward of 1? If the answer is yes, that rule is not actually being graded, regardless of how many tests exist.

This is the logic of mutation testing applied to verifiers. For each material requirement, an author constructs a thin mutant, a candidate solution that is correct except for a single deliberate violation, and confirms that the verifier rejects it. ABC's checklist gestures at this when it asks verifiers to measure test quality with objective metrics, to address edge cases, and to generate inputs the code under test is sensitive to [5]. The KernelBench finding is a textbook case of an input domain that was never varied along the dimension where incorrect solutions differed from correct ones [5].

Several patterns recur when authors look for undiscriminated rules. Boundaries are tested on one side only, so an implementation that caps a legal range passes. A field's presence is checked but not its value. One branch of a fallback or one version of a format is tested while its siblings are not. Two rules are each tested in isolation but never together, so an implementation that handles each separately but fails their interaction passes. Aggregate outputs mask a wrong intermediate result. Fixed test corpora allow solutions that dispatch on file names or content hashes. In each case, the remedy is the smallest input that distinguishes the mutant from a correct solution, paired with an assertion that flips when the rule is violated.

To illustrate the method, consider a hypothetical task that asks an agent to implement a configuration loader in which environment variables override a file, which overrides defaults. A verifier that only checks a scenario where all three layers agree, or where only the file is present, will pass a loader that ignores environment variables entirely. A single test that sets conflicting values in all three layers and asserts that the environment value wins turns that mutant into a failure. The test count barely changes; the discriminating power changes completely.

6.3 The opposite failure: overstrict verifiers

Discrimination has a mirror image that is just as damaging and less often discussed. A verifier can reject correct work by enforcing requirements the instruction never stated: a specific output ordering where the contract allows any order, exact whitespace in a serialization that is semantically equivalent, a numerical tolerance tighter than the instruction implies, or a dependence on implementation structure such as particular function names or file layouts. ABC's checklist addresses this from several angles, asking that string matching consider semantically equivalent expressions, that state matching include "all states achievable after success," and that required answer formats be specified

in the task description [5]. Its OSWorld case study documents the result when this goes wrong: tasks an agent could complete correctly but not pass, which the auditors fixed by updating evaluation scripts to match changed external websites [5].

Overstrict verifiers inflate apparent difficulty. A task that most agents fail because of an undisclosed formatting requirement looks hard on a leaderboard but measures compliance with a hidden rule rather than capability. The corresponding discipline is to test valid alternatives: for each place the instruction leaves freedom, construct a different but legitimate solution and confirm it passes. An oracle is one valid solution, not the definition of validity.

6.4 Judges, partial credit, and aggregation

Programmatic tests are not the only verifiers. Harbor supports LLM- and agent-as-a-judge grading, and its RewardKit library packages reusable criteria for files, commands, structured data, HTTP responses, images, and trajectories [3][17]. The RewardKit authors describe the motivation candidly: across popular benchmarks, they found verifiers to be “complicated scripts that are hard to read, extend, and reuse,” and copy-pasted boilerplate “causes subtle errors in verifiers during evals” [17].

Reusable components reduce one class of error but introduce another: aggregation semantics. Harbor’s changelog records a breaking change in which RewardKit’s `all-pass`, `any-pass`, and `required-pass` aggregations now count a score as passing only when it reaches 1.0. Previously, any score above zero passed, so a root `all-pass` over partial-credit criteria “returned 1.0 for partly correct solutions” [6]. This is precisely the kind of verifier defect that an oracle run cannot detect, since the oracle scores full marks either way, and that a thin-mutant battery would expose immediately. It is to the maintainers’ credit that the fix and its rationale are documented publicly.

LLM judges carry additional obligations. ABC asks that judges show documented evidence of accuracy, self-consistency, and agreement with humans, and that they be designed to resist adversarial inputs [5]. In an agentic setting the adversarial input is often the agent’s own output, which may include text addressed to the judge. Where a programmatic check can establish correctness, it should be preferred, and judges should be reserved for properties that genuinely require interpretation.

7. Boundary four: accounting

The accounting boundary separates what the agent did from what the infrastructure did. An evaluation run produces outcomes that are not all equally informative: some trials fail because the agent was wrong, some because it ran out of time, some because a container failed to start, a model API was rate-limited, or the verifier itself crashed. Mixing these into one success rate corrupts both capability estimates and RL reward signals.

Harbor makes this boundary explicit through a single error-classification policy [18]. Each exception type maps to a cause, and the cause determines whether the trial is retried, whether the verifier still runs, and whether the reward is valid, meaning whether it counts in benchmark and RL results. Agent timeouts and context-window exhaustion are classified as `agent_exhausted_budget`: the verifier runs and the reward is valid, because running out of budget is a property of the agent [18]. Sandbox failures such as image build errors or out-of-memory conditions are retried and produce no valid reward. Model provider errors such as rate limits are retried and invalid. Verifier timeouts and missing reward files are classified as `verifier_failed` and do not produce a valid reward [18].

This taxonomy encodes a judgment that every evaluation must make, whether explicitly or not. The alternative, counting every non-pass as a failure, quietly converts infrastructure noise into apparent difficulty. A benchmark run on a congested cloud provider would look harder than the same benchmark run locally.

7.1 Determinism

Accounting depends on determinism. If the same agent output can receive different rewards on different runs, the reward is measuring something other than the work. ABC lists flaky end-to-end tests, unspecified tool versions, and inconsistent API availability among its validity concerns [5].

The Terminal-Bench 2.0 corpus offers a useful, concrete view of how verification dependencies were handled before separate verifiers became common. In the public repository as of commit 2fd12b8 (29 April 2026), all 89 tasks use schema version 1.1 with internet access enabled, and 82 of the 89 verifier scripts run `apt-get` and download a pinned version of the `uv` installer at verification time before running pinned test dependencies [7]. The pinning reflects careful practice, and the design was reasonable for a shared-container verifier. Even so, installation at grading time ties every reward to the availability and contents of external package mirrors at the moment of grading. Harbor’s documentation now recommends the alternative: a separate verifier image with dependencies installed ahead of time, which “can reduce installation flakiness and speed up the verifier phase” [3].

Separate verification also unlocks regrading. Harbor can rerun an updated verifier against the recorded outputs of completed trials without re-running the agent, which requires no agent credentials and incurs no new agent cost [19]. This turns verifier development into an iterative, testable process: when a verifier gap is found, the fix can be applied retroactively to existing trajectories and the change in scores measured directly. It also makes verifier changes auditable, since the effect of each change on historical results is visible.

Other sources of nondeterminism are familiar from software testing and remain relevant: uncontrolled random seeds, wall-clock time, file-system ordering, leftover outputs from earlier steps, races between processes, and fixed sleeps used in place of readiness checks. Each should be eliminated or bounded by tolerances that the instruction makes explicit.

8. Measuring difficulty honestly

Benchmarks are useful only if they are hard enough to separate systems, so authors are under constant pressure to make tasks difficult. The four boundaries above determine whether that difficulty is genuine.

A useful classification of failed trials distinguishes failures caused by the agent’s reasoning or skill from failures caused by the task: ambiguity in the instruction, hidden requirements, overstrict verification, flaky tests, broken environments, or unfair resource limits. Only the first kind measures capability. A task that fails most agents on the same one or two tests deserves scrutiny before it is called hard; concentrated failure is as often a sign of an undisclosed requirement as of a real crux.

Legitimate difficulty tends to come from the substance of the task: domain knowledge an expert would bring, constraints that interact so that satisfying one complicates another, inference from evidence available in the environment, configuration precedence, nested structures, plausible decoys, and verifiers that exercise public rules on held-out inputs. Terminal-Bench 2.0’s own metadata reflects this orientation, recording expert time estimates alongside difficulty labels for 88 of its 89 tasks [7].

Sample size is the final constraint. Agent runs are expensive, so difficulty is often estimated from a small number of trials. Small samples are noisy in ways that are easy to underestimate. As an illustrative calculation using the Wilson score interval at 95 percent confidence, an agent that passes 5 of 8 trials has an observed success rate of 62.5 percent, but the data are compatible with a true rate anywhere from about 31 to 86 percent. Three passes in eight is compatible with roughly 14 to 69 percent. Even a perfect 8 of 8 is compatible with a true rate as low as about 68 percent. With 32 trials, 20 passes narrows the interval to roughly 45 to 77 percent.

Observed passes	Trials	Observed rate	Approx. 95% Wilson interval
3	8	37.5%	14% to 69%
5	8	62.5%	31% to 86%
8	8	100%	68% to 100%
20	32	62.5%	45% to 77%

Table: Illustrative uncertainty for per-task success rates. Values are computed from the Wilson score formula, not from experimental data.

Two practical implications follow. First, per-task difficulty estimates from a handful of runs should be treated as coarse, and decisions near a threshold should allow for the possibility that a different batch would land on the other side. Second, aggregate benchmark results should report uncertainty. ABC lists reporting statistical significance, such as confidence

intervals, among its reporting requirements [5]. The pass^k metric used in τ -bench, which ABC also reports, offers a complementary view by asking whether an agent succeeds consistently across repeated attempts rather than at least once [5].

9. A working checklist for sandboxed agent tasks

The four boundaries translate into a compact set of questions that a task author or reviewer can apply before publishing a task. The checklist below synthesizes Harbor’s documented features and ABC’s audit criteria with the practices proposed in this article; it is not an official specification of either project.

Boundary	Question to ask	Supporting practice
Execution	Does the sandbox enforce the network and resource policy the task assumes?	Phase-specific network policy; fail closed on unsupported modes [4]
Information	Can the agent read, write, or predict anything the verifier relies on?	Separate verifier image; explicit artifacts; treat artifacts as untrusted [3][15]
Judgment	Does violating any single stated rule flip the reward to 0?	Oracle and NOP runs; thin-mutant battery per requirement [5][16]
Judgment	Does every contract-valid alternative still receive reward 1?	Valid-alternative battery; disclose formats and tolerances [5]
Accounting	Are infrastructure errors kept out of capability scores?	Explicit error classification and retry policy [18]
Accounting	Will the same output receive the same reward tomorrow?	Baked verifier dependencies; controlled randomness and time; regrading [3][19]
Difficulty	Do failures come from the task’s substance rather than its defects?	Classify failures by cause; report uncertainty [5]

Two principles run through every row. The first is that discrimination matters more than test count. A verifier with ten tests that each check presence is weaker than one with three tests that each isolate a rule. The second is that wrong work must fail while valid alternatives pass. A verifier that achieves only one of these properties is not sound.

10. Open research questions

Several problems remain open, and we present them as proposed research directions rather than completed work.

Automated mutant generation for verifiers. Constructing thin mutants by hand is labor-intensive. A natural direction is to use language models to generate candidate violations of each stated requirement and measure the fraction a verifier rejects, analogous to mutation scores in software testing. The research challenge is ensuring generated mutants are genuinely plausible rather than trivially broken, since trivially broken mutants overstate verifier strength.

Cross-runtime determinism. As harnesses support many sandbox providers with different kernels and isolation mechanisms, the same task may yield different outcomes depending on where it runs. Systematic studies that execute oracle and NOP baselines, and fixed agent outputs, across providers would quantify how much provider choice contributes to score variance.

Verifier robustness to adversarial agents. ImpossibleBench shows that agents can be measured for their propensity to exploit tests [14]. A complementary direction is to measure verifiers for their resistance: deliberately prompting agents to pass without solving, and recording which verifier designs withstand the attempt. Separate verification closes many direct channels, but artifacts and judge inputs remain attack surfaces.

Trajectory-aware grading. Standard formats such as ATIF make it practical to grade how an agent reached its result, not only the final state [12]. Trajectory evidence could detect tampering or shortcutting that outcome checks miss. It also creates new risks, including judges influenced by agent-authored text and grading criteria that penalize legitimate but unusual strategies.

Statistical design for difficulty calibration. Given the width of small-sample intervals, there is room for sequential designs that allocate additional trials only to tasks whose observed difficulty sits near a decision boundary, reducing cost while controlling misclassification.

11. Conclusion

Sandboxing solved an important problem. Agent evaluations that once depended on fragile local setups can now run in thousands of isolated, reproducible environments, and frameworks such as Harbor and Inspect make that capability broadly available [1][13]. Harbor’s task format is especially valuable because it separates instruction, environment, reference solution, and verifier into explicit artifacts that can be reviewed independently, and because features such as separate verifiers, phase-aware network policies, explicit error classification, and regrading give authors concrete tools for each of the boundaries discussed here [3][4][18][19].

The tools do not make an evaluation valid by themselves. A task can run in a perfectly isolated sandbox and still reward a do-nothing agent, leak its expected output, reject a correct solution because of an unstated formatting rule, or count a rate-limited API call as a

capability failure. The published audits of widely used benchmarks show that these failures occur in practice, including in projects built by careful teams [5].

For researchers building benchmarks, the practical recommendation is to treat the verifier as the primary research artifact: test it against thin mutants and valid alternatives, isolate it from the agent, keep its dependencies fixed, and report uncertainty honestly. For model developers, the recommendation is to read benchmark results through the same lens and to prefer evaluations whose authors document how they validated their verifiers. For organizations buying or deploying agents, the question to ask of any reported score is not only where the agent ran, but how the evaluation established that a reward of 1 meant the work was done.

References

-
- [1] Harbor Framework Team. *Harbor: A framework for evaluating and optimizing agents and models in container environments* (software, v0.24.0, released 5 October 2026). GitHub repository and README. <https://github.com/harbor-framework/harbor>. Concept DOI: <https://doi.org/10.5281/zenodo.20953922>
 - [2] Harbor documentation. “Core concepts” and “Tasks: Overview.” <https://github.com/harbor-framework/harbor/blob/main/docs-mintlify/core-concepts/index.mdx> and <https://github.com/harbor-framework/harbor/blob/main/docs-mintlify/tasks/overview.mdx>
 - [3] Harbor documentation. “Verifier” and “Separate verifier.” <https://github.com/harbor-framework/harbor/blob/main/docs-mintlify/tasks/verifier.mdx> and <https://github.com/harbor-framework/harbor/blob/main/docs-mintlify/tasks/separate-verifier.mdx>
 - [4] Harbor documentation. “Network policies.” <https://github.com/harbor-framework/harbor/blob/main/docs-mintlify/tasks/network-policies.mdx>
 - [5] UIUC Kang Lab. *Agentic Benchmark Checklist: Establishing Best Practices in Building Rigorous AI Agent Benchmarks*. Repository README, checklist (ABC.md), and benchmark case studies. <https://github.com/uiuc-kang-lab/agentic-benchmarks> and <https://github.com/uiuc-kang-lab/agentic-benchmarks/blob/main/ABC.md>
 - [6] Harbor changelog (entries on RewardKit pass-style aggregations and trajectory media forwarding). <https://github.com/harbor-framework/harbor/blob/main/CHANGELOG.md>
 - [7] Laude Institute. *Terminal-Bench 2.0* task repository, README, and FAQ (task corpus examined at commit 2fd12b8, 29 April 2026). <https://github.com/laude-institute/terminal-bench-2>
 - [8] M. A. Merrill, A. G. Shaw, N. Carlini, et al. “Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces.” *The Fourteenth International Conference on Learning Representations (ICLR)*, 2026. <https://openreview.net/forum?id=a7Qa4CcHak>. Original harness: <https://github.com/laude-institute/terminal-bench>
 - [9] C. E. Jimenez, J. Yang, A. Wettig, et al. “SWE-bench: Can Language Models Resolve Real-World GitHub Issues?” *The Twelfth International Conference on Learning Representations (ICLR)*, 2024. arXiv:2310.06770. <https://arxiv.org/abs/2310.06770>. Repository, containerized harness (June 2024), and SWE-bench Verified (August 2024): <https://github.com/SWE-bench/SWE-bench>
 - [10] Python Package Index. “harbor” release history (first release 0.1.0, 2 November 2025). <https://pypi.org/project/harbor/>
 - [11] Google. *gVisor* repository and README. <https://github.com/google/gvisor>

- [12] Harbor documentation. “ATIF: Agent Trajectory Interchange Format” and RFC 0001. <https://github.com/harbor-framework/harbor/blob/main/docs-mintlify/agents/atif.mdx> and <https://github.com/harbor-framework/harbor/blob/main/rfcs/0001-trajectory-format.md>
- [13] UK AI Security Institute. *Inspect* framework README and “Sandboxing” documentation. https://github.com/UKGovernmentBEIS/inspect_ai and https://github.com/UKGovernmentBEIS/inspect_ai/blob/main/docs/sandboxing.qmd
- [14] *ImpossibleBench* repository and README, including the October 2026 bug notice; paper at arXiv:2510.20270. <https://github.com/safety-research/impossiblebench> and <https://arxiv.org/abs/2510.20270>
- [15] Harbor documentation. “Artifacts.” <https://github.com/harbor-framework/harbor/blob/main/docs-mintlify/tasks/artifacts.mdx>
- [16] Harbor source and documentation for built-in agents, including the oracle and NOP agents. <https://github.com/harbor-framework/harbor/blob/main/src/harbor/agents/nop.py> and <https://github.com/harbor-framework/harbor/blob/main/docs-mintlify/agents/pre-integrated-agents.mdx>
- [17] Harbor documentation. “RewardKit: Motivation and design.” <https://github.com/harbor-framework/harbor/blob/main/docs-mintlify/rewardkit/motivation-and-design.mdx>
- [18] Harbor documentation. “Error classification.” <https://github.com/harbor-framework/harbor/blob/main/docs-mintlify/jobs/error-classification.mdx>
- [19] Harbor documentation. “Regrade.” <https://github.com/harbor-framework/harbor/blob/main/docs-mintlify/jobs/regrade.mdx>