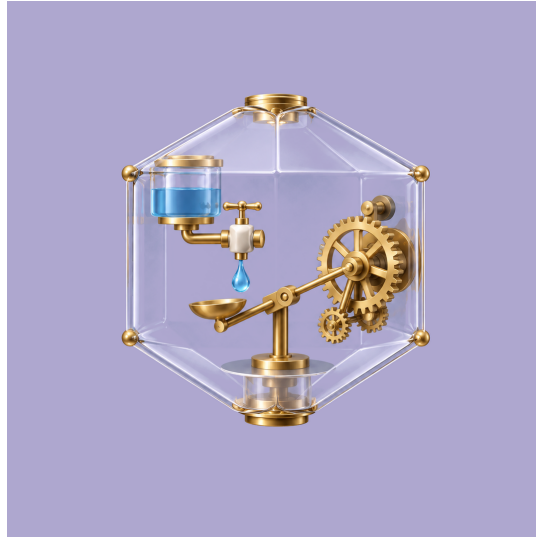




The Environment Is the Experiment: Causal Fidelity in Reinforcement Learning for AI Agents

Why faithful state transitions, partial observability, and delayed consequences matter more than interfaces that merely look real.

Ahsan Raza | Deaimer Research

Abstract

Reinforcement learning (RL) agents learn from the consequences of their actions. For tool-using language agents, those consequences are not confined to simulated movement or a numerical reward: a command changes a repository, an API call changes account state, and an apparently successful repair may fail after a restart. A training environment that imitates the appearance of these workflows but omits their causal structure can reward strategies that are ineffective outside the simulator. This article develops **causal fidelity** as a research perspective for agent environments: the degree to which the environment preserves task-relevant relationships among actions, hidden state, observations, and downstream effects. Drawing on partially observable decision processes, Gymnasium’s environment contract, and systems such as WebArena, AppWorld, ToolSandbox, τ -bench, and SWE-Gym, we distinguish realistic presentation from faithful behavior. We describe failure modes in state transitions, information visibility, delayed effects, resets, and counterfactual branching. We then propose an environment-audit protocol and paired training experiment designed to test whether fidelity affects behavioral transfer. These protocols are proposals, not completed experiments. The practical argument is that an executable training world must be judged not only by whether an agent can act within it, but by whether the world responds in ways that teach the intended capability.

1. Introduction: the world is part of the policy

Imagine an AI agent asked to repair a failing data-ingestion service. It can read code, inspect logs, query a database, edit a configuration file, and restart a process. In a simplified training environment, the service becomes healthy as soon as the agent writes a plausible con-

figuration value. In the deployed system, the same edit also affects a retry queue, changes a consumer’s schema expectations, and may be overwritten when a process restarts. The two environments present nearly identical instructions and tools, but they teach different lessons.

The first teaches that success follows a recognizable edit. The second requires the agent to discover dependencies, predict side effects, check persistence, and adapt when the initial intervention changes another component. An agent can optimize effectively for the first world and still be poorly equipped for the second. That is a failure of **environment validity**, not necessarily of optimization.

Reinforcement learning makes this distinction especially important. In supervised learning, examples are selected and labeled before training. In RL, the environment produces new experience in response to the learner’s evolving behavior. A policy determines which actions occur; those actions determine which states the policy will see next. The training world therefore participates in generating the effective dataset, including its blind spots. Sutton and Barto’s account of RL places this interaction between agent and environment at the center of the discipline [1].

The growing literature on tool-using language agents has made environments more executable and consequential. WebArena brought self-hosted websites into a reproducible agent benchmark [4]. AppWorld constructed interdependent application APIs with state-based tests that also check collateral changes [5]. ToolSandbox explicitly targeted state dependencies between tool calls [6]. SWE-Gym paired real software-engineering tasks with runnable repositories and tests for agent training [8]. These contributions are significant, but they do not eliminate a general research question: **which aspects of the world’s behavior must be preserved for learning to transfer?**

This article proposes causal fidelity as a useful lens for answering it. The claim is not that every RL environment should recreate a production system. That would be expensive, slow, unsafe, and often scientifically unnecessary. The claim is narrower: simplifications should preserve the distinctions the target skill depends upon. A training world is faithful enough when the actions that work for the right reasons in that world are also the actions whose relevant consequences the target setting would support.

This framing extends, rather than repeats, questions about benchmark verification. A verifier can accurately grade a task within a simulator even when the simulator itself encodes the wrong dynamics. If an environment instantly declares a deployment healthy after a superficial edit, a perfectly correct verifier for that simplified environment still supplies the learner with an impoverished lesson.

2. A formal lens: transitions, observations, and objectives

A common starting point for RL is a Markov decision process. Let a state be s_t , an action be a_t , and a transition kernel be $P(s_{t+1} \mid s_t, a_t)$. The reward function $r(s_t, a_t, s_{t+1})$ describes the immediate training signal. A policy chooses actions to maximize expected cumulative reward. In the setting of language agents, the action may be a shell command, a function call, a browser operation, or a request for information.

The Markov description is helpful for environment authors even when the agent cannot directly see the state. In many practical systems, the learner instead receives an observation

o_t , produced from state through an observation process $O(o_t \mid s_t)$. It must act from a history of observations and actions, rather than a complete view of hidden state. The established framework for such problems is a partially observable Markov decision process (POMDP) [2].

A useful environment specification therefore separates at least five objects:

Object	What it represents	Example in a software-agent task
Hidden state	Everything that can affect future behavior	Disk content, process status, queue position, permissions
Action	An intervention the agent is allowed to make	Edit file, restart service, query API
Transition	How the system changes after an intervention	Worker reloads config; pending jobs are retried
Observation	What the agent is actually shown	Terminal output, logs, API response
Objective and reward	What counts as useful progress or completion	Durable recovery without corrupting queued jobs

A sixth object—the initial-state distribution—determines which combinations of these conditions agents encounter. Two environments can provide the same textual prompt and action names while differing radically in that distribution and in their transition rules. Their learning problems are then different.

For the purposes of this article, **causal fidelity is task-relative**. It is not photographic resemblance, identical source code, or exact preservation of every production detail. It asks whether interventions preserve the consequences that matter to the intended learning objective. An environment modeling a customer-service workflow need not reproduce an entire commercial billing platform. It does need to preserve whether a refund changes a ledger, whether two refunds can be issued for the same transaction, and whether later queries reveal the change, if those distinctions are part of the target skill.

One can express the concern as a relationship between two transition systems: P_{train} , the training-world dynamics, and P_{target} , the dynamics of the intended application domain. A mismatch does not automatically invalidate learning. Indeed, controlled simplification is often the point of simulation. The risk arises when the mismatch changes which policies succeed. Even small one-step discrepancies can accumulate across long horizons; however, any quantitative bound would require assumptions about the policy, observability, horizon, and transition distances. No universal transfer guarantee follows merely from labeling a simulator “realistic.”

3. Boundary one: actions must have the right effects

Many agent environments start by exposing an action interface: a terminal, a browser, or a collection of tools. A tool schema says what an operation is called and how to invoke it. It does not, by itself, tell us whether the operation has realistic consequences.

Consider an illustrative API named `reserve_inventory(item, quantity)`. A shallow simulator might always return a receipt and decrement a quantity field. A state-faithful simulator must decide what happens when the remaining stock is insufficient, when two reservations race, when the request is retried, when authorization has expired, and when a later cancellation occurs. The agent’s strategic task is partly to reason about those conditions.

A useful design method is to write **transition invariants** before implementing the tool. Inventory cannot be negative unless backorders are explicitly modeled; an authorized reservation changes exactly one ledger event; retrying a request with the same idempotency key does not create an unintended duplicate. These are not arbitrary hard-coded test cases. They describe relationships that should remain true across many possible trajectories.

AppWorld is instructive because its original 2024 paper describes multiple interconnected everyday applications rather than isolated API calls. Its programmatic evaluation checks not only desired final outcomes but also unexpected state changes [5]. The broader design lesson is that a successful action may have **both required effects and prohibited collateral effects**. In a training environment, the distinction needs to exist in state and not solely in a textual instruction.

A second problem is **action aliasing**. Two commands may look different while secretly producing the same simplified simulator update. If `restart_service` and `reload_config` both set a `healthy=true` flag, an agent cannot learn the operational difference between them. Conversely, two actions that ought to be equivalent under the task contract should not behave incompatibly because they happen to enter separate branches of simulation code. Designing a world requires identifying which equivalences should hold and which distinctions must remain observable.

A third problem is **missing negative consequences**. Simulators often model the happy path first: write data, receive success, continue. Yet refusal, validation errors, throttling, concurrent writes, and partial transactions are integral to the target capability. A policy that never encounters the consequences of an unauthorized change may learn that checking authorization is wasted effort. This is not an argument for injecting arbitrary errors. Failure conditions should correspond to clear causal rules that the agent could learn to recognize and handle.

4. Boundary two: observation is not ground truth

An operational agent rarely sees a complete, trusted state representation. It receives logs that may lag reality, API messages that describe only one subsystem, and observations that differ in granularity. POMDP theory formalizes the resulting gap between the latent state and what the agent knows [2].

For training, this means the observation channel must be designed as carefully as the transition channel. A simulator may track rich state internally and then accidentally reveal it through diagnostic output. An API that returns `expected_final_state` or reports whether an agent’s current plan will succeed converts a discovery problem into answer retrieval. The opposite error is possible: the simulator may hide information that real operators would reasonably access, forcing guessing instead of diagnosis.

A practical audit asks three questions. First, **which facts are observable at the beginning?** Second, **which facts can an agent discover through authorized actions?** Third,

which facts should remain inaccessible even after extensive exploration? The distinction matters because useful agent behavior often consists of deciding what to inspect before intervening.

ToolSandbox provides a concrete research precedent. Its 2024 design includes stateful tool execution, dependencies among tool calls, a user simulator, and evaluation of intermediate as well as final milestones [6]. Those properties make observation order and information gathering part of the interaction rather than assumptions hidden in a static prompt. In τ -bench, simulated conversations, policy instructions, and domain-specific tools similarly require an agent to coordinate information from users with actions that alter database state [7]. These are evaluation environments, not automatic demonstrations that a particular RL curriculum transfers; their architectures nevertheless expose the design issues a training environment must handle.

Observation fidelity also requires **temporal fidelity**. A log line can describe a state that was true before an action. A monitoring endpoint may converge later than the database behind it. A command may report successful submission while asynchronous execution remains pending. If a simulator answers every query with the most recent complete global truth, it can unintentionally remove the skill of reconciling conflicting evidence over time.

An illustrative remedy is to specify observation timing independently from state updates. After `submit_job`, the job may enter `queued`, later become `running`, and only then reach `completed` or `failed`. A status query before completion should not reveal the final outcome. The environment need not introduce random latency everywhere; a deterministic, reproducible event schedule can be enough to preserve the relevant reasoning problem.

5. Boundary three: consequences persist beyond the immediate response

Long-horizon tasks often fail for reasons that are invisible at the instant an action is taken. An edit appears valid but breaks a dependent client. A migration succeeds once but is not safe to rerun. A file is written but not flushed before a simulated crash. An agent must learn that the objective concerns the system’s *durable* state, not a momentary green indicator.

The failure mode is especially acute when environment authors collapse a workflow into a single success predicate. Suppose a task asks an agent to rotate credentials while keeping an application online. A shallow world may award success when a configuration file contains the new value. A richer world models the credential provider, workers that cache credentials, active sessions, and a safe transition interval. The agent must then consider ordering: provisioning a new credential before revoking the old one can matter.

The relevant concept is **causal dependency**, not simply more simulation steps. If the environment introduces many steps that do not change the strategic choices available, it adds cost but little learning value. Conversely, one carefully modeled delayed dependency can transform the problem from pattern completion into planning.

SWE-Gym, published at ICML 2025, shows why executable software context matters for agent training. Its authors assemble 2,438 Python software-engineering tasks with runtime environments, natural-language requests, and unit tests, and report gains for their trained agents on designated SWE-bench evaluations [8]. That published result supports the value

of executable interaction in the specific experimental setup. It does not establish that every component of every generated environment reproduces operational deployment behavior, nor that the gains would transfer to unrelated domains. These are separate empirical questions.

A task author’s design question should therefore be: *which changes must still matter after the agent stops interacting?* For data workflows the answer might include referential integrity and replayability; for infrastructure it might include restart behavior and service availability; for web workflows it might include the later visibility of a changed account setting. If an environment deliberately excludes such consequences, the target learning objective should be narrowed accordingly.

6. Boundary four: reset, replay, and branching must be credible

RL training revisits environments repeatedly. The `reset()` operation is therefore not a convenience; it is part of the experimental definition. Gymnasium’s documented interface separates `reset()` from `step()` and, since the newer step API, distinguishes a true terminal state from truncation caused by an external limit [3]. The difference matters for learning algorithms that bootstrap values at episode boundaries.

For agent environments backed by files, processes, or databases, a genuine reset must restore **all causally relevant state**: persistent storage, caches, queues, credentials, process memory, background jobs, and pseudorandom generators where controlled seeding is required. Removing one visible output file is not enough if a database sequence counter or idempotency record survives between episodes.

Three checks are particularly informative:

1. **Replay**: starting from the same snapshot and seed, does the same action sequence yield equivalent states and observations under the documented determinism assumptions?
2. **Independence**: does running trajectory A before trajectory B change B’s outcome compared with running B from a fresh baseline?
3. **Branching**: after copying one intermediate state, can two different next actions be compared without the branches contaminating one another?

The third check supports counterfactual inquiry. If an agent chooses to restart a worker, a researcher may want to ask what would have happened had it reloaded configuration instead. A branchable environment helps identify the causal effect of an intervention; a simulator that silently shares a queue between branches cannot answer the question reliably.

These properties are not guaranteed by containerization alone. Containers may isolate process namespaces, but an experiment can still depend on shared remote services, time-dependent package downloads, external API responses, and persistent state mounted outside the container. Reproducible training requires an explicit inventory of those dependencies.

Gymnasium’s environment-creation guidance recommends deterministic seeding through `reset(seed=...)` and the environment’s own random generator [9]. The analogous practice for software agents is to record the initial snapshot, dependency versions, event-sequence seed, and any intentional nondeterminism. Full bitwise equality may be unnecessary when

the task only requires semantic equivalence, but the criterion must be named rather than assumed.

7. What existing environments teach us—and what they do not

Several research systems embody pieces of causal fidelity. Comparing them is more informative than treating “realistic” as a single score.

System	Relevant architectural contribution	Question still requiring separate study
WebArena [4]	Self-hosted, functional websites and long-horizon web tasks	Which simulated dependencies matter most for transfer beyond the hosted sites?
AppWorld [5]	Interdependent application state and collateral-change checks	How do different state abstractions alter policy learning?
ToolSandbox [6]	Stateful tool calls and implicit ordering dependencies	Which tool-state failures become learnable under RL?
τ -bench [7]	Simulated users, domain rules, tools, and final database state	How sensitive are learned behaviors to user-simulator assumptions?
SWE-Gym [8]	Runnable repository tasks suitable for agent training	Which task and runtime factors explain cross-repository transfer?

WebArena’s maintainers have also recommended the AgentLab/BrowserGym infrastructure in their December 2024 repository update to improve parallel experimentation, integrations, and handling of edge cases [4]. This matters methodologically: one should discuss the current recommended tooling rather than treating early harness limitations as permanent properties of a research program.

The systems differ in purpose. Some were introduced primarily as **evaluations**; others explicitly support **training**. A benchmark demonstrating that a setting challenges existing models does not by itself demonstrate that training within that setting improves deployed behavior. It is necessary to separate the validity of the environment, the correctness of the reward, the behavior of the optimizer, and the transfer of the resulting policy.

Recent procedural approaches make this distinction particularly pressing. RandomWorld, published at EMNLP 2025, generates interactive tools and compositional tool-use data and reports downstream improvements for models trained with its generated data [10]. This is evidence that generated interaction structures can be useful; it also increases the importance of understanding which generated causal patterns generalize and which remain peculiar to a synthetic tool universe.

One tempting response is to make every training environment maximally detailed. That would be misguided. Real deployments contain innumerable irrelevant details. An envi-

ronment about safe refunds need not model a retailer’s office lighting or page animations unless those features change the target decisions. Environment designers should preserve **decision-relevant causal relationships**, not indiscriminate complexity.

8. An illustrative failure: the service that only looks repaired

Consider the following hypothetical environment pair. Both present an agent with the instruction: “Restore the event-processing service without losing accepted events; ensure recovery survives restart.” Both permit log inspection, configuration editing, event replay, and service restart. Both start with a malformed retry policy and a backlog of accepted events.

World A: appearance-faithful, causally shallow. Editing the retry policy to a syntactically valid value immediately marks the service healthy. The environment exposes a plausible success log, but the backlog is not modeled. `restart_service` simply preserves the boolean healthy flag. If the agent uses a risky setting, no downstream loss can occur because events have no independent lifecycle.

World B: task-causally faithful. Accepted events have identifiers and durable queue positions. The worker acknowledges an event only after its target write commits. Retry behavior controls how failures are rescheduled. Restart reconstructs work from persisted queue state. A superficially attractive retry setting may stop the visible error yet discard unacknowledged events under a specific failure ordering.

An agent that learns in World A can earn consistent high reward by editing one configuration value and stopping. In World B, the same strategy could be incomplete. A more appropriate strategy might inspect queue state, change the retry policy, restart, replay a controlled set of events, and reconcile acknowledgments against persisted writes.

The example is deliberately not presented as a measurement. It illustrates a **policy-ranking reversal**: a policy that looks optimal under one transition model may cease to be optimal when the state machine preserves the relationships relevant to the objective. The experiment that would establish such a reversal is straightforward to specify, but it has not been run here.

It is also possible to overcorrect. If World B imposes arbitrary restart delays or undocumented event-order quirks that do not exist in the target domain, it may favor fragile simulator-specific tactics. Causal fidelity requires documenting the target abstraction and validating each added mechanism, not merely adding opportunities for failure.

9. Proposed causal-fidelity audit

We propose an audit built around **causal contracts**. A causal contract states what a family of actions must do to task-relevant state and what the agent should be able to observe. It is more general than enumerating one successful final artifact and more specific than asserting that a simulator feels realistic.

Step 1: identify decisions. List the decisions the intended capability requires: whether to inspect before editing, whether to retry, whether to request authorization, when to restart, and how to verify persistence. Exclude environmental details that cannot affect these decisions.

Step 2: construct a state dependency map. For each decision, record affected state vari-

ables and downstream dependencies. In the illustrative service, a retry-policy edit influences retry scheduling, queue age, processing order, and eventual writes. The map need not be a complete representation of the target system; it should explain why each included edge matters.

Step 3: create paired interventions. From one controlled state, apply two distinct authorized actions and test whether the resulting difference agrees with the contract. Examples include retrying the same request with and without an idempotency key, or restarting a service before and after persisting a checkpoint.

Step 4: add metamorphic checks. Some properties can be tested without knowing every exact expected result. Reordering independent operations should leave relevant final state unchanged; unauthorized operations should leave protected state unchanged; repeating an idempotent operation should not duplicate its effect. Such relations help reveal simulator shortcuts.

Step 5: test observability. Verify that hidden state is not exposed prematurely, that authorized inspection can reveal needed facts, and that observations arrive according to documented timing. Test both excessive information leakage and insufficient information access.

Step 6: test reset and temporal behavior. Replay fixed trajectories from fresh baselines, interleave unrelated trials to detect leakage, and perform restart or persistence probes where relevant. Treat nondeterministic phenomena as distributions requiring measured stability, not as excuses to skip testing.

Step 7: compare acceptable strategies. Determine whether the world preserves more than one legitimate way to achieve the objective. A simulator can accidentally force a particular command sequence when the target system would allow equivalent repairs. An overly prescriptive environment teaches conformity to the simulation rather than flexible problem solving.

The output of this proposed audit would be a *fidelity dossier*: a concise record of the modeled state, intentional simplifications, observable signals, supported causal invariants, known divergences, and tests used to establish those claims. It would not be a universal scalar certifying realism. Fidelity is meaningful only relative to a defined target family of tasks.

10. Proposed experiment: isolate the environment’s contribution

A useful empirical study would compare policies trained under different environment dynamics while holding other elements as constant as practical. We propose a two-by-two design. The first factor is **causal fidelity** (shallow versus contract-faithful state transitions). The second is **surface variability** (fixed versus varied names, layouts, and incidental messages). This separates a change in the world’s logic from a change in its appearance.

Each training arm would receive a matched budget of interaction steps, model updates, and compute. Where exact equality is impossible—because one environment requires more expensive actions—the study should report both wall-clock and environment-step budgets. All arms would use the same family of task objectives and reward criteria compatible with their stated environment contracts. A strong evaluation requires a separate, higher-fidelity held-out environment with unseen initial states and mechanisms not trivially inferable from training identifiers.

The principal outcome should not be training reward alone. Report transfer success on the held-out world, durability after restart, forbidden side effects, the frequency of useful information-gathering actions, and the rate at which the agent recognizes incomplete evidence. A secondary outcome is how rapidly policies adapt when a known mechanism changes. Statistical estimates should be accompanied by uncertainty intervals and repeated runs with independently seeded task instances.

Ablations can then answer more precise questions. Does preserving queue persistence improve transfer while adding realistic log wording does not? Does training with delayed observations improve the agent’s use of diagnostic tools? Does full state fidelity help only when combined with enough initial-state diversity? These are hypotheses to test, not conclusions asserted in advance.

The design must also anticipate confounds. If the faithful environment takes longer to run, the lower number of trajectories may offset any advantage. If the evaluator shares source code with the faithful simulator, both can reproduce the same bug. If test cases resemble the training generator, a reported gain may represent interpolation within one synthetic world. The protocol should disclose these risks and use independent implementations or cross-checks for critical mechanisms.

11. Fidelity has a cost; abstraction has a purpose

There is no single optimal fidelity level. A complete enterprise application may be costly to license, maintain, reset, and secure. Simpler worlds allow faster iteration, more controlled comparisons, and cleaner diagnosis of a particular causal relation. A sequence of environments of increasing complexity can be scientifically valuable if each stage preserves the mechanism under study.

One useful design principle is **minimal sufficient dynamics**. Start with the smallest state machine in which the intended skill is genuinely necessary. Add state and dependencies when a plausible shortcut can otherwise succeed for the wrong reason. Remove incidental detail that consumes budget without changing policy choices. This keeps difficult tasks difficult because of reasoning, not because of arbitrary friction.

The approach also clarifies the role of synthetic data. A synthetic environment can be more useful than an uncontrolled real service when it supports precise intervention, clean reset, and reproducible counterfactuals. Conversely, access to real software is not enough if the training setup strips away the state changes that make the workflow meaningful. The distinction is not “synthetic versus real”; it is **modeled causal structure versus target decision requirements**.

12. Implications for Deaimer and environment buyers

Deaimer publicly describes work on custom RL environments, terminal-style task packages, and outcome-driven agent evaluations. Its Termetheus research article frames executable tasks as environments containing objectives, initial state, tools, dependencies, and evidence of completion [11]. Causal fidelity offers a complementary way to specify what such a package should model: not just which tools exist, but which consequences their use must preserve.

For teams commissioning environments, the practical request is to identify target decisions, observable state, persistent effects, and known simulation boundaries before deciding how many tasks to generate. This is a proposed purchasing and research criterion, not a claim that Deaimer has already implemented or empirically validated the entire audit outlined above. Teams exploring custom environments can discuss task specifications and validation goals through Deaimer’s public website.

13. Conclusion

An RL environment is not a neutral container surrounding an agent. It determines which actions are possible, what happens after those actions, what the learner can discover, and which experiences enter training. A convincing interface can still encode misleading dynamics. A carefully simplified environment can be powerful when its abstractions preserve the decision-relevant causal relationships.

The existing literature has established useful building blocks: partially observable models for reasoning under uncertainty, standardized environment contracts, reproducible websites, connected application state, stateful tool dependencies, and executable software-training tasks [1–10]. The open question is how to establish **which particular causal mechanisms** matter for transfer in a given agent domain.

The proposed answer is to treat environments as experimental instruments. Specify the behavior they claim to model; challenge those claims through paired interventions, temporal probes, replay, and branching; and evaluate the resulting policies outside the training world’s familiar surface. The strongest environment is not necessarily the most elaborate. It is the one whose consequences teach the intended skill—and whose limitations the researcher can explain.

References

- [1] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*, second edition. MIT Press, 2018. <https://mitpress.mit.edu/9780262039246/reinforcement-learning/>
- [2] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. “Planning and acting in partially observable stochastic domains.” *Artificial Intelligence* 101(1–2), 99–134, 1998. [https://doi.org/10.1016/S0004-3702\(98\)00023-X](https://doi.org/10.1016/S0004-3702(98)00023-X)
- [3] Farama Foundation. *Gymnasium: Env API*, version 1.3.0 documentation; includes `step()`, `reset()`, and termination/truncation semantics. <https://gymnasium.farama.org/v1.3.0/api/env/>
- [4] S. Zhou, F. F. Xu, H. Zhu, et al. “WebArena: A Realistic Web Environment for Building Autonomous Agents.” arXiv:2307.13854, 2023; repository maintenance update 5 December 2024. <https://arxiv.org/abs/2307.13854> and <https://github.com/web-arena-x/webarena>
- [5] H. Trivedi, T. Khot, M. Hartmann, et al. “AppWorld: A Controllable World of Apps and People for Benchmarking Interactive Coding Agents.” *ACL 2024*, 16022–16076. <https://aclanthology.org/2024.acl-long.850/>
- [6] J. Lu, T. Holleis, Y. Zhang, et al. “ToolSandbox: A Stateful, Conversational, Interactive Evaluation Benchmark for LLM Tool Use Capabilities.” arXiv:2408.04682, 2024. <https://arxiv.org/abs/2408.04682>

- [7] S. Yao, N. Shinn, P. Razavi, and K. Narasimhan. “ τ -bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains.” arXiv:2406.12045, 2024. <https://arxiv.org/abs/2406.12045>
- [8] J. Pan, X. Wang, G. Neubig, et al. “Training Software Engineering Agents and Verifiers with SWE-Gym.” *ICML 2025*, PMLR 267:47717–47737. <https://proceedings.mlr.press/v267/pan25g.html>
- [9] Farama Foundation. *Gymnasium: Create a Custom Environment*, environment creation and reproducible seeding documentation (accessed October 2026). https://gymnasium.farama.org/introduction/create_custom_env/
- [10] M. Sullivan, M. Hartmann, and A. Koller. “Procedural Environment Generation for Tool-Use Agents.” *EMNLP 2025*, 18544–18562. <https://aclanthology.org/2025.emnlp-main.936/>
- [11] Deaimer Research. “Termetheus and the Foundations of Environment Based AI Learning.” Published 7 October 2026. <https://www.deaimer.com/research/termetheus-and-the-foundations-of-environment-based-ai-learn-wwqsr>