Echoverse: Deep, Evolving Environments for Training Computer-Use Agents at Scale

Yash Pandya **Sahil Gupta*** **Sarthak Harne** **Archana Yadav*** **Kavyansh Chourasia**
Hussein Mozannar **Vibhav Vineet** **Sara Abdali** **Corby Rosset**
Yash Lara **Ahmed Awadallah** **Ece Kamar** **Akshay Nambi†**
Microsoft Research

Abstract

Computer-use agents learn from what their actions change, so training one needs applications it can act on, break and reset. The ones that matter most are login-gated and stateful, so synthetic environments stand in for them. Recent pipelines generate such environments in bulk, moving the bottleneck from how many exist to what is inside each one. The returns come from three properties: how much behavioural depth an environment carries, whether it targets the interaction an agent actually fails, and whether it improves alongside the model. We present Echoverse, which compiles specifications into stateful applications whose tasks are graded against the application's own database, and a co-evolution loop that reads every graded rollout twice: as repairs to the environment, its tasks and its verifier, and as training signal. Trained on twelve such environments, a 9B model improves from 36.5% to 67.1% across fourteen evaluation splits, within fourteen points of the much larger frontier model that taught it. Taking the three properties in turn: on the same domains, shallow environments push live-site accuracy below the base model ($80.0 \rightarrow 75.0$) while deep ones raise it ($80.0 \rightarrow 85.0$ and $48.0 \rightarrow 65.0$); drilling one interface control across many renderings transfers to held-out widget families and to the open web; and repairing a single environment lifts the model trained on it from 16.2% to 38.5%. The same worlds serve as reinforcement-learning environments: a reward combining the grounded verifier with a dense per-step judge raises held-out score from 58.8% to 68.0%. We release four environments as a benchmark, with their applications, seed data and graders. Code: `https://aka.ms/echoverse`.

1 Introduction

A computer-use agent learns the results of its actions only where those actions have consequences. A click changes saved state, a message reaches a real person, or a page that refuses to move tells the agent that its last move did nothing. A screenshot shows what an interface looks like; only a running application shows what an action caused.

The consequences worth learning from are stateful, and most of them sit behind a login: email and chat, banking, health records, the internal consoles for cloud and machine learning. None of these can be trained against directly. Every attempt writes to a real account, there is no reset between tries, and the true state stays hidden behind the screen. We instead rebuild the application as a synthetic environment whose database we own, so that state changes for real but is safe to break, quick to reset, and gradable from the data rather than from a screenshot.

By a *world* we mean three things bound together: an **environment** (the application, its state, and the actions that change it), the **tasks** that set goals in it, and a **verifier** that grades the outcome against ground truth. Recent pipelines[Zhou, 2026, Zhang et al., 2026, Aggarwal et al., 2026, Wang et al., 2026] produce such worlds in bulk, yielding hundreds of environments and thousands of checkable

*Work done while at Microsoft Research.
†Corresponding author: `akshayn@microsoft.com`

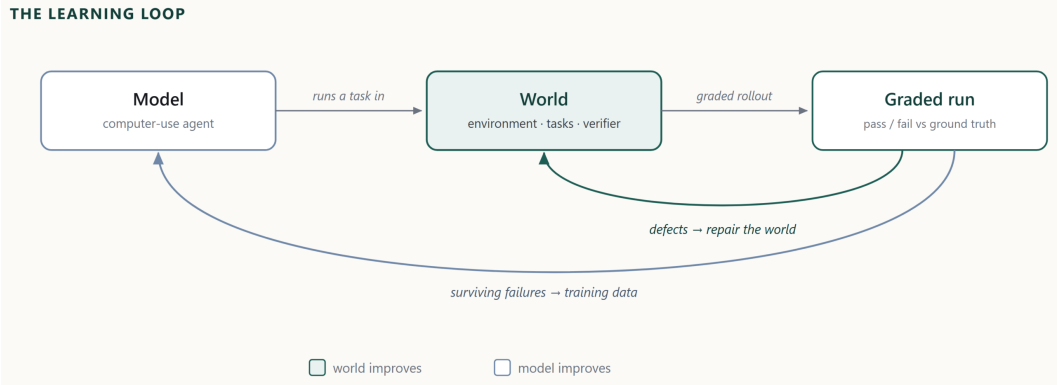

Figure 1: The learning loop. Every graded run is read twice. Failures that survive triage become model training data; defects in the environment, its tasks or its verifier become repairs. The same graded run that measures the model also sharpens the world.

tasks, and this work builds on that progress. Once worlds are plentiful, the bottleneck moves inside them, into whether each one holds together under real use, which is a question of depth rather than count: whether state stays coherent across users and screens, whether workflows keep their dependencies, whether a weak skill recurs in enough forms to generalise, and whether success is judged by outcome or by appearance.

Thesis. We argue that the gains come less from adding worlds than from a loop that keeps improving the ones already built. Echoverse treats building the environment and training the model as one process rather than two stages: run a model in a world, find where it fails, make the environment, its tasks and its verifiers more faithful or more demanding at that point, train on the sharper signal, and repeat. Ordinary fine-tuning improves only the model. Here the same graded run that measures the model also improves the world that judged it, so a static benchmark saturates while the loop compounds.

Three levers keep the loop productive, none of them raw environment count. **Depth**: worlds built backwards from the workflows they are meant to teach, so that every one of those workflows can actually be carried out in them, including in closed and proprietary domains. **Capability targeting**: narrow worlds built around the exact interaction a model keeps failing. **Co-evolution**: improving the environment, its tasks and its verifiers on every graded run, not just the model. Figure 1 shows how the two readings of a single graded run feed each other.

Reinforcement learning changes what is at stake. For supervised training, the argument above is one about data quality: a better world yields better demonstrations. Under reinforcement learning it becomes a precondition. Imitation carries a ceiling, because a clean demonstration never shows how to recover from a mistake or when to stop, and those are the failures that break agents in the wild. RL removes that ceiling but demands a volume of graded, resettable episodes the live web cannot supply: it will not reset, it throttles automated traffic far below the required rate, and it exposes no ground truth, so reward must come from a second model judging a screenshot. Every property that makes an Echoverse world good supervised data is also a prerequisite for RL, so the worlds serve as reinforcement-learning environments without modification. Sec. 7 develops the argument and reports a run over five worlds that lifts the held-out judged score from 58.8% to 68.0%. That run uses a composite reward, grounded at the trajectory level and model-judged per step, for reasons we set out in Sec. 7.3.

Contributions.

1. **An operational definition of depth** (Sec. 3): depth as completeness with respect to a target workflow set rather than as feature count, decomposed into five properties and discharged at build time against machine-checkable claims, so that "deep" is something a world is shown to be rather than something claimed about it.

2. **The Echoverse factory** (Sec. 4): a two-phase pipeline that compiles seeds into an application verified against machine-checkable claims, then grows a task corpus regrounded on live data and proven solvable through the real user interface, every task carrying a database-grounded verifier.
3. **The co-evolution loop** (Sec. 4.6): one graded rollout, read as two orthogonal signals, under an ordering in which the world is repaired before the model's failures are trusted, and under a discipline that never weakens a goal to raise a pass rate.
4. **Evidence for three levers** (Sec. 6), including the negative result that shallow worlds are worse than not training at all, and the finding that added environments keep helping after added trajectories have stopped.
5. **Echoverse as a reinforcement-learning environment** (Sec. 7): the same worlds, unmodified, meet the reset, throughput and reward requirements of RL, and a run over five of them trains a policy beyond its supervised initialisation.
6. **A public benchmark release** (Sec. 8): four worlds, the deep domains ECHOSTAY and ECHOFORGE and the two capability worlds with their held-out splits, each shipping as a runnable application with its seed database and, for every released task, the grounded verifier that scores it. We release evaluation tasks only, so the benchmark measures agents rather than doubling as training data for them. Environment code and graded tasks are at `https://aka.ms/echoverse`.

2 Related work

Live-web benchmarks. WebVoyager [He et al., 2024], Mind2Web [Deng et al., 2023] and Online-Mind2Web [Xue et al., 2025] evaluate agents against real public websites and remain the best available measure of open-web competence. Their coverage is bounded by what an unauthenticated visitor can do. No benchmark can hand out real credentials or let an agent write to a stranger's account, so the tasks come almost entirely from the logged-out surface of the web and are dominated by information finding: locate a page and read a value off it. The workflows that motivate computer-use agents sit on the other side of the login, where an action commits a booking or moves money and success is a change to stored state rather than a string the agent reports. Their verdicts are also model-generated. With no access to a site's backend, both judge success from screenshots and text with a multimodal LLM [Zheng et al., 2023], so a score is an opinion about appearance rather than a fact about application state, and the characteristic error is accepting an agent's confident claim to have completed an action that never landed. That is tolerable noise for a leaderboard read in aggregate, but it rules the signal out as a training reward (Sec. 7.2). These benchmarks also make poor training grounds because they do not hold still: pages are redesigned, listings and dates roll forward, and hosts throttle or block automated traffic. An occasional evaluation can absorb that drift; training cannot, since it runs the same task thousands of times and needs the same world each time. We use these benchmarks only as held-out transfer measurements (Sec. 6.5).

Synthetic environment suites. WebArena [Zhou et al., 2024], VisualWebArena [Koh et al., 2024], WorkArena [Drouin et al., 2024], OSWorld [Xie et al., 2024] and AndroidWorld [Rawles et al., 2025] established hand-built, resettable environments with programmatic checks, and BrowserGym [de Chezelles et al., 2025] consolidated several of them behind one harness. A more recent line automates their construction [Zhou, 2026, Zhang et al., 2026, Aggarwal et al., 2026, Wang et al., 2026], and UltraCUA [Yang et al., 2026] trains a foundation model over such data. These pipelines scale environment count. Our contribution is complementary: we hold count fixed and vary interior quality, and we show that below a depth threshold additional environments contribute noise rather than signal.

What makes a training environment good. The closest published statement of requirements to ours comes from practice rather than from a benchmark paper. Lin and Wang [2026] list five properties a web "gym" needs in order to drive learning: realism, explorability, data diversity and hydration, correct verifiers, and infrastructure stability. Three of those correspond directly to properties in Table 1, and the two lists differ in emphasis in a way worth naming. Theirs is written from the perspective of keeping a large RL system running, so it includes infrastructure stability, which we treat as engineering rather than as a property of a world. Ours is written from the perspective of what a single episode teaches, so it includes coherent cross-actor state and workflow dependency, which are exactly the properties a shallow but visually convincing clone can fail while still passing inspection.

We share their conclusion that task design dominates task quantity; Sec. 6.6 gives a measured version of it, in which environment diversity continues to help while trajectory volume saturates.

Verification. Tool-agent benchmarks such as τ-bench [Yao et al., 2024] grade against database state rather than text overlap, and we extend that tradition: our verdict of record is a database round-trip, so an answer copied from the public web is rejected because it does not reproduce. This matters more for training than for evaluation. A tolerant LLM judge [Zheng et al., 2023] used as a training filter teaches the policy the judge's blind spots, and under RL the same failure is sharper still, since a verifier that rewards work the agent did not do is a reward the optimiser will find and exploit [Lin and Wang, 2026]. A model still makes the final comparison in our loop, since equivalence is not string identity, but it compares an observed outcome against a reference read out of the database rather than assessing a trajectory (Sec. 3.2), which is a much narrower question than the one such judges usually fail. Sec. 4.6 treats the verifier as an object of repair for this reason, alongside the environment and the task it scores. Where ground truth is not available, careful judge design remains the alternative; Rosset et al. [2026] show that separating process and outcome criteria and attending to the whole trajectory drives judge false-positive rates close to zero. Our two-term reward in Sec. 7.3 follows the same process-plus-outcome split, with the outcome term grounded rather than judged.

Relation to our own prior work. Fara-7B [Awadallah et al., 2025] and Fara-1.5 [Awadallah et al., 2026] are our earlier computer-use agents, trained on data from a generation pipeline that draws on both live websites and synthetic environments; the synthetic component of that pipeline used an early version of the worlds described here. Those papers report end-to-end models built from a mixed pipeline. This paper takes the environments themselves as the object of study and asks which of their properties make a world worth training on, which is why the models reported here are trained on synthetic data alone and are not comparable to the Fara releases.

Curriculum and task curation. Which tasks to spend RL budget on is its own design problem, because tasks the policy already solves produce no gradient and tasks it never solves produce noise. Lin and Wang [2026] track per-task success rates and concentrate sampling in a mid-range band of roughly 30 to 70 percent. We apply the same idea as a one-off filter rather than as a running curriculum: we sample the starting policy four times per task and keep only the tasks it solves once, twice or three times, discarding both the ones it never solves and the ones it always solves (Sec. 7.4).

Automatic curriculum and environment design. Co-evolution is adjacent to open-ended and unsupervised environment design [Wang et al., 2019, Dennis et al., 2020, Parker-Holder et al., 2022, Portelas et al., 2020] and to self-play [Silver et al., 2018]. The difference is what is evolved. Those methods search over a parameterised task distribution inside a fixed, correct simulator. We co-evolve the implementation of the environment, its task corpus and its verifier, because in generated software all three are defective at the start and a defect in any one silently poisons measurement of the others.

Distillation and reinforcement learning. Our supervised stage is verifier-filtered rejection-sampled distillation, following STaR [Zelikman et al., 2022] and rejection fine-tuning [Yuan et al., 2023], and inherits the corresponding teacher ceiling. Our RL stage uses a group-relative policy gradient [Shao et al., 2024, DeepSeek-AI, 2025] with the unnormalised advantage of Liu et al. [2025], which suits sparse, verifier-supplied reward over long browser trajectories. The practical obstacles in this regime, namely the train-inference gap, credit assignment over hundreds of steps, and rollout throughput, are documented in Lin and Wang [2026]. What distinguishes our setting is not the algorithm but the environment requirement (Sec. 7.2).

3 Problem setting: environments, tasks and grounded grading

3.1 Environments, tasks and worlds

An *environment* is a tuple $E = (\mathcal{S}, \mathcal{A}, T, \mathcal{O}, D_0)$. The state space $\mathcal{S}$ is the application's database, not its rendering; $\mathcal{A}$ is a browser action space; T is the application's own transition function, realised by its backend; $\mathcal{O}$ maps state to what the agent sees; and D_0 is the seeded initial database. Taking the state to be the database rather than the pixels is what makes reward groundable and reset exact, and the rest of the paper follows from it.

A *task* is a tuple $\tau = (g, r, \kappa)$, where g is a natural-language goal, $\kappa \in \{\text{READ}, \text{WRITE}, \text{READ\_WRITE}\}$ is the task kind, and r is the reference the outcome is checked against: a *reference answer* for a READ task, a *reference state change* for a WRITE task, and both for a READ_WRITE task. References are minted from D_0 at generation time by executing SQL against the live database, so a task is true by construction rather than by assertion.

A *world* $W = (E, \mathcal{T}, V)$ is an environment together with a task corpus $\mathcal{T}$ and the verifier V that grades outcomes in it, as introduced in Sec. 1. The distinction matters throughout: the environment is the application, while the world is the trainable unit, and the co-evolution loop of Sec. 4.6 repairs all three components.

3.2 Grounded grading

Let D_T denote the database after the agent terminates. Grading is defined per task kind:

- WRITE is graded against the reference state change: it must be present in the `sqldiff` between the pristine per-task copy D_0 and D_T, which establishes both that the write landed and that it is the write the task asked for, and stops a task passing on state that was already there.
- READ is graded against the reference answer, which was itself read back from the database at generation time. The agent's response must match it under semantic equivalence ($288 for $287.62 passes).
- READ_WRITE carries both references and scores the minimum of the two.

How the comparison is made. Neither check is a string match. An answer of $288 should count against a reference of $287.62, and a booking written with the right listing, dates and guest count should count whether or not incidental fields differ. We therefore make the final comparison with an LLM under fixed guidelines, reproduced in full in Appendix C, and it is worth being precise about what that judge does and does not see. Its input is two items: the agent's reported answer, or the `sqldiff` between D_0 and D_T for a write, together with the corresponding reference. It does not receive the trajectory, the screenshots or the agent's own account of what it did, and it is never asked whether the agent behaved sensibly, only whether the observed outcome matches the reference it is given. Changes beyond those the reference requires are tolerated, so an agent that also stars an email while marking it read is not penalised for it. The call is therefore small and cheap enough to run on every rollout of a training run.

Reliability follows from that narrowness rather than from the judge's strength. Deciding whether two amounts agree, or whether an observed row diff carries the same content as a reference one, is a closed comparison against a value that was itself read out of the database. That is a far easier question than the one a live-web judge must answer, which is whether a fifty-step episode succeeded given only screenshots and the agent's own summary, with no ground truth to compare against. The grounding lives in where the reference comes from and in how little the judge is asked to decide.

Grading is therefore hard to game, grounded rather than labelled, and uniform across an ECHOSTAY booking, an ECHOFORGE issue and an ECHOBANK transfer. It eliminates the two dominant false-positive modes we see in practice: the agent claiming a success it did not achieve, and a tolerant judge accepting an answer retrieved from the public web.

3.3 Depth, defined operationally

An environment can be stable, plausibly styled and still be useless to train on. What separates a world worth training on from one that is not is depth, and depth is not an absolute quantity. A world is deep when its environment supports every workflow its task corpus demands, end to end through the interface, and each leaves an effect in state precise enough to check.

Completeness is therefore relative rather than exhaustive. We do not try to reproduce every feature of a banking product, only every action its transfers, bill payments and disputes actually need, with the balances, permissions and histories those actions touch. An environment missing a feature no target workflow exercises is not shallow; an environment whose booking flow cannot set a guest count is shallow however many pages it renders.

This is what makes the definition operational. The factory takes the workflows as an input rather than discovering them afterwards: scenarios, seed data, diagnosed capability gaps and the intended tasks all enter at the start (Sec. 4), are compiled into machine-checkable claims about the routes and state they require, and the database, backend and frontend are repaired until those claims pass and the tasks are completable by an agent driving the real interface. "Deep" is then a statement about what a world has been shown to support rather than a property asserted for it.

Table 1: What supporting a target workflow set requires. The first four are structural obligations on the world; the fifth decides which workflows are worth imposing them for.

Property	What it requires
Behavioural fidelity	Controls, permissions and errors follow the product's logic
Coherent state	A sent message appears for its recipient; a cancelled meeting clears both calendars
Workflow depth	An early choice constrains what happens later
Authoritative verification	Success is a property of application state, not of pixels
Domain value	The workflow is worth improving

Table 1 decomposes that obligation. A workflow cannot be driven if controls, permissions and errors do not follow the product's logic; it cannot span actors if a sent message never reaches its recipient; it is not a workflow at all if an early choice constrains nothing later; and it cannot be graded if success is visible only in pixels. The fifth property is a choice rather than a requirement, and decides where the other four are worth paying for.

In the systems that matter most, the difficulty lives in permissions, shared state and audit histories, which is exactly the structure a shallow clone skips. Above this bar, more environments add variety; below it, they add noise, and Sec. 6.2 measures that difference directly. Depth also requires the world to be fixed in time and data: the calendar does not move, the seed data does not churn, and a task means the same thing on the thousandth rollout as on the first. We trade a little surface realism for that control, and in Sec. 7 the trade stops being a convenience and becomes the precondition for training at all.

4 Echoverse: the factory and the co-evolution loop

Echoverse is a single pipeline that produces two distinct types of outputs: full-domain worlds that preserve the depth and structure of realistic workflows, and capability worlds that isolate and vary a specific diagnosed interaction or capability. Both rely on our owning the database underneath, so that success is a property of the application's own state rather than of a model's reading of a screenshot. Figure 2 shows the pipeline end to end, organised into two phases.

In Phase 1 we build and validate the world itself. The process begins with hand-written seeds describing representative scenarios, tasks, capabilities and features. These seeds are expanded into a structured specification and a set of explicit, verifiable claims, which are then used to generate the complete application, including its frontend, backend and database. Generation is followed by an iterative verification-and-repair cycle: claims are checked against the generated application, failures are localised and fixed at the appropriate layer, and the application is re-verified, until the specification is satisfied and a validated environment remains.

In Phase 2 we grow a grounded task corpus on top of that environment. The original seeds are regrounded against the live database to generate tasks whose expected outcomes can be determined directly from application state. Each task, together with the corresponding state of the environment, is then evaluated for quality and solvability. Failures are diagnosed and repaired, and the tasks are re-executed and re-scored against database-derived ground truth, until enough tasks clear every verification criterion for the corpus to be exported.

The two phases run the same cycle, and are better read as one loop than as two stages. Sec. 4.6 makes that loop the object of study, because it does not stop when a world ships: the failures of the model being trained on a world are themselves evidence about the world.

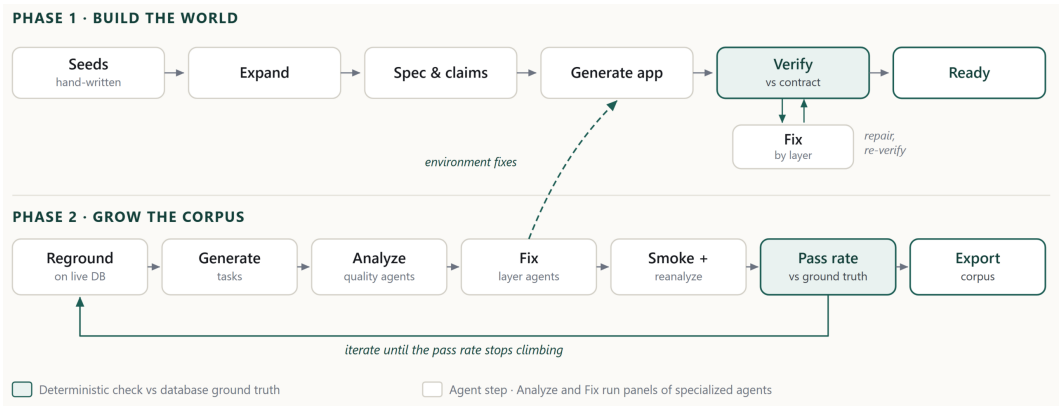

Figure 2: The environment factory. Phase 1 expands a handful of seeds into an application, then repairs the database, backend and frontend until it passes machine-checkable claims. Phase 2 regrounds tasks on live data, runs a panel of verifier, triage and layer-specific fixer agents, and re-scores against database ground truth until the corpus clears its verification bar. Many of those fixes land in the environment itself (dashed arrow).

4.1 An agent workforce

Every stage in Figure 2 is carried out by an agent rather than by a fixed script. The pipeline is implemented as a set of GitHub Copilot SDK agents, each given a role, the repository of the world under construction, and the tools an engineer would use in that role: a terminal for running the stack, its migrations and its tests; Playwright [Microsoft, 2024] for driving and inspecting the running interface as a user meets it; and database tools exposed over the Model Context Protocol [**?**] for querying and editing the application's own state. The roles divide into four kinds.

- **Builders** produce the artefacts: schema and seed data, backend routes, the frontend that exercises them, and the task corpus.
- **Verifiers** test what the builders produced, against the specification in Phase 1 and against quality and solvability in Phase 2.
- **Triagers** read the verifiers' findings across the whole world at once, group failures that appear to share a cause, and turn them into a smaller set of consolidated work items, each attributed to the layer that must change.
- **Fixers** take those items, debug against the running system, and re-check the repair, rolling it back if it regresses something else.

Two features of this arrangement do the work. Verification is separated from construction, so no agent is the sole judge of its own output. And triage sits between verification and repair, so fixes are planned over the whole set of observed failures rather than one task at a time, which is what lets a single repair clear many tasks (Sec. 4.7). Human supervision and QA sit over the whole process as an additional layer of oversight on agent-driven generation, verification and repair.

4.2 Phase 1: building the environment

The pipeline expands a handful of seed scenarios into a specification, then compiles it into machine-checkable claims about routes, state and behaviour. Only then is the application generated: a FastAPI and SQLite backend under a React interface. Verifier agents exercise every claim against the running environment, driving the interface and reading the database behind it, and fixers repair the database, backend or frontend until the claim holds. We advance a world when at least 95% of its claims pass; the remainder is written into a *readiness record* that separates hard blockers from advisory risks, and an environment with an open blocker does not advance at all. This is the first half of the mechanism behind Sec. 3.3: the claims are derived from the workflows the world is meant to carry, so passing them is evidence that those workflows are supported rather than merely rendered. Phase 2 supplies the second half by requiring that the tasks themselves can be completed through the interface.

4.3 Phase 2: growing the corpus

An environment that builds cleanly is still not training data. From the seeds, we create tasks grounded on the live database, drawing goals from entities that actually exist, then send every goal through a panel of verifiers. These check whether its entities are real, whether the goal is plausible, whether its difficulty matches the work, and, most consequentially, whether the goal is feasible at all.

That last check is white-box, and deliberately so. The verifier agent drives the running interface with Playwright, but it also reads the application's source and queries its database directly, so it can separate a goal no interface can satisfy from one that is merely hard. Feasibility is therefore a property of the application, established by a factory agent, rather than a record of what some particular policy happened to solve: no model that we later train or evaluate takes part in deciding which tasks exist. This matters for how the evaluation splits should be read, since a corpus filtered by a frontier model's success would be a corpus shaped to that model's strengths.

Every failure becomes an issue tagged by the layer that must change: database, backend, frontend, task text, or verifier. Triagers consolidate the issues that share a cause, and layer-specific fixers apply the repair, re-check it, and roll it back if it regresses. The loop re-scores against database ground truth until at least 95% of the surviving corpus passes verification, and each exported task carries the exact check that grades it.

Fixing or improving the environment and growing the corpus are one loop rather than two stages, and together they produce the world. In our experience most defects belong to the environment rather than to the task text, so we re-version it on every iteration. Harder tasks expose gaps in the environment, and a sturdier environment can carry harder tasks.

4.4 Seeding

Where a rich public dataset exists we build on it. ECHOSTAY is seeded from InsideAirbnb [Cox, 2024], a public dataset compiled from publicly visible short-term-rental listings, so its listings, hosts, reviews and amenities are real rather than invented, and ECHOFORUM sits on a public forum corpus of 2.55 million comments. Where none exists, as with mail, calendar, banking and health records, a seeding pipeline generates the state under strict constraints: dense and internally consistent, not a handful of placeholder rows. Table 3 gives per-world statistics.

4.5 Determinism, isolation and reset

Every rollout runs against an isolated per-task copy of the database, so no task can contaminate another and the environment side of any run can be reproduced exactly. Whenever seed data changes, the grounding database is regenerated, re-validated and re-published, so corpus and shipped artefact never drift. Sec. 7 reuses these reproducibility properties unchanged as the reset primitive of a reinforcement-learning environment.

4.6 One rollout, two signals

The score an agent earns is never a measure of the model alone. It reflects a coupled stack comprising the agent, the environment, the task, and the verifier. A zero can therefore have many causes: the agent may have failed, but the environment may also be broken, the requested state may be impossible to reach, or the verifier may be checking the wrong condition. Treating every failure as an agent capability gap turns defects in the synthetic world into erroneous supervision. That limits what the agent can learn, and risks teaching it to avoid otherwise valid features and interaction patterns simply because they were broken in its training environments. We therefore treat every graded rollout as a test of the entire stack. Failures attributable to the environment, task, or verifier identify defects that should be repaired rather than learned from. What remains, failures on tasks that have been established to be valid, solvable, and correctly verified, is a cleaner signal of the agent's actual capability gaps and therefore of the hard cases it should train on next.

Attribution needs no separate apparatus. Failing rollouts go back through the same verifier and triage agents the factory already runs, which ask of a rollout what they ask of a candidate task: whether the goal is reachable through the interface, whether the state it names exists, and whether the check that graded it was the right one. The distinction we act on is coarse, between failures that implicate

the world and failures that implicate the model, and where the reading is unclear we prefer to repair, since a task withheld from training costs less than a task that teaches the wrong lesson.

4.7 Repair before training, and why one fix clears many tasks

Repairing the world before training is what makes the resulting model signal trustworthy. A failure caused by a broken environment is noise, not a useful learning target. The principle is therefore not simply that generated benchmarks contain bugs, but that these bugs must be identified and repaired before agent failures are used as supervision. This ordering also imposes an important discipline on the repair loop: goals are never weakened merely to increase pass rates. Fixers may change the environment, the task text or the verifier, but a repair that leaves a task asking for less than it did is treated as a regression rather than a fix, and the bar a corpus must clear is the one stated in Sec. 4.3 rather than one relaxed to fit what the model can already do. If a task is valid, solvable, and correctly verified, then a model failure is precisely the signal we want to preserve.

Repair also has a multiplicative effect because failures that appear independent at the task level often share a common underlying cause. Triage groups failures by that cause and prioritises fixes according to how many tasks they affect. This makes central environment defects particularly valuable repair targets: a single broken control or interaction can block a large share of a corpus, and restoring it restores all of those tasks at once. Sec. 6.7 reports several such rounds, and what one of them does to the model trained on the repaired world.

Many of these defects are easy to miss during ordinary human QA because they are specific to how automated agents interact with the application. A human user, for example, may set a guest count without giving the control a second thought. If an agent cannot reliably operate that same control, however, it silently leaves the default value unchanged, causing every booking task that depends on guest count to fail. Agent-driven execution therefore exposes a class of environment defects that may look innocuous to humans but systematically distort the training signal for agents.

4.8 From tasks to trajectories

Only once a world has been through that loop do its tasks become training data, and they do so through a single process. A frontier model (GPT-5.4) solves each task in the live environment, the grounded verifier of Sec. 3.2 keeps only the trajectories that pass, and those become the supervised fine-tuning corpus behind every experiment in Sec. 6. The selection here is over trajectories, not over tasks: which tasks exist was settled in Phase 2 by the feasibility check of Sec. 4.3, and a task the teacher fails still stands, it simply contributes no demonstration. Appendix B gives the per-world composition of the resulting corpus. This is verifier-filtered, rejection-sampled distillation [Zelikman et al., 2022, Yuan et al., 2023], and it inherits the teacher's ceiling. Sec. 7 addresses that ceiling.

5 The environment suite

5.1 Ten deep domains

The domains with the most consequential work are the hardest for public benchmarks to reach: closed, proprietary systems where the difficulty lives in permissions, shared state and history rather than layout. What matters is not the pixels but that an action's consequences reach across screens and users, so that a task can run a real workflow and be graded on the state it leaves behind. Table 2 groups the ten domains by the work they represent, and Figure 3 shows the suite as an agent meets it.

That accumulated state is what carries consequences across screens and users. A booking in ECHOSTAY moves through search, listing, availability and payment across roughly 87 routes and 23 tables, with no confirmation screen to short-circuit it. An ECHOMAIL thread carries intent from draft through delivery, reply and label state. An ECHOCARE order writes each change to an audit trail. The tasks are expensive because of it, often five to thirty actions deep, and finished only when the underlying state has changed.

Table 2: The ten full-domain worlds, grouped by the work they represent. Each is named for the workflow rather than for any product.

Workflow category	Worlds	Depth the world must carry
Communication & coordination	EchoMail, EchoCalendar, EchoChat	Shared threads, schedules, participants, permissions, histories
Technical creation & operations	EchoML, EchoForge	Artefacts, configuration, dependencies, roles, multi-stage changes
Regulated records & transactions	EchoBank, EchoCare	Balances or records, authorisation, audit history, consequential writes
Community, media & travel	EchoForum, EchoTunes, EchoStay	Persistent preferences, social state, search, booking, account actions

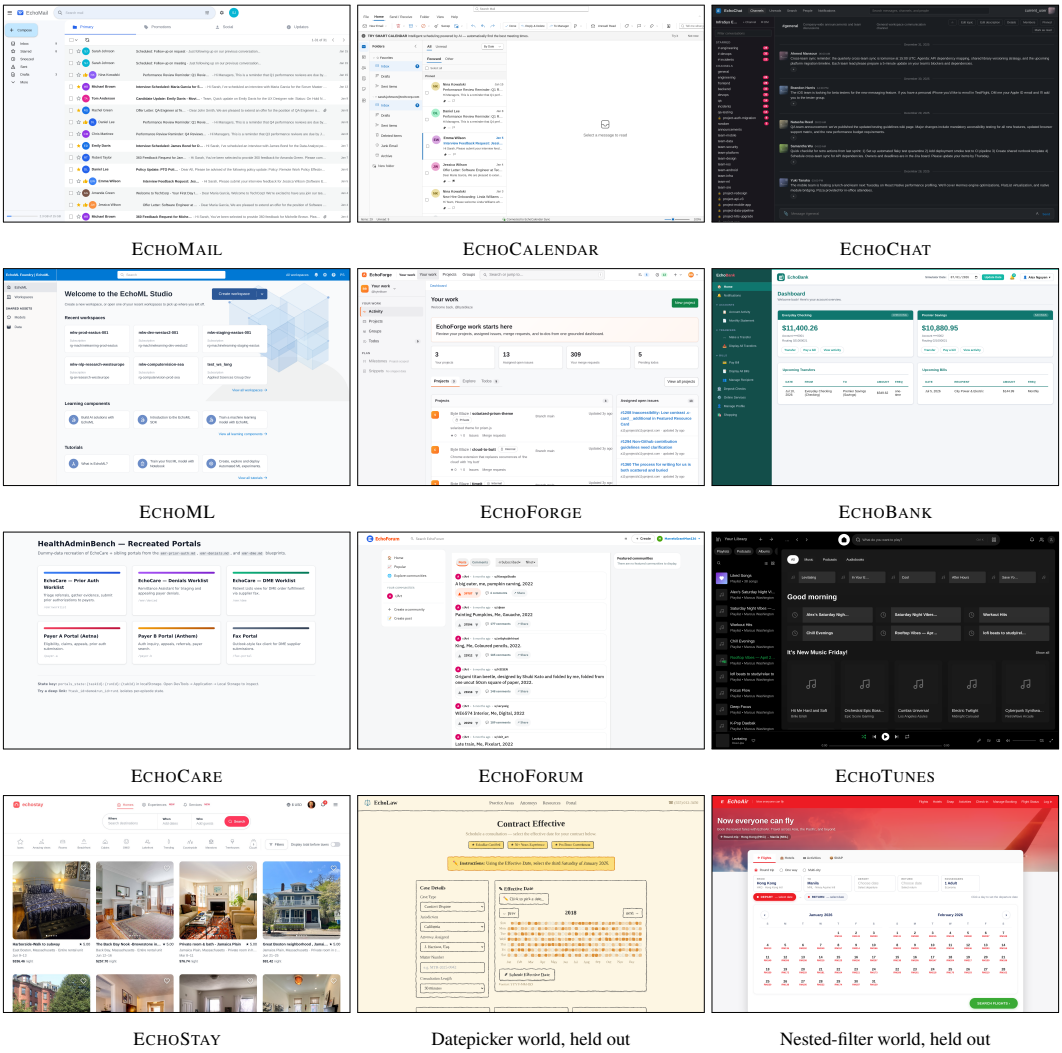

Figure 3: The suite as an agent meets it. The first ten panels are the full-domain worlds tabulated above, each a behaviourally faithful clone of a class of application rather than of any particular product. The last two are held-out renderings from the capability worlds of Sec. 5.2: a calendar heatmap in a legal-services frontend and a compound booking-search panel in an airline frontend, neither of which appears in training. What a task depends on is the state behind these screens rather than their appearance.

Table 3: Grounded database state for each of the ten full-domain worlds. All counts are measured from the shipped seed database, not from a specification.

World	Class of application	Tables	Rows	Principal seeded entities
ECHOMAIL	Email workspace	13	5.2K	1.1K emails, 1.1K threads, 170 labels, 374 contacts
ECHOCALENDAR	Calendar and mail	13	2.2K	196 events, 102 folders, 374 contacts
ECHOCHAT	Team messaging	15	34.7K	188 channels, 1.3K workspaces, 3.2K users
ECHOML	ML platform	11	3.3K	173 models, 444 datasets, 152 endpoints, 309 compute resources
ECHOFORGE	Developer collaboration	15	705K	175 projects, 81K issues, 134K merge requests
ECHOBANK	Online banking	10	1.8K	100 accounts, 4.1K transactions, 201 transfers, 672 bills
ECHOCARE	Health revenue cycle	14	2.8K	250 prior authorisations, 250 denials, 200 orders, 1.3K documents
ECHOFORUM	Social forums	11	4.0M	95 forums, 127K posts, 2.55M comments, 662K users
ECHOTUNES	Music streaming	42	21.5K	2.5K songs, 1.1K albums, 863 artists, 544 playlists
ECHOSTAY	Lodging marketplace	23	180K	3.7K listings, 5.4K bookings, 5.3K reviews, 6.1K users

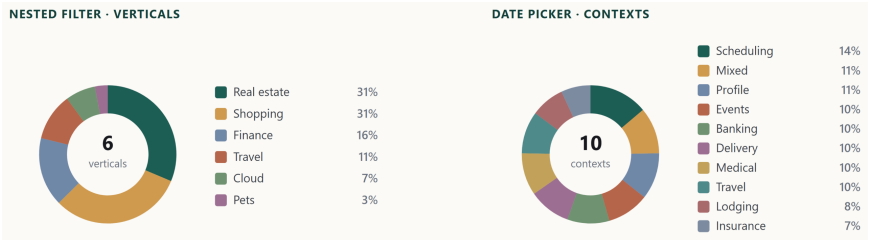

Figure 4: Thematic spread of the two capability worlds. The same control is re-themed so that the skill cannot be learned as a single layout: nested filters over six verticals, date pickers over ten contexts. Percentages are shares of generated frontends within each world.

5.2 Two capability worlds

Not every weakness is a missing domain. Some are a single control the agent cannot reliably operate. Consider an agent that searches, filters and opens the right listing, then stalls at the date picker, unable to turn "the second week of March" into the right clicks on an unfamiliar calendar. Building another booking site would not fix that. The skill is learned only when the control itself appears in enough forms, and date pickers and nested filter-and-search controls are rendered a hundred different ways on the live web, which is variability a single deep application cannot supply.

We therefore isolate the control and widen the interaction, mass-producing it across layouts, states and constraints, then generating grounded tasks over each. The datepicker world renders one date control as six core widgets across ten contexts, 100 frontends in six visual styles, and holds out ten unseen widgets over 36 further scenarios, 80 frontends in nine styles, from calendar heatmaps to scroll wheels and fiscal-quarter pickers. Its hardest tasks turn transcription into reasoning, resolving "the last Thursday of January 2026" or "10 business days after a start date" to one exact, widget-reachable date. The nested-filter world varies twenty widget families over 200 frontends in fifteen styles, and holds out nine compound panel families over 100 frontends in nine styles, grading every submission by whether the filtered results actually meet the requested conditions, judged by the application's own logic rather than by appearance. Neither world lets a single theme dominate, since a control learned only against one vertical is a layout rather than a skill; Figure 4 gives the thematic spread of both, and Appendix D lists every family with the interaction it demands.

5.3 Twelve training worlds, fourteen evaluation splits

To avoid ambiguity in the tables that follow: training uses **twelve worlds**, the ten full domains plus the two capability worlds, while evaluation reports **fourteen splits**, the ten domains plus an in-distribution and a held-out split for each capability world. All averages in this paper are unweighted means over the fourteen splits.

6 Supervised experiments

6.1 Setup

Three models run through this section: **Base** (π_{base}), Qwen3.5-9B [Qwen Team, 2026] given only enough synthetic trajectories to align it to the browser action space; **Ours** (π_{SFT}), that same 9-billion-parameter network trained on the full corpus of Sec. 4.8, 21,009 verified trajectories over the twelve worlds (Appendix B); and **GPT-5.4**, the far larger frontier model that generated it.

Policies are served with vLLM [Kwon et al., 2023]. Every rollout is capped at 100 actions, and a task that reaches the cap without emitting `terminate` is recorded as over-budget. Tasks in the synthetic worlds are graded by the database-grounded verifiers of Sec. 3.2, while WebVoyager and Online-Mind2Web use their standard published judges. Live-web runs are reported both through Browserbase [Browserbase, 2025], which strips the datacentre bot-blocks and rate limits that depress every agent's score, and without it.

6.2 Deep worlds transfer; shallow worlds set the model back

A shallow world is the cheap option: it stands up fast and looks convincing, but it rehearses only isolated, correct-looking clicks, so a model trained on it picks up habits the easy world never punished, over-stepping, looping and repeating dead actions. A deep world costs more, but its trajectories carry the dependent structure that transfers.

To isolate this we take two live WebVoyager domains, Allrecipes and Hugging Face, and compare three checkpoints: π_{base} and two ablation models, π_{shallow} and π_{deep}, trained on shallow-world and deep-world trajectories built for those two domains alone, so neither is π_{SFT}. The shallow world poses short, self-contained tasks; the deep world poses tasks that run across dependent steps, where an early action changes the state, options and verification available later. Both see the same two domains and corpora of comparable size, though not matched trajectory for trajectory. Evaluation uses public WebVoyager tasks for these domains, run on the live sites outside any training world.

On Allrecipes π_{shallow} falls below base, $80.0 \rightarrow 75.0$; on Hugging Face it stays flat at 48.0. Only π_{deep} improves both, lifting Allrecipes to 85.0 and the harder Hugging Face split to 65.0 (Figure 5). Since both runs cover the same domains at comparable scale, depth is the systematic difference between them: what separates improvement from regression is not how much the model saw but whether what it saw preserved the structure of the work. The mechanism shows in the trajectories: π_{deep} loops less, and of the 37 Hugging Face tasks those that exhaust their step budget fall from 15 to 9. With two domains and a few dozen tasks each, we read this as evidence that a shallow world can be worse than no training at all, not as an estimate of how much depth is worth.

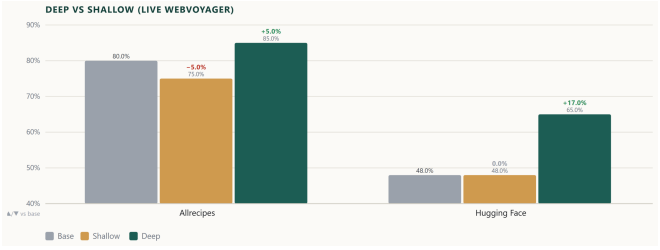

Figure 5: Deep versus shallow worlds on two live WebVoyager domains, with the same domain coverage and differing task depth. Deep lifts both; shallow drops below base on Allrecipes and stalls on Hugging Face.

6.3 The full scorecard: base, our model and a frontier reference

This is the main supervised result: what the twelve worlds, through the 21,009-trajectory corpus of Appendix B, do to a single 9B model. Three policies run the same fourteen splits under the same harness, the base model π_{base}, our supervised model π_{SFT} trained on that corpus, and GPT-5.4 as frontier reference and as the teacher that produced it. Every entry is task success under the database-grounded verifiers of Sec. 3.2, not a judgement about appearance. Table 4 lists the splits; Figure 6 plots the same numbers as distance still to travel. Across all fourteen splits π_{SFT} nearly doubles base, $36.5 \rightarrow 67.1$, and on the six splits where base was weakest it climbs three- to nine-fold, from single or low double digits into the forties through sixties. That leaves a 9B model within fourteen points of GPT-5.4 on the average, matching or beating it on ECHOMAIL, ECHOBANK and both nested-filter splits, on training data that is deep, targeted and checkable rather than on scale.

Table 4: Task success (%) over all fourteen evaluation splits, ordered by π_{SFT}. Bold marks the splits where the 9B model matches or beats GPT-5.4.

Split	Base π_{base}	Ours π_{SFT}	GPT-5.4
ECHOBANK	76.6	**94.6**	92.8
Datepicker (in-distribution)	60.0	87.2	89.8
Nested filter (in-distribution)	76.0	**87.0**	87.0
Nested filter (held out)	62.8	**84.1**	82.8
ECHOMAIL	64.2	**81.1**	74.5
ECHOTUNES	35.9	70.6	81.7
ECHOCALENDAR	13.6	66.9	78.1
ECHOFORUM	20.4	62.0	88.7
ECHOCARE	15.2	61.6	78.4
ECHOML	6.5	56.0	80.6
ECHOFORGE	16.8	52.5	74.3
Datepicker (held out)	34.0	52.0	94.7
ECHOCHAT	8.6	46.7	76.2
ECHOSTAY	20.5	37.6	50.4
Average	**36.5**	**67.1**	**80.7**

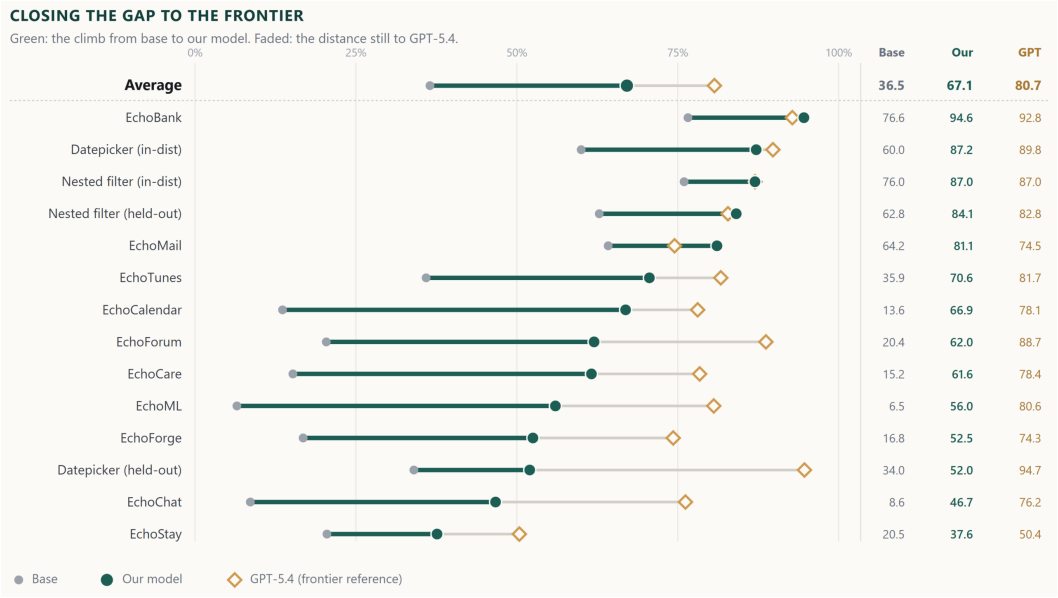

Figure 6: Closing the gap to the frontier, per evaluation split. The solid bar is the climb from π_{base} to π_{SFT}; the faded remainder is the distance still to GPT-5.4.

6.4 Precision about one skill

The datepicker and nested-filter worlds drill the controls our evaluations flagged, and the two skills reinforce each other rather than competing. The four checkpoints below are capability-only ablations, trained on those two worlds rather than on the full corpus, so none of them is π_{SFT} and their scores are not the ones Table 4 reports.

Two findings matter, both visible in Table 5 and summarised in Figure 7. First, gains hold on forms never trained on, with held-out date pickers rising $34.0 \rightarrow 57.3$ and held-out filter panels $62.8 \rightarrow 84.8$, which indicates these models learned a rule rather than a layout. Second, the skills transfer across each other rather than cannibalising: training either alone still lifts the other (datepicker training lifts held-out filters to 82.1, filter training lifts held-out date pickers to 50.7), and training both is the best all-rounder on every split. Against GPT-5.4 as a frontier reference, the +Both model edges ahead on nested filters and closes most of the datepicker in-distribution gap, trailing clearly only on held-out date pickers, where GPT-5.4 is exceptionally strong (94.7). The effect also reaches the open web: +Both lifts Online-Mind2Web from 29.5 to 34.3 on sites it never saw.

Table 5: Capability ablation: four models over four splits (task success, %). Best per row in bold. "Held out" widget families never appear in training.

Split	Base	+Datepicker	+Nested filter	+Both
Datepicker, in-distribution	60.0	82.6	73.1	**83.5**
Datepicker, held out	34.0	54.0	50.7	**57.3**
Nested filter, in-distribution	76.0	78.0	84.0	**88.0**
Nested filter, held out	62.8	82.1	84.1	**84.8**
Mean	58.2	74.2	73.0	**78.4**

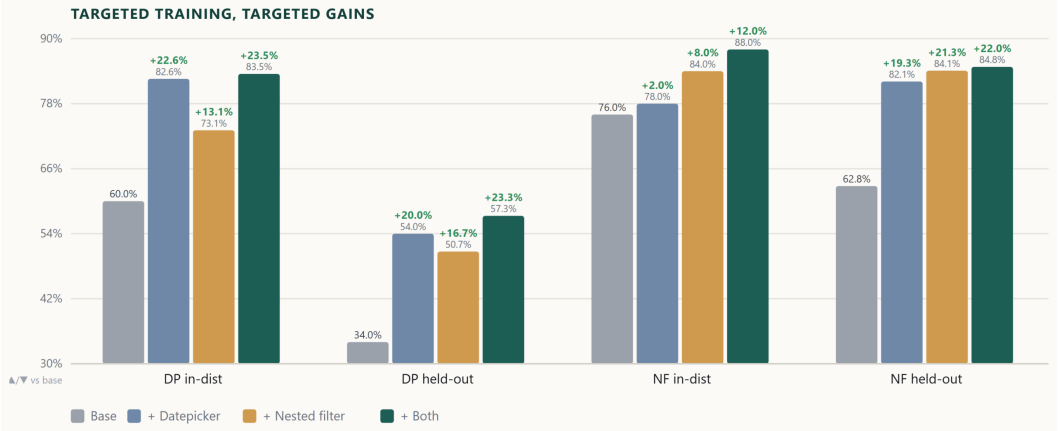

Figure 7: Targeted training, targeted gains. Training either skill lifts both controls, including held-out widgets and compositions neither was trained on, and training both is the best all-rounder on every split.

6.5 Transfer to the live web

To test whether the skill survives the open web, we evaluate π_{SFT}, unchanged, on WebVoyager and Online-Mind2Web, benchmarks it never trained on. These barely overlap with what we built, since both are dominated by open, public sites and read-mostly browsing while our worlds train login-gated, write-heavy workflows. We therefore expect direction rather than a large jump. Table 6 reports both browser conditions.

With no live-web data in the mix, this is transfer rather than memorisation. The modest size of the gain reflects coverage rather than a ceiling, and aiming at a live domain grows it. ECHOFORGE, our code-hosting world, is the same kind of application as GitHub, one of the live sites WebVoyager tests, and adding it to an earlier, smaller training mix lifted that mix's live GitHub score from 58.5 to 63.4, with its overall live scores rising too (WebVoyager $50.9 \rightarrow 52.9$, Online-Mind2Web $29.5 \rightarrow 31.1$);

Table 6: Live-web transfer (task success, %). Neither benchmark contributes any training data. Hosted rows run through a browser service that removes datacentre bot-blocks; datacentre rows do not.

Benchmark	Browser	Base π_{base}	Ours π_{SFT}
WebVoyager	hosted	66.5	71.5
WebVoyager	datacentre	50.9	55.6
Online-Mind2Web	hosted	40.5	43.4
Online-Mind2Web	datacentre	29.5	37.2

those figures are that ablation's, not π_{SFT}'s. The average reflects that most of what we built sits in domains these benchmarks never touch.

6.6 What scaling buys, and what it does not

We scaled two axes separately: more trajectories through a fixed set of environments, subsampled from the 21,009-trajectory corpus of Appendix B at its own per-world proportions, and more distinct environments. They behave differently, as Figure 8 shows.

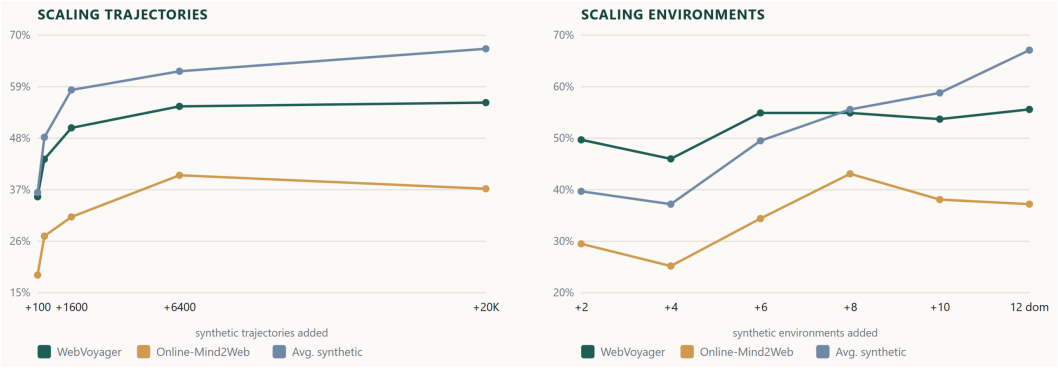

Figure 8: Two scaling axes, scored without a hosted browser. Left: more trajectories over a fixed set of worlds, subsampled from the full corpus at its own per-world proportions, with the horizontal axis spaced by actual trajectory count. The synthetic average keeps rising, but live-web transfer saturates. Right: more environments, where breadth keeps both the synthetic average and WebVoyager climbing.

More trajectories on the same worlds keep lifting the in-domain average, though the gains shrink, while transfer to the live web flattens outright: from 6 400 to 20 000 trajectories, WebVoyager holds steady ($54.8 \rightarrow 55.6$) and Online-Mind2Web slips ($40.1 \rightarrow 37.2$). Since every point holds the per-world mixture fixed, this is not an artefact of composition. Each environment holds only so much transferable skill, and once a model has drawn it out, more rollouts polish what it already does.

Scaling environments produces the opposite result: the average keeps climbing as breadth grows, and WebVoyager reaches its best only with the full set. For generalisation the lever is diversity rather than volume. Scale by itself is not the lever on either axis, since a large trajectory budget spent on shallow worlds, or graded against the wrong answer, moves the synthetic number and goes nowhere on the live web.

6.7 The model is not the only thing that learns

ECHOSTAY made the co-evolution loop visible. Its failures traced to the world rather than the agent: a guest-count control silently broke booking tasks, so a correct booking could never register. Fixing it raised the share of those bookings that could be completed at all from 48% to 78%, recovering 15 of the 24 that had been blocked. The loop finds different faults elsewhere. ECHOFORUM needed frontend fixes and a page-load speed-up, which took one failing set of 37 tasks from 0 solved to 36. ECHOCHAT's verifier had drifted out of sync with the data, and realigning it lifted the share of gradable tasks from 34% to 99%. ECHOCARE needed one state-wiring fix, and ECHOFORGE had the backend logic but no user-interface control to reach it.

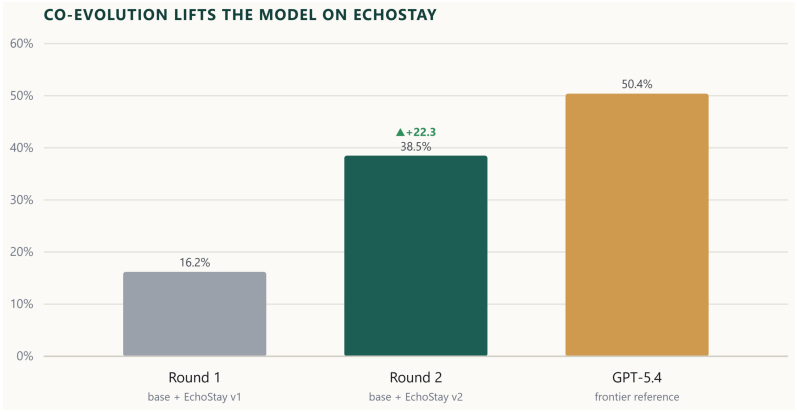

Figure 9: Co-evolution lifts the model on ECHOSTAY. As the world went from v1 to v2, the model trained on its corpus more than doubled, from 16.2% to 38.5%. This is a separate measurement from the world's own solve rate.

As the world sharpens, the model climbs with it (Figure 9). Re-running the loop on ECHOSTAY across two rounds, the model trained on its corpus more than doubles, from 16.2% to 38.5%, two-thirds of the distance to GPT-5.4's 50.4%.

The environment, the corpus and the verifier move together here, and that is what co-evolution is rather than a confound we could have controlled away. Which tasks exist is a function of the environment. A task is grounded on live state and exported only if the feasibility check of Sec. 4.3 finds it reachable through the running interface, so a booking flow that cannot set a guest count cannot yield a task that asks for one. Repairing that control does not merely make existing tasks easier: it brings tasks into existence that could not previously be posed, and the references those tasks are graded against are minted from the repaired database. The two rounds are therefore each measured in their own world, against the tasks that world can pose. An environment-only or verifier-only ablation is not well defined under this design, and what the comparison reports is a round of the loop as a whole rather than the effect of any one repair.

The boundary the live web erases. Inside a controlled environment the distinction between repairing the world and teaching the model is easy to hold, because both are inspectable. The live web erases it. There is no world to repair mid-task, so when an action lands on nothing, correctness rests entirely on the agent noticing and choosing differently. That is the last thing a world has to teach, and it is where imitation is weakest.

7 Reinforcement learning on the worlds

7.1 Imitation carries a ceiling

Every result so far comes from imitation: the 9B model copies the trajectories GPT-5.4 got right. Filtering on the verifier is what makes that corpus clean, and cleanliness is the one thing it cannot teach. A trajectory that only ever goes right contains no instance of a wrong turn being noticed and undone, and none of an agent judging that it is finished. Sec. 6.7 identified those as the residue that survives once the world itself is correct.

Keeping the failed trajectories would not close the gap either, because the failures would still be the teacher's. A frontier model goes wrong in ways a 9B policy mostly does not, and the 9B policy goes wrong in ways the teacher rarely does: misreading a control that is plainly visible, repeating an action on an element that never responds, stopping early on a page it has not understood. A demonstration can only show recovery from the mistake the demonstrator made, so imitation supplies repairs for errors the student will seldom meet and none for the ones it will. Reinforcement learning optimises the outcome we grade on the policy's own rollouts, so the failures it learns to recover from are its own.

Table 7: The requirements reinforcement learning places on an environment, and how the two candidate substrates meet them.

Requirement	Live web	Echoverse
Exact reset to a known state	None. Listings, dates and layouts move, so no two rollouts of the same task begin alike.	Per-task database snapshot and restore (Sec. 4.5).
Episode volume	Hosts throttle and block automated traffic well before the rate RL requires.	Self-contained applications, replicated and run in parallel.
Trustworthy reward	No ground truth is exposed, so the outcome must be inferred from a screenshot by a second model.	The grounded verifier of Sec. 3.2: the outcome is compared against a reference read out of the database, not inferred from the trajectory.
Stationarity within a run	Pages are redesigned and listings roll forward, so a task's meaning drifts during training.	Fixed seed data; a task means the same thing on the first episode and the last.
Safety of destructive actions	Writes hit real accounts, irreversibly.	Safe to break and quick to reset; a corrupted world is discarded rather than repaired.

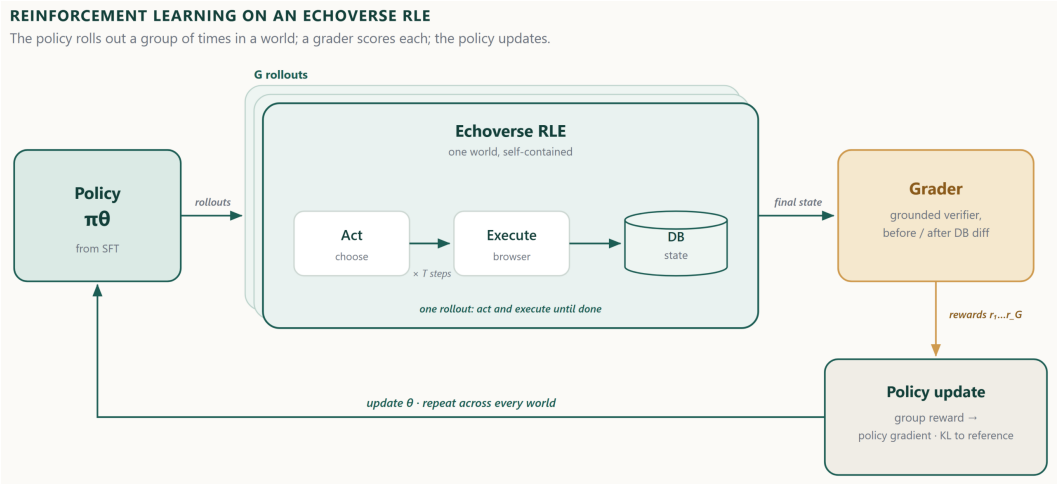

Figure 10: Reinforcement learning on an Echoverse world. From the supervised policy we roll out a group of trajectories in one environment; each is a sequence of act-and-execute steps that changes the database. A grader, the same grounded verifier that filtered the supervised data, sits outside the environment and scores each rollout's final database state into a reward. The group of rewards updates the policy, and the loop repeats across every training world. The diagram shows the grounded trajectory grader only; the run also applies a dense per-step reward described in Sec. 7.3.

7.2 Why the live web is not an RL environment

Reinforcement learning needs an environment it can drive at scale: an exact reset to a known state, throughput to sample in parallel, and a reward it can trust, sustained over a run that replays the same task thousands of times. The live web supplies none of them. Table 7 sets the requirements against both candidate substrates.

Echoverse meets all five by construction and needed no new machinery to do so: the reset primitive is the per-task database isolation introduced in Sec. 4.5 for reproducibility, and the reward is the verifier of Sec. 3.2 that already filtered the supervised data. The same worlds that benchmark an agent train one. Figure 10 shows the resulting loop.

7.3 Method

We take π_{SFT} as the starting policy and optimise with a group-relative policy gradient [Shao et al., 2024], producing π_{RL}, which avoids training a value network over long multimodal browser trajectories. A low-variance KL penalty against a fixed reference anchor keeps the policy near a checkpoint that already speaks the action format.

Each rollout earns two rewards, summed with unit weight:

- a **trajectory reward** from the database-grounded verifier of Sec. 3.2, with GPT-4.1 making the closed comparison between the observed outcome and the reference read out of the database, under the prompts of Appendix C;
- a **dense per-step reward**, binary in each turn, from a multimodal judge (GPT-4.1 vision) that sees the pre-action screenshot with the click target marked. A turn scores 1 when the action demonstrably took effect, that is when state or address changed and the following turn builds on it, and 0 for a repeat, a no-op, an error, a loop, aimless scrolling, or a `terminate` fired before the task was done. The judge is instructed to compare consecutive screenshots and to disregard the agent's own account of what it did, which is otherwise the most common source of unearned credit.

The split is deliberate, and worth stating plainly because it qualifies the grounding argument made elsewhere in this paper. The database settles whether an episode succeeded, which is the quantity we care about, but it is largely silent in the middle of an episode: most intermediate states are neither right nor wrong, so a purely terminal reward gives a fifty-turn trajectory a single bit of signal. The dense term supplies the per-step shaping the database cannot, at the cost of reintroducing a judged signal. Three couplings keep the grounded term in charge. The final step's dense reward is forced to zero whenever the trajectory verdict is wrong, so process credit cannot be farmed by guessing when to stop; terminating with the wrong outcome carries an explicit penalty; and the dense term is applied to training rollouts only, so validation is scored on the grounded trajectory reward alone and the held-out figures in Sec. 7.5 are pure outcome.

Credit assignment over steps. A rollout is not one training example. Each turn becomes its own row, tagged with its step index and with the identifier of the prompt it came from, so a T-turn trajectory emits T rows and a single optimisation step yields far more rows than trajectories. The trajectory reward is broadcast onto every row of its rollout and the step reward is added to the row it belongs to, giving a per-row reward

$$R_{\mathrm{row}} \;=\; R_{\mathrm{traj}} \;+\; \lambda\, R_{\mathrm{step}}, \qquad \lambda = 1, \quad R_{\mathrm{step}} \in \{0, 1\}.$$

The departure from the usual formulation is what the group contains. A vanilla group-relative method groups the G rollouts of a prompt and uses only their final rows. We widen the group to every row that shares a prompt identifier, across all G rollouts and all of their turns, and centre on that group:

$$A_{\mathrm{row}} \;=\; R_{\mathrm{row}} - \mathrm{mean}(R_{\mathrm{group}}).$$

We do not divide by the group standard deviation, following Liu et al. [2025]: normalising by it up-weights groups that happen to be nearly unanimous, which in this setting means the easiest and the hopeless tasks, and it would also destroy the absolute scale on which the two reward terms are commensurable. A single normalisation then carries both signals. A successful trajectory lifts every row it produced, which is the outcome credit a group-relative method already gives; and within the same group a turn scored 1 sits exactly λ above an otherwise identical turn scored 0, so productive turns are preferred inside losing trajectories as well as winning ones.

7.4 Setup

We train on five worlds, ECHOBANK, ECHOFORGE, ECHOFORUM, ECHOSTAY and ECHOTUNES, one from each workflow category of Table 2, on 32 GPUs.

Task pool. The pool is filtered, balanced and disjoint from evaluation. We sample the starting policy four times on every candidate task and keep only those it solves one, two or three times out of four. Both extremes are discarded: a task solved zero times out of four gives a group of rollouts that is uniformly wrong, and a task solved four times out of four gives one that is uniformly right,

Table 8: Reinforcement-learning task pool. Train rows include duplication from upsampling; unique counts do not. Validation is a held-out set per world with zero overlap against training.

World	Train rows	Train unique	Validation	Overlap
ECHOTUNES	60	41 (upsampled)	25	0
ECHOBANK	60	35 (upsampled)	25	0
ECHOFORUM	70	70	25	0
ECHOSTAY	87	87	25	0
ECHOFORGE	120	120	25	0
Total	**397**	**353**	**125**	**0**

Table 9: Reinforcement-learning configuration.

Setting	Value
Policy	
Initialisation	π_{SFT}
Algorithm	Group-relative policy gradient; advantage not normalised by group standard deviation
Learning rate	1×10^{-7}
KL coefficient	0.001, low-variance estimator, fixed anchor
Entropy coefficient	0
Off-policy correction	Token-level rollout importance sampling, threshold 2.0
Batching	
Prompts per step / group size	16 / 8 (128 trajectories per step); micro-batch 2
Rollout	
Temperature / top-p	0.7 / 1.0, top-k disabled; repetition penalty 1.1
Max turns	50
Response / prompt length	1024 / 16000 tokens, 25000 max context
Reward	
Composition	trajectory reward $+ 1.0\times$ step reward
Trajectory judge	GPT-4.1 against the database reference
Per-step judge	GPT-4.1 vision, binary per turn, 768 px pre-action screenshot, target marked
Credit assignment	one row per turn; advantage centred over all step-rows sharing a prompt
Couplings	final step reward zeroed on wrong outcome; wrong-termination penalty 0.1; abort penalty on the final step only; dense term on training rollouts only
Schedule	
Training length	a little over two epochs, 25 steps per epoch
Validation / checkpoint	every 5 / every 10 steps; reported checkpoint at the two-epoch mark, step 50

and in either case the advantage in Sec. 7.3 is identically zero and the task contributes no gradient. Retaining only the mid-range does not guarantee a mixed group, since pass@4 records the behaviour of the starting policy rather than a property of the policy at the step where the task is drawn, but it makes one substantially more likely. Within that pool we balance per-world counts into a band of 60 to 120 rows: ECHOTUNES and ECHOBANK fall below the floor and are upsampled by duplication, ECHOFORGE is capped from 237 to 120 by fixed-seed subsampling. Table 8 gives the resulting mix. The filter does not make the pool easy: of the 353 unique tasks, 189 are labelled hard, 124 medium and 40 easy, and 288 of them require a write (180 READ_WRITE, 108 WRITE, 65 READ). The largest categories are multi-step planning (112 tasks), messaging (38), playlists (25), wishlists (21) and community actions (20).

Optimisation and rollouts. Table 9 lists the configuration. A step draws 16 prompts at group size 8, giving 128 trajectories per step collected in a single wave. Rollouts are sampled at temperature 0.7 with a mild repetition penalty and capped at 50 turns. Note that this is half the 100-action budget used for every evaluation in Sec. 6 and for the held-out validation reported below, so the policy is trained under a tighter budget than it is scored under. Validation is greedy, so the reported score is not inflated by sampling.

7.5 Results

We report two quantities. The *judged score* is the mean trajectory reward over the held-out set, the grounded verifier's verdict of Sec. 3.2, scored on Table 8's validation column. The *training reward* is the mean total reward earned on training rollouts, and so includes the dense per-step term. The judged score rises from 58.8% to 68.0% at the two-epoch mark, step 50, and the training reward trends upward over the same period (Figure 11).

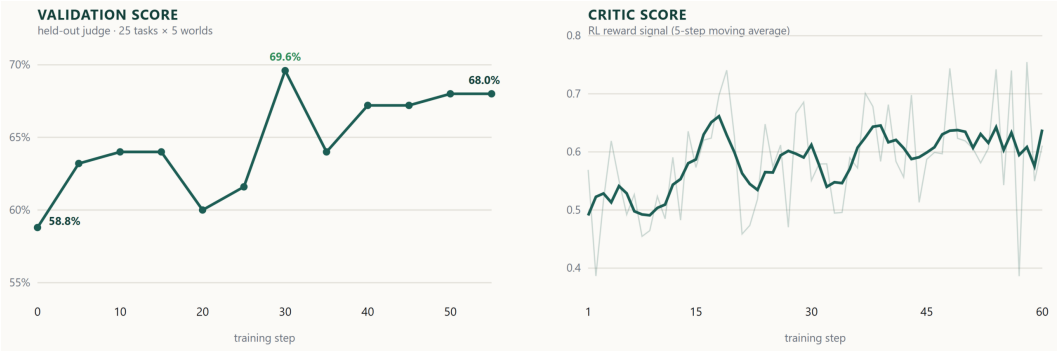

Figure 11: Reinforcement learning on five worlds, a little over two epochs. Left: the held-out judged score, 25 tasks per world graded by the database-grounded verifier. Right: the training reward, which unlike the judged score includes the dense per-step term.

8 Conclusion

A world is more than a benchmark to score against: it is something the loop keeps improving, because the same graded run that measures the model also shows what to fix in the world. Deep worlds transfer where shallow clones pull capability down; one control rebuilt in a hundred forms teaches a skill that carries to held-out widgets and to the open web; co-evolution moves model and world together; and reinforcement learning against the same worlds takes the policy from 58.8% to 68.0% on held-out tasks.

What matters is not the largest inventory of synthetic websites but a factory that finds what an agent cannot yet do, builds or repairs the world that teaches it, and runs the loop again. More deep worlds for the closed domains public benchmarks cannot reach, more capability worlds for the interactions models keep failing, and above all more reinforcement learning against grounded worlds: these interact, since broader worlds admit longer training runs and every round exposes the next capability to build.

We release environment code and graded test tasks for four worlds: the deep domains ECHOSTAY and ECHOFORGE, and the two capability worlds with their in-distribution and held-out splits. Every task carries the database-grounded verifier that scores it, so an agent is measured against application state rather than against a judge, and we release evaluation tasks only so they stay uncontaminated as a measure. Code: `https://aka.ms/echoverse`.

Acknowledgements

We thank Alexey Taymanov, Andrew Zhao, Aravind Rajeswaran, Chinmay Karkar, Luiz Do Valle, Pashmina Cameron, Rafah Hosn, Spencer Whitehead, Zach Nussbaum and Yadong Lu for their valuable help, insightful discussions, and continued support throughout this work.

References

Pranjal Aggarwal, Graham Neubig, and Sean Welleck. Gym-Anything: Turn any software into an agent environment. *arXiv preprint arXiv:2604.06126*, 2026.

Ahmed Awadallah, Yash Lara, Raghav Magazine, Hussein Mozannar, Akshay Nambi, Yash Pandya, Aravind Rajeswaran, Corby Rosset, Alexey Taymanov, Vibhav Vineet, Spencer Whitehead, and Andrew Zhao. Fara-7B: An efficient agentic model for computer use. *arXiv preprint arXiv:2511.19663*, 2025.

Ahmed Awadallah, Sahil Gupta, Yash Lara, Yadong Lu, Hussein Mozannar, Akshay Nambi, Zach Nussbaum, Yash Pandya, Aravind Rajeswaran, Corby Rosset, Alexey Taymanov, Luiz do Valle, Vibhav Vineet, Spencer Whitehead, and Andrew Zhao. Fara-1.5: Scalable learning environments for computer use agents. *arXiv preprint arXiv:2606.20785*, 2026.

Browserbase. Browserbase: Headless browser infrastructure for AI agents. `https://www.browserbase.com`, 2025.

Murray Cox. Inside Airbnb: Adding data to the debate. `http://insideairbnb.com`, 2024.

Thibault Le Sellier de Chezelles, Maxime Gasse, Alexandre Drouin, Massimo Caccia, Léo Boisvert, Megh Thakkar, Tom Marty, Rim Assouel, Sahar Omidi Shayegan, Lawrence Keunho Jang, Xing Han Lù, Ori Yoran, Dehan Kong, Frank F. Xu, Siva Reddy, Quentin Cappart, Graham Neubig, Ruslan Salakhutdinov, Nicolas Chapados, and Alexandre Lacoste. The BrowserGym ecosystem for web agent research. *arXiv preprint arXiv:2412.05467*, 2025.

DeepSeek-AI. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2Web: Towards a generalist agent for the web. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Michael Dennis, Natasha Jaques, Eugene Vinitsky, Alexandre Bayen, Stuart Russell, Andrew Critch, and Sergey Levine. Emergent complexity and zero-shot transfer via unsupervised environment design. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H. Laradji, Manuel Del Verme, Tom Marty, Léo Boisvert, Megh Thakkar, Quentin Cappart, David Vazquez, Nicolas Chapados, and Alexandre Lacoste. WorkArena: How capable are web agents at solving common knowledge work tasks? In *International Conference on Machine Learning (ICML)*, 2024.

Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. WebVoyager: Building an end-to-end web agent with large multimodal models. *arXiv preprint arXiv:2401.13919*, 2024.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Chong Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Ruslan Salakhutdinov, and Daniel Fried. VisualWebArena: Evaluating multimodal agents on realistic visual web tasks. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2023.

Daisy Lin and XJ Wang. A practical recipe for training computer-use agents with reinforcement learning. `https://www.amazon.science/blog/a-practical-recipe-for-training-computer-use-agents-with-rl`, 2026. Amazon Science Blog, Amazon AGI Lab.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Understanding R1-zero-like training: A critical perspective. *arXiv preprint arXiv:2503.20783*, 2025.

Microsoft. Playwright: Fast and reliable end-to-end testing for modern web apps. `https://playwright.dev`, 2024.

Jack Parker-Holder, Minqi Jiang, Michael Dennis, Mikayel Samvelyan, Jakob Foerster, Edward Grefenstette, and Tim Rocktäschel. Evolving curricula with regret-based environment design. In *International Conference on Machine Learning (ICML)*, 2022.

Rémy Portelas, Cédric Colas, Lilian Weng, Katja Hofmann, and Pierre-Yves Oudeyer. Automatic curriculum learning for deep RL: A short survey. *arXiv preprint arXiv:2003.04664*, 2020.

Qwen Team. Qwen3.5: Towards native multimodal agents. `https://qwen.ai/blog?id=qwen3.5`, February 2026.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, Daniel Toyama, Robert Berry, Divya Hakimi, Yuan Xu, Oriana Riva, and Timothy Lillicrap. AndroidWorld: A dynamic benchmarking environment for autonomous agents. In *International Conference on Learning Representations (ICLR)*, 2025.

Corby Rosset, Pratyusha Sharma, Andrew Zhao, Miguel Gonzalez-Fernandez, and Ahmed Awadallah. The art of building verifiers for computer use agents. *arXiv preprint arXiv:2604.06240*, 2026.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

Bowen Wang, Dunjie Lu, Junli Wang, Tianyi Bai, Shixuan Liu, Zhipeng Zhang, Haiquan Wang, Hao Hu, Tianbao Xie, Shuai Bai, Dayiheng Liu, Que Shen, Junyang Lin, and Tao Yu. CUA-Gym: Scaling verifiable training environments and tasks for computer-use agents. *arXiv preprint arXiv:2605.25624*, 2026.

Rui Wang, Joel Lehman, Jeff Clune, and Kenneth O. Stanley. Paired open-ended trailblazer (POET): Endlessly generating increasingly complex and diverse learning environments and their solutions. *arXiv preprint arXiv:1901.01753*, 2019.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Tianci Xue, Weijian Qi, Tianneng Shi, Chan Hee Song, Boyu Gou, Dawn Song, Huan Sun, and Yu Su. An illusion of progress? assessing the current state of web agents. *arXiv preprint arXiv:2504.01382*, 2025.

Yuhao Yang, Zhen Yang, Zi-Yi Dou, Anh Nguyen, Keen You, Omar Attia, Andrew Szot, Michael Feng, Ram Ramrakhya, Alexander Toshev, Chao Huang, Yinfei Yang, and Zhe Gan. UltraCUA: A foundation model for computer use agents with hybrid action. *arXiv preprint arXiv:2510.17790*, 2026.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

Zheng Yuan, Hongyi Yuan, Chengpeng Li, Guanting Dong, Keming Lu, Chuanqi Tan, Chang Zhou, and Jingren Zhou. Scaling relationship on learning mathematical reasoning with large language models. *arXiv preprint arXiv:2308.01825*, 2023.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. STaR: Bootstrapping reasoning with reasoning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

Ziyun Zhang, Zezhou Wang, Xiaoyi Zhang, Zongyu Guo, Jiahao Li, Bin Li, and Yan Lu. InfiniteWeb: Scalable web environment synthesis for GUI agent training. *arXiv preprint arXiv:2601.04126*, 2026. Accepted to ACL 2026.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-Bench and chatbot arena. In *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2023.

Shuyan Zhou. WebArena-Infinity: Generating browser environments with verifiable tasks at scale. `https://webarena.dev/webarena-infinity/`, March 2026.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations (ICLR)*, 2024.

A Naming conventions

Every world in this paper is built on a behaviourally faithful clone of a class of application rather than of any particular product, and is named for the workflow it represents. Public benchmarks and public data sources are referred to by their real names, since they are cited rather than cloned.

B The supervised corpus

Table 10 gives the composition of the supervised fine-tuning corpus of Sec. 4.8, the single body of data behind every π_{SFT} result in Sec. 6. Every trajectory in it is a GPT-5.4 rollout that the grounded verifier of Sec. 3.2 scored as passing; rollouts that failed are discarded rather than kept as negatives. The per-world counts are uneven because the worlds are, in how many distinct task sets they support and in how long a task takes: an ECHOSTAY booking runs to nearly 34 actions on average while a nested-filter submission takes fewer than five. The trajectory-scaling sweep of Sec. 6.6 subsamples this same corpus and holds these proportions fixed at every point, so the only quantity that varies along that axis is volume.

Table 10: The supervised corpus, per world. *Task sets* counts the separately exported task datasets from that world that contribute to the corpus. Steps are agent actions; a trajectory ends when the agent emits `terminate` or exhausts its budget.

World	Task sets	Trajectories	Steps	Avg. steps/traj.
ECHOMAIL	5	4,482	91,155	20.34
ECHOCALENDAR	3	4,109	91,891	22.36
ECHOCHAT	1	832	14,954	17.97
ECHOML	4	2,450	70,397	28.73
ECHOFORGE	1	824	20,596	25.00
ECHOBANK	2	852	7,801	9.16
ECHOCARE	1	703	13,680	19.46
ECHOFORUM	3	1,472	25,149	17.08
ECHOTUNES	2	919	15,759	17.15
ECHOSTAY	3	757	25,586	33.80
Datepicker world	3	1,827	19,475	10.66
Nested-filter world	3	1,782	8,371	4.70
Total	**31**	**21,009**	**404,814**	**19.27**

C Verifier prompts

The comparison described in Sec. 3.2 is a single LLM call per task, made with the prompts reproduced below. Neither judge receives the trajectory, the screenshots, or the agent's own account of what it

did. The read judge sees the goal, the reference answer and the agent's answer. The write judge sees the goal, the reference state change and the `sqldiff` between the pristine per-task database D_0 and the final database D_T, which is what makes the write check a statement about the change rather than about the end state alone. Both return a binary score with a short justification.

C.1 Read tasks

```
You are an evaluation judge. You compare an agent's answer to a reference answer for a Website/
application task.

The task asks the agent to look up information (a "read" task). You must decide
whether the agent's answer is semantically equivalent to the reference answer.

Rules:
- Minor formatting differences (extra whitespace, different date formats, quoting) are OK.
- Synonyms and paraphrases are OK as long as the meaning is the same.
- Monetary amounts may be rounded (e.g., $287.62 and $288 are equivalent; $181.63 and $182 are equivalent)
. Small rounding differences in prices, fees, or totals are acceptable because the agent reads displayed
values from the UI which may round differently than the database.
- Numeric values may be expressed in equivalent units or human-readable formats. For example: 157286400
bytes and "150 MB" are equivalent; 2500000000 and "2.5 billion" or "2,500,000,000" are equivalent;
42000000 and "42M" or "42,000,000" are equivalent. Accept any representation that conveys approximately
the same quantity (within reasonable rounding for unit conversion).
- Counts and IDs should match, but minor differences due to UI display rounding or pagination are
acceptable if the values are close.
- Email addresses and names must match (case-insensitive).
- If the agent provides MORE information than needed but includes the correct answer, that counts as
correct.
- If the agent includes additional context (e.g., a date alongside a time) but the core answer is correct,
 that counts as correct.
- If the agent's answer is completely wrong or missing the key information, it fails.

Return a JSON object with exactly these keys:
{
  "score": <int 0 or 1>,
  "reasoning": "<brief explanation>"
}

score guide:
  1 = answer is semantically correct
  0  = no required changes present or completely wrong changes
```

The user message is instantiated per task as:

```
## Task goal
$question

## Reference answer (ground truth)
$ref_answer

## Agent's answer
$agent_answer

Compare the agent's answer against the reference answer and return your JSON verdict.
```

C.2 Write tasks

```
You are an evaluation judge for a Website/application. You determine whether
a task was completed correctly by comparing the intended state change with
the actual database changes.

You receive:
1. The task goal (what the agent was supposed to do)
2. The expected state change (reference_state_change)
3. The actual SQL diff (output of sqldiff showing what changed in the database)

The SQL diff contains SQL statements that would transform the BEFORE database
into the AFTER database. Common patterns:
- UPDATE table SET col=val WHERE ...  -> a field was changed
- INSERT INTO table ...               -> a new row was added
- DELETE FROM table WHERE ...          -> a row was removed

Rules:
- The agent may have made additional changes beyond what was required (e.g.,
```

```
  marking an email as read while also starring it). Extra changes are OK as
  long as the required change was made.
- If the diff is empty, the agent made no changes -> likely a failure.
- Focus on whether the REQUIRED changes (from reference_state_change) are
  present in the diff.

Return a JSON object with exactly these keys:
{
  "score": <int 0 or 1>,
  "reasoning": "<brief explanation of what matched and what didn't>"
}

score guide:
  1  = all required changes are present
  0  = no required changes present or completely wrong changes
```

The user message is instantiated per task as:

```
## Task goal
$goal

## Expected state change (reference)
$ref_state_change

## Actual database diff (sqldiff output)
$sql_diff

Evaluate whether the required changes were made. Return your JSON verdict.
```

D Capability-world widget catalogue

Every widget family the two capability worlds render, with the interaction each one demands of the agent. Table 11 covers the nested-filter world and Table 12 the datepicker world. Held-out families never appear in training and are used only for evaluation.

Table 11: Nested-filter widget families. Training covers 20 families over 200 generated frontends in 15 visual styles; the held-out group covers 9 compound families over 100 frontends in 9 styles.

Training (in-distribution)		**Held out**	
Family	**Interaction**	**Family**	**Interaction**
Calculator and menu	Compute a value, pick from a dropdown	Booking search	Destination with check-in and check-out dates
Calendar filter	Pick a date to filter by	Override calculator	Calculator with default overrides
Filter dialog	Open an “all filters” modal	Calendar combination	Calendar plus secondary controls
Search field	Type a free-text query	Cost calculator	Region and usage cost estimator
Facet checkboxes	Tick multiple category boxes	Property facets	Beds, baths and price facets
Range slider	Set a minimum and maximum	Modal filter flow	Multi-step modal filters
On/off toggles	Boolean switches	Strict facets	Exact-match facet filter
Filter chips	Click pill toggles on and off	Adoption filters	Species, compatibility and sex
Calculator field	Type or derive a numeric threshold	Full property panel	Complete real-estate filter panel
Active-filter bar	Chips of the applied filters		

Table 12: Datepicker widget families. Training covers 6 core types over 10 contexts, 100 generated frontends in 6 visual styles; the held-out group covers 10 unusual widgets over 36 scenarios, 80 frontends in 9 styles.

Training (in-distribution)		**Held out**	
Family	**Interaction**	**Family**	**Interaction**
Date and time	A date plus a time of day	Calendar heatmap	Grid coloured by density
Single date	Pick one calendar date	Scroll wheel	Spinning wheel columns
Date range	A start and end span	ISO week picker	Select a whole calendar week
Bounded date	Date within minimum and maximum limits	Multi-date	Pick several dates at once
Date of birth	Birthdate entry	Duration stepper	Step an ISO duration up or down
Month and year	Month and year only	Time-only clock	Hours, minutes and seconds, no date
(no further training families)		Quarter picker	Select a fiscal quarter
		Fiscal-year picker	Pick a fiscal year by decade
		Timeline slider	Slide along a date axis