LIVEUIBENCH: A BEHAVIORAL BENCHMARK FOR AGENT-AS-GUI WORKFLOW SURFACES

Anonymous authors
Paper under double-blind review

Project page: `https://liveuibench.pages.dev`

ABSTRACT

Recurring personal work is stateful and accretes across days: a user should be able to return tomorrow to visible, editable task state an assistant created today. The artifact that supports this is an *agent-native app* — an interactive application whose interface, state, and behavior are generated and maintained by an agent — evaluated here in its *Agent-as-GUI workflow surface* form: the generated interface whose controls dispatch into an app-scoped agent workflow that drives real services. Such surfaces are easy to make look done and hard to make do, and whether one genuinely behaves is invisible to look-and-feel evaluation. We present LIVEUIBENCH, a behavioral benchmark that scores any generator on five executable contracts — visible task state, action routing, real-API grounding, grounded visible update, and reopen persistence — checked deterministically over reserved open-submit-revise-reopen episodes against execution traces, API-call records, and AppWorld database diffs, with Full Workflow Resolution (FWR) primary and Assertion Pass Rate (APR) diagnostic. The suite comprises 500 cases over nine AppWorld single-app domains plus the `multi_tool` and `shared_world` families, with 2,386 episodes and 10,480 deterministic assertions. As its strongest available baseline we build GUIOS, an agent-native app runtime engineered contract-by-contract to satisfy this spine, so the concept is shown buildable by the same system the benchmark then tries to refute. A pre-registered measurement plan — six scripted controls calibrating how far APR overstates workflow competence, plus a no-grounding ablation — tests the hypothesis that surface-level contracts are largely buildable while real-API grounding is the open frontier; preliminary evidence from the suite's construction audits is consistent with it.

1 INTRODUCTION

On Monday, a user asks an assistant for a weekly commute-playlist builder that remembers last week's picks. On Tuesday they revise a constraint — swap out anything over four minutes. On Wednesday they reopen the builder before leaving home. Each of today's dominant interaction forms fails this three-day workflow in a different way. A chat answer is one-shot: Tuesday's revision requires restating Monday's context. A static generated mockup looks like a playlist builder, but its buttons dispatch to nothing. Hidden agent memory may retain the facts, yet gives the user no visible object to inspect, edit, or trust. A GUI agent can operate an existing music app, but it cannot create the missing workspace itself. What the user needs is an *agent-native app*: an interactive application whose interface, state, and behavior are generated and maintained by an agent (Figure 1) — a generated interface holding the user's task state, visible, operable, and recoverable across days. It is that surface, not a transcript, that the user reopens on Wednesday.

A concept like this is easy to assert and hard to honor. For it to be more than a slogan it must be both *buildable* (a runtime can produce it from a single sentence) and *falsifiable* (a benchmark can tell a genuine agent-native app from a plausible shell by executing it). This paper's primary contribution is the falsification instrument: LIVEUIBENCH, a behavioral benchmark that scores any generator on five executable contracts (§2) — *visible task state*, *action routing*, *real-API grounding*, *grounded visible update*, and *reopen persistence* — deterministically, from executed behavior rather than from the surface's own claims. To show the concept is buildable, and to give the benchmark a strong baseline, we also build GUIOS: a runtime that turns one whiteboard sentence into a validated spec

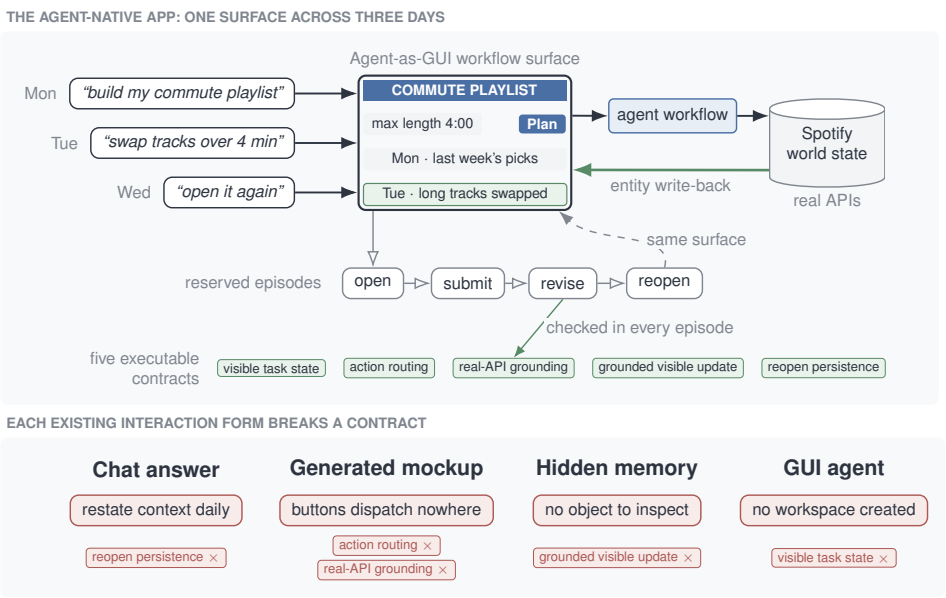

Figure 1: The agent-native app and why it must exist. *Top:* a three-day recurring workflow (generate Monday, revise Tuesday, reopen Wednesday) keeps returning to the same Agent-as-GUI workflow surface, whose controls dispatch into an app-scoped agent workflow that invokes real APIs against world state; badges mark the five executable contracts of §2 — visible task state, action routing, real-API grounding, grounded visible update, and reopen persistence. *Bottom:* each dominant interaction form breaks a contract — chat answer (reopen persistence), static mockup (action routing, real-API grounding), hidden agent memory (grounded visible update), GUI agent (visible task state).

package, rendered by one universal front end as a live interactive app whose controls dispatch to a unified agent backend making real API calls, engineered contract-by-contract to pass. GUIOS enters the benchmark through the same adapter boundary as any other generator: the system we build is the system we then try to refute, so any contract that still fails under it marks a genuine capability frontier rather than a weak baseline.

Evaluating such a surface requires a shift in what benchmarks measure. The evaluated artifact is not a patch (Jimenez et al., 2024), a rebuilt program (Yang et al., 2026), an action trace in existing software (Zhou et al., 2024; Xie et al., 2024; Trivedi et al., 2024), an agent-graded website (Lu et al., 2025), or a one-shot work product (Mialon et al., 2023; Sun et al., 2026). It is a closed interaction loop: a generated surface, controls that route into an app-scoped agent workflow, agent output that becomes visible state, and persistence that lets the user reopen and continue. Correctness is defined behaviorally, by executing the surface through reserved episodes — a stance shared with ProgramBench (Yang et al., 2026) and AppWorld (Trivedi et al., 2024). Section 2 turns the loop into the five executable contracts; the rest of the paper is organized around that shared spine. Our contributions are:

- **Task and contracts.** We define the agent-native app and decompose it into five executable contracts that are simultaneously a benchmark's scored assertions and a runtime's design invariants (Sections 2 and 3).
- **Dataset and deterministic harness.** LIVEUIBENCH: 500 recurring-workflow cases (2,386 episodes, 10,480 deterministic assertions) over nine AppWorld single-app domains plus the `multi_tool` and `shared_world` families, scored by a no-repair harness that verifies real backend effects through database snapshots (Sections 4 and 5).
- **The GUIOS baseline.** We build an agent-native app runtime — a no-fallback multi-agent pipeline and a closed-loop, real-API grounding action turn — and run it through the benchmark's standard adapter boundary as its strongest available baseline (Section 6).
- **Measurement plan and audit evidence.** We pre-register an analysis plan — a six-control suite quantifying how far assertion-level scores overstate workflow competence, and a no-grounding ablation isolating the real-API grounding contract — and report preliminary audit evidence

consistent with grounding being the dominant open capability (Section 7). [PENDING: headline resolve-rate and localization].

2 THE AGENT-NATIVE APP: CONCEPT AND EXECUTABLE CONTRACTS

We define an *agent-native app* as an interactive application whose interface, state, and behavior are generated and maintained by an agent; its *Agent-as-GUI workflow surface* — the term the paper is named for — is the evaluated artifact form of that concept: the generated interface, together with the app-scoped agent workflow its controls dispatch into, that the benchmark admits, executes, and scores. We use "agent-native app" for the concept and "surface" for the evaluated object throughout. The definition excludes three adjacent artifacts (the failure modes of the introduction): *generated UI without behavior* — a mockup is a picture of an app, not an app; *agent memory without a surface* — retained facts the user cannot see give no object to trust; and a *GUI-operating agent* — driving pre-existing software creates no workspace. The agent-native app has all three at once: a visible surface, real behavior behind it, and persistent state the user returns to.

Five executable contracts. A definition earns its keep only when it can be checked. We decompose the agent-native app into five contracts, grouped by the part of the definition each operationalizes. *Interface:* **visible task state** — opening the surface renders task-relevant state and controls. *Behavior:* **action routing** — a control dispatches into the app-scoped agent workflow it declares, not a dead handler — and **real-API grounding** — that workflow invokes the named backend API and mutates the corresponding application database. *State:* **grounded visible update** — the concrete returned entity (a playlist title, an amount, a file path) is written back into user-visible state — and **reopen persistence** — after teardown and reopening, that entity is still visible and editable. A surface is an agent-native app to the exact extent that these five hold; a case resolves only if all five hold across every episode, and we award no case-level partial credit, following the full-completion convention of recent agent exams (Sun et al., 2026).

Real-API grounding is the load-bearing contract. The other four contracts are visible to inspection; grounding is not. A surface can render task state, route a click, echo a plausible result, and persist it — and still have called no real service. Naming an API is not calling it, so grounding is precisely the property a benchmark must check against execution traces, API-call records, and database diffs rather than against the surface's own claims.

One concept, two instantiations — and the bias that entails. Each contract has two faces: an assertion LIVEUIBENCH scores any generator against (§3) and a design invariant GUIOS is engineered to guarantee (§6). Because the two share this spine, the benchmark is co-designed with the strongest system it evaluates, and we scope claims accordingly. First, the contracts are user-observable requirements, not GUIOS internals; any architecture that satisfies the user's recurring workflow — including a code-first generated web app with server persistence behind an adapter — can in principle satisfy them. Second, GUIOS is offered as a baseline, not a superior system: no cross-system ranking is claimed, and adapter kits ship with the release so architecturally different generators pay a documented, inspectable adapter cost. Third, the co-design cuts in the benchmark's favor: a contract that fails even under the runtime engineered to guarantee it marks a genuine capability gap, not a baseline artifact.

3 THE LIVEUIBENCH TASK

Task definition. Given an end-user intention x and optional context c, a system must produce a workflow surface (§2): $\texttt{generate}(x, c) \rightarrow \textit{surface} \mid \textit{failure}$. The surface is serialized as a package containing a manifest, UI description, routes, an orchestrator, a state schema, persistence bindings, and assets. The evaluator never grades this serialization directly. Instead, the evaluator renders the surface and executes a reserved episode sequence against it; the package is merely the carrier of behavior.

Episodes. Every case runs a reserved episode sequence in order, mirroring how a user returns to a workspace over days: *open* the surface; *submit* initial task information through its controls; *revise*

Contract	Episodes	Checked behavior	Typical failure
Visible task state	open	task state and controls render	blank shell, transcript answer
Action routing	submit, revise	events dispatch into app-scoped agent workflow	static controls, dead handlers
Real-API grounding	submit, revise	the named AppWorld API is invoked and the application database changes	API named but never called
Grounded visible update	submit, revise	the concrete returned entity becomes user-visible state	hidden backend memory, generic echo
Reopen persistence	reopen	the entity remains visible and editable after teardown	state reset, route loss

Table 1: The five executable contracts (§2), the episodes that test them, and their typical failures.

constraints or continue the workflow; optional *followup* turns; and *reopen* after teardown. Single-app cases run the canonical four (open–submit–revise–reopen); `multi_tool` cases add followup turns as the workflow spans more services, and `shared_world` cases run two such sequences as episode groups, the second of which must discover state the first really wrote. Episodes are executed through user-level actions only — fill, click, observe — never through privileged APIs into the package.

Contracts and assertions. Table 1 maps the five contracts to the episodes that test them and the assertion types that encode them. Each assertion's expected value is a concrete case entity (e.g., "the visible result names the *Monday Commute* playlist"), so a generator must ground the workflow in real objects rather than echo memorized phrasings.

Metrics. **Full Workflow Resolution** (FWR) is the primary metric: the fraction of cases whose assertions all pass across every episode in the case's sequence. **Assertion Pass Rate** (APR) is reported only as a diagnostic, because a package can pass many layers while never touching a real API. Per-layer pass rates (routing, API invocation, database change, grounded visibility, persistence) localize failures to a contract; verifying environment state through database diffs follows AppWorld (Trivedi et al., 2024; Yao et al., 2024), applied here to surfaces the agent itself generated.

4 THE LIVEUIBENCH DATASET

LIVEUIBENCH targets recurring personal workflows in which a persistent visible surface is more appropriate than a one-off answer, and grounds every case in the executable application APIs of AppWorld (Trivedi et al., 2024). Each case exposes only intention and context to the generator, while the grounding metadata (target APIs) and assertions remain evaluator-side.

App domains and families. The 500 cases span nine single-app AppWorld domains (`spotify`, `amazon`, `gmail`, `phone`, `venmo`, `splitwise`, `todoist`, `simple_note`, `file_system`) plus the two compositional families of §3: `multi_tool` (one app spanning three or more services) and `shared_world` (two apps over one world). By type, the suite is 302 `single_app`, 130 `multi_tool`, and 68 `shared_world` cases, so roughly 40% are cross-service. Every intention is a first-person request for a recurring, stateful agent-native app — a weekly commute-playlist builder, a shared-expense settle-up tracker, a download-folder sorter — whose workflow necessarily drives named AppWorld APIs (Table 3 and the per-domain distribution are in Appendix A).

Scale and splits. The suite runs 2,386 episodes carrying 10,480 deterministic assertions — roughly 21 per case — typed per contract, from API invocation and database change to grounded visibility and reopen persistence (per-type counts in Appendix A). Splits are stratified within every app domain and family: 147 development, 172 public-test, and 181 sealed-test cases. The sealed split is withheld for hosted evaluation, following the contamination-control practice of live and sealed benchmarks (Jain et al., 2024; Sun et al., 2026). A fully worked case with its episode schema appears in Appendix H.

Construction controls. Intentions and assertions were authored by two multi-agent pipelines, adversarially reviewed (every grounding value a concrete case entity, every named API in the domain

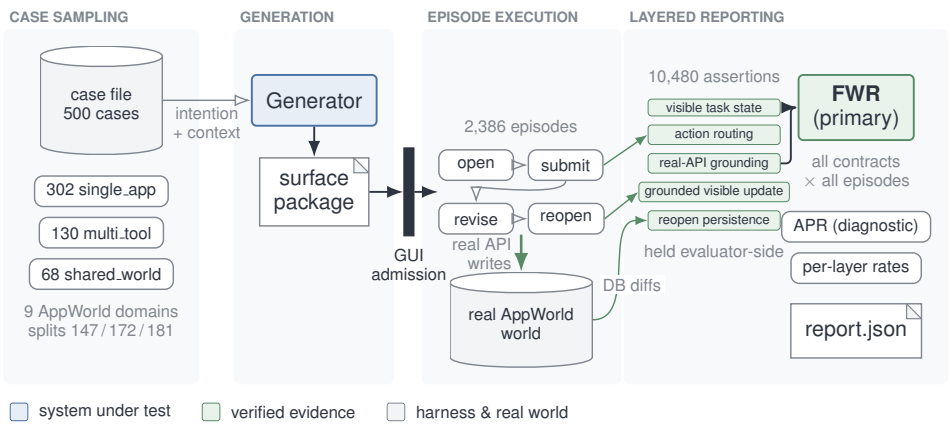

Figure 2: Benchmark pipeline. Case sampling exposes only intention and context to the generator, which returns a workflow-surface package; after the GUI admission gate, the harness executes the reserved open-submit-revise-reopen episodes against the real AppWorld world, and execution evidence plus database diffs feed the evaluator-side deterministic assertions over the five contracts. Layered reporting aggregates them into FWR (primary), with APR and per-layer rates as diagnostics, archived per run.

catalog), and validated mechanically; domain labels are never exposed to the generator. Two live-world audits then repaired cases via a tracked override layer, grew the seedable world fixtures to 367 files, and quarantined unsatisfiable cases; the audited suite drives the behavioral path below, and Appendix A details the protocol.

5 Evaluation Harness

Adapter boundary. All systems are evaluated through a single adapter boundary: the harness supplies intention and context, and receives a workflow-surface package or an explicit failure — from an end-to-end agent, a code-first pipeline behind an adapter, or a scripted control alike. The harness never repairs missing routes, missing state, invalid assets, or unrenderable surfaces; a broken package fails exactly as it would for a user.

Deterministic episode scoring. The benchmark runner executes each case's episodes against the generated surface and checks every assertion against observable execution artifacts. Routing is read from the runtime's agent trace; API invocation from the per-action tool-call record; database change from AppWorld database snapshots diffed before and after each call; grounded visibility from the surface's user-visible result state; persistence from a fresh state query after teardown. Because AppWorld apps are real executable services with real databases, "the workflow created the *Monday Commute* playlist" is a checkable fact, not a rubric judgment. Figure 2 summarizes the pipeline from case sampling through layered reporting; because the harness never edits the surface, "could not render a plausible surface" stays distinguishable from "rendered a surface that does not behave," a distinction the layered metrics quantify.

Validation of the scorer. Beyond the deterministic runner, the harness exposes two evaluator endpoints over the same generated surface (Appendix B): a *GUI evaluator* that gates admission with deterministic rules, and a *behavior evaluator* that drives long-horizon exploratory sessions online through the same runtime action path, ending in an LLM judgment over the full trajectory with archived snapshots and frames (Appendix F). All reported scores come from the deterministic runner; the behavior evaluator's role is validation from the user's side of the screen. On a stratified public probe the two agreed on [PENDING: judge-vs-deterministic agreement on the 500-case suite, e.g. N of M cases]; disagreements are adjudicated by re-executing the deterministic run, so the probe checks the runner rather than contributing a benchmark score.

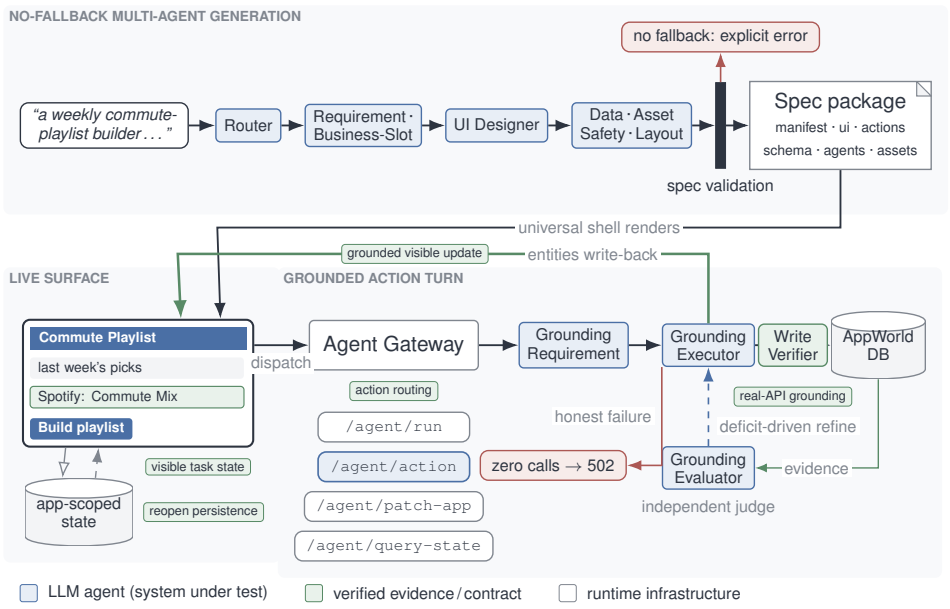

Figure 3: The GUIOS runtime end to end. A whiteboard sentence enters a no-fallback multi-agent generation pipeline; the designed surface must pass spec validation or the runtime returns an explicit error, and the admitted spec package renders through one universal shell as a live app card (*visible task state*). Every control dispatches through the Agent Gateway (*action routing*) into a grounded action turn: real AppWorld API calls, write-first and read-verify (*real-API grounding*); an independent evaluator judging only execution evidence; and returned entities written into the visible result (*grounded visible update*), which survives reopen (*reopen persistence*). An action wired to real permissions that makes zero real calls returns HTTP 502: honest failure, never fabrication.

6 The GUIOS Baseline: An Agent-Native App Runtime

LiveUIBench needs a strong baseline: a generator engineered, contract by contract, to pass. GUIOS is that system — a runtime that turns one natural-language whiteboard sentence ("a weekly commute-playlist builder that remembers last week's picks") into a live, grounded, interactive application (Figure 3). It enters through the same adapter boundary as any other generator (§5), and it embodies three design principles we argue must hold of *any* generator one intends to measure honestly; the full mechanism inventory is in Appendix C.

Principle 1: the app is a spec package, not a code repo. GUIOS never emits application source. Each generated app is a validated declarative *spec package* — manifest, UI description in our A2UI surface-description format, control-to-workflow action bindings, app-scoped state schema, backend agent workflow, and sanitized assets — rendered by one universal shell. Because the artifact is declarative, well-formedness is checkable *before* admission against a fixed component vocabulary, a binding whitelist with an acyclicity and exactly-once reference requirement, and per-action capability permissions (Appendix C). Controls dispatch through an agent gateway and run no front-end business logic, so a control that renders is a control that routes (*visible task state*, *action routing*). A generator emitting opaque code offers the harness no such pre-admission guarantee.

Principle 2: no fallback, honest failure. Generation uses a multi-agent pipeline with no app-type templates and no canned-app fallback: when a stage cannot produce a valid artifact, the runtime returns an explicit error rather than a plausible shell, and an action wired to real permissions that makes zero real calls returns an explicit failure rather than fabricating a local success. This is the runtime-side mirror of the harness's no-repair stance (§5), and it is what makes measured failures attributable: a generator that papers over broken behavior would inflate surface-contract scores and contaminate exactly the localization the benchmark exists to provide.

Principle 3: verify evidence, never swap semantics. Real-API grounding is the hypothesized hardest contract, so each action turn closes a quality loop: a requirement agent parses the submission into a goal contract; an executor makes real AppWorld API calls — write-first, read-verify — and writes the returned entities wholesale into user-visible state (*grounded visible update*); an *independent* evaluator agent judges only the execution evidence — goal contract, call log, visible result, never model narration — and drives a bounded, deficit-driven refine loop that stops on zero progress, in the lineage of self-refinement and verifier-guided retrying (Madaan et al., 2023; Shinn et al., 2023). Critically, the in-loop *Write Verifier* may edit a call's arguments but may never swap the API the workflow selected: a verifier that silently substituted APIs would repair grounding failures the benchmark is supposed to measure. Together the constraints keep the baseline strong without making it self-grading.

7 EXPERIMENTS

We present a registered-report-style measurement plan: the systems, controls, ablation, and analyses are fixed here in advance, and every model-performance cell in Table 2 is marked [PENDING: value from archived run artifact] until compiled from an archived run artifact by checked-in scripts (Appendix E). Two kinds of claims are distinguished throughout: *structural* claims that hold by construction of the suite and require no run, and *empirical* hypotheses the runs will test. The plan tests two hypotheses: that real-API grounding, not surface fluency, is the dominant unsolved capability of frontier agentic systems, and that assertion-level scores overstate workflow competence. All evaluation is over the 172-case public-test split of the 500-case suite; sealed-split evaluation is reserved for hosted leaderboard runs.

Control suite. Six scripted controls isolate the false positives that assertion-level scoring invites — *direct answer*, *static shell*, *route-only*, *hidden state*, *schema-first*, and *multi-agent* — each constructed to pass some contracts and fail a targeted one (Table 4, Appendix A), so the control rows of Table 2 calibrate what APR alone would overcount.

Evaluated systems and ablation. We evaluate the end-to-end GUIOS baseline: the GUIOS generator plus its agentic backend executor that, on each GUI action, selects and invokes real AppWorld APIs (authentication handled by the harness bridge) and writes the returned entities into user-visible state. We run this baseline on multiple frontier backing models ([PENDING: frontier model A], [PENDING: frontier model B]) behind one OpenAI/OpenRouter-compatible endpoint; the harness resolves the LLM configuration per request, retries only transient transport faults (never scored), and resets AppWorld to a pristine world before every run (full runtime configuration in Appendix E). To break the circularity of evaluating only the benchmark authors' own runtime, the plan additionally registers a *single-LLM spec emitter*: one frontier model prompted to emit a complete workflow-surface package in one shot through the same adapter boundary, with no pipeline, no validation retry, and no grounding loop. A *no-grounding ablation* disables the API-invoking executor while keeping the identical generator, isolating the real-API grounding contract; by construction it issues no real calls, so it passes zero API-invocation and database-change assertions (a structural fact, not a measurement) and thereby bounds how much residual score is surface behavior. The ablation runs on a subset of the public split (case count in Table 2); the full run configuration is logged per run (Appendix E).

Registered analyses. The registered analyses below are fixed in advance; each states what is structural versus what the archived artifacts will decide.

1. *APR–FWR gap.* Structurally, the hidden-state and multi-agent controls reach non-trivial APR at FWR $= 0$ (Table 4), so a single assertion-level score overstates workflow competence by construction. Empirically, the runs test whether the gap also holds for the baseline (APR [PENDING: baseline APR] against FWR [PENDING: baseline FWR]); the control rows ([PENDING: control APR]) calibrate its magnitude.
2. *Contract localization.* Hypothesis: the surface contracts are largely buildable — routing [PENDING: route rate], reopen persistence [PENDING: persist rate], grounded visibility [PENDING: grounded rate] — while real API invocation ([PENDING: API-invocation rate]) and database change ([PENDING: DB-change rate]) lag, making grounding GUI actions in authenticated,

System	Cases	FWR	APR	Route	API	DB	Grounded	Persist
Direct answer	172	[PENDING: FWR]	[PENDING: APR]	[PENDING: route]	[PENDING: API]	[PENDING: DB]	[PENDING: grounded]	[PENDING: persist]
Static shell	172	[PENDING: FWR]	[PENDING: APR]	[PENDING: route]	[PENDING: API]	[PENDING: DB]	[PENDING: grounded]	[PENDING: persist]
Route-only	172	[PENDING: FWR]	[PENDING: APR]	[PENDING: route]	[PENDING: API]	[PENDING: DB]	[PENDING: grounded]	[PENDING: persist]
Hidden state	172	[PENDING: FWR]	[PENDING: APR]	[PENDING: route]	[PENDING: API]	[PENDING: DB]	[PENDING: grounded]	[PENDING: persist]
Schema-first	172	[PENDING: FWR]	[PENDING: APR]	[PENDING: route]	[PENDING: API]	[PENDING: DB]	[PENDING: grounded]	[PENDING: persist]
Multi-agent control	172	[PENDING: FWR]	[PENDING: APR]	[PENDING: route]	[PENDING: API]	[PENDING: DB]	[PENDING: grounded]	[PENDING: persist]
GUIOS baseline ([PENDING: frontier model A])	172	[PENDING: FWR]	[PENDING: APR]	[PENDING: route]	[PENDING: API]	[PENDING: DB]	[PENDING: grounded]	[PENDING: persist]
GUIOS baseline ([PENDING: frontier model B])	172	[PENDING: FWR]	[PENDING: APR]	[PENDING: route]	[PENDING: API]	[PENDING: DB]	[PENDING: grounded]	[PENDING: persist]
single-LLM spec emitter ([PENDING: frontier model A])	172	[PENDING: FWR]	[PENDING: APR]	[PENDING: route]	[PENDING: API]	[PENDING: DB]	[PENDING: grounded]	[PENDING: persist]
no-grounding ablation ([PENDING: frontier model A])	[PENDING: subset]	[PENDING: FWR]	[PENDING: APR]	[PENDING: route]	[PENDING: API=0 by constr.]	[PENDING: DB=0 by constr.]	[PENDING: grounded]	[PENDING: persist]

Table 2: Public-split measurement plan (172-case public-test split of the audited suite). All metric columns are pass rates in $[0, 1]$, compiled from archived run artifacts (Appendix E); Route = events entering the app-scoped agent workflow, API = named AppWorld API invoked, DB = application database mutated, Grounded = concrete returned entity visible, Persist = entity survives reopen; APR = fraction of assertions passing (diagnostic). Control rows (upper block) each fail a targeted contract by construction and calibrate the metrics; the no-grounding ablation's API and DB cells are structurally zero.

correctly-parameterized API execution the dominant unsolved capability. The no-grounding ablation (zero API/DB by construction) bounds exactly how much of the remaining score is surface behavior, and the single-LLM spec emitter tests whether the localization pattern is specific to the GUIOS architecture or generic to current generators.

3. *Model and family discrimination.* The resolve-rate spread across backing models ([PENDING: per-model FWR]) tests whether the benchmark discriminates tool-grounding ability rather than surface fluency; if it does, the advantage should concentrate in the API-invocation contract. Relatedly: single-user CRUD app domains are hypothesized tractable ([PENDING: per-domain resolved counts]) while multi-party ledger app domains (`venmo`, `splitwise`, `todoist`) and `shared_world` discovery resolve near zero ([PENDING: ledger/shared_world resolved counts]), leaving workflows whose APIs require coordinating other principals' state open.

Preliminary behavioral evidence from the construction audits. Although the registered runs are pending, the suite's two construction audits (Section 4) already yield behavioral evidence: repeated live sampling under a fixed no-repair configuration with archived artifacts. Five reconstructed sessions (Appendix G) are consistent with the plan's hypotheses, scoped as audit-time observations rather than benchmark scores. In every session the surface rendered, routed, and persisted; the failures concentrated in grounding, and the dominant failure family — multi-party ledger state discipline — decomposed into distinct, reproducible arithmetic-encoding errors rather than one monolithic weakness: *divisor anchoring*, *wrong-service binding*, *duplicate-write inflation*, and *aggregation-from-first-page*, each reconstructed at the API-call level in Appendix G. A fifth session under the same configuration completed a read-route-write workflow end to end, including reopen — evidence that the repaired cases are satisfiable and that these failures measure capability, not benchmark defects. Notably, the failures survive the runtime engineered contract-by-contract to prevent them, which is what makes real-API grounding the frontier the benchmark is built to track.

8 Related Work

Operating existing software. GUI, OS, and app-operation benchmarks evaluate agents acting in environments that already exist: web navigation and grounding (Shi et al., 2017; Deng et al., 2023; Zhou et al., 2024; Koh et al., 2024; Yao et al., 2022; He et al., 2024), desktop and mobile control (Xie et al., 2024; Rawles et al., 2024; 2023; Kapoor et al., 2024), office and enterprise work (Drouin et al., 2024; Wang et al., 2024; 2025), safety and live-web robustness (Levy et al., 2024; Garg et al., 2025; Anupam et al., 2025), and app ecosystems with database-level state checks (Trivedi et al., 2024); GUI-agent systems and data pipelines (Xu et al., 2024; Sun et al., 2025) consume such environments. AppWorld in particular grades tasks by verifying resulting environment state rather than agent output, a philosophy LiveUIBench adopts — but LiveUIBench moves the evaluated object one step earlier: the agent must *create* the environment that later interactions operate.

Generating software and grading agent work. Code-generation benchmarks grade code as the artifact (Chen et al., 2021; Austin et al., 2021; Hendrycks et al., 2021; Lai et al., 2023; Lu et al., 2021; Cassano et al., 2023; Jain et al., 2024; Zhuo et al., 2024; Ren et al., 2020) and repair benchmarks grade patches (Just et al., 2014; Lin et al., 2017; Widyasari et al., 2020; Madeiral et al., 2019; Karampatsis

& Sutton, 2020; Le Goues et al., 2015; Tan et al., 2017; Jimenez et al., 2024; Aleithan et al., 2024; Tian et al., 2026); ProgramBench grades a rebuilt program's behavior under reserved tests and fuzzed probes (Yang et al., 2026), a stance LIVEUIBENCH shares while changing the artifact from a program to a user-operable workflow surface with an interaction loop. Tool and workflow-agent benchmarks stress tool calling, multi-turn interaction, and professional outcomes (Liu et al., 2023; Qin et al., 2023; Li et al., 2023; Yao et al., 2024; Mialon et al., 2023; Yoran et al., 2024; Shridhar et al., 2021; Wang et al., 2022), grading the work an agent performs rather than the persistent surface it leaves behind. Agents' Last Exam grades complete work products with full-completion scoring (Sun et al., 2026); LIVEUIBENCH applies the same all-or-nothing convention at the case level, but to a recurring interaction loop rather than a single deliverable.

Generated interfaces. Work on generated and malleable interfaces splits along two evaluation axes. Generative-interface research synthesizes UIs from queries or evolving user data (Chen et al., 2025; Cao et al., 2025; Vaithilingam et al., 2024; Park et al., 2023) and evaluates authoring experience or human preference — *GUI evaluation without behavior evaluation*; deployed assistants likewise generate interactive artifacts and canvases inside chat products with no behavioral contract on routing, grounding, or persistence. Conversely, WebGen-Bench scores generated websites by dispatching an LLM navigation agent over per-functionality test cases (Lu et al., 2025), and AUI-Gym positions a computer-use agent as judge of task solvability on generated, agent-oriented UIs (Lin et al., 2025) — *behavior evaluation without GUI admission standards*, scored per test by model-based judges within a single session. LIVEUIBENCH requires both axes over one shared runtime: deterministic admission of the surface, deterministic assertions over reserved episodes, and credit only for reopen-persistent, fully resolved workflows. LIVEUIBENCH follows current benchmark-methodology practice on contamination control, documentation, and shortcut analysis (Jain et al., 2024; Liang et al., 2023; Gebru et al., 2021; Geirhos et al., 2020), with public and sealed splits, artifact-backed reports, and layer-attributed failures instead of a single score. Table 8 (Appendix A) summarizes the positioning.

9 LIMITATIONS

First, asset validity and renderability checks are proxy diagnostics; LIVEUIBENCH measures verifiable interaction outcomes, not subjective GUI preference or visual design quality. Second, workflows are synthetic — no real personal data, habit drift, or third-party integrations — a deliberate trade: synthetic AppWorld services make every assertion deterministically checkable, and the grounding bottleneck they expose would only be harder against live services. Third, public-split results are reported from archived artifacts, while sealed-split evaluation is reserved for hosted execution; until sealed-run artifacts are archived, no leaderboard claim is made. Fourth, code-first systems require adapters that preserve surface/action/state semantics, and adapter quality is a confound we report alongside results; relatedly, the benchmark is co-designed with its strongest baseline, a bias we scope explicitly in §2. Fifth, intentions and assertions were pipeline-authored, adversarially reviewed by other pipelines, and repaired through two live-world audits, but the suite has not undergone an exhaustive independent human sign-off pass, which the hosted release will add.

10 CONCLUSION

LIVEUIBENCH makes the agent-native app falsifiable: a workflow is resolved only when all five contracts hold in every reserved open-submit-revise-reopen episode, six scripted controls calibrate how far assertion-level scores overstate workflow competence, and a no-grounding ablation isolates the one contract invisible to look-and-feel evaluation. GUIOS, built contract-by-contract to pass, shows the concept is buildable and gives the benchmark a strong, honest-failure baseline. The registered plan tests whether real-API grounding remains the open frontier; the construction-audit evidence — ledger arithmetic-encoding failures that survive a runtime engineered to prevent them — already points that way. [PENDING: headline resolve-rate and localization from archived run artifacts].

ETHICS STATEMENT

LIVEUIBENCH is fully synthetic: intentions, contexts, and assertions are authored against the simulated AppWorld application environment and contain no real user data, personal identifiers, or

scraped content. No human subjects were involved, and no real third-party service is called during evaluation — all APIs are AppWorld's executable simulations. The benchmark targets everyday personal workflows (planning, record keeping, shared expenses), and the evaluator checks behavioral contracts rather than content preferences. The sealed split is withheld to limit contamination, and leaderboard claims require hosted execution with archived reports.

Reproducibility Statement

All empirical numbers are generated from archived run artifacts (`report.json` and `run-summary.json`) by checked-in scripts rather than transcribed by hand. Appendix E defines the run-manifest and claim-traceability format that every reported run satisfies. The tracked case file, split construction rules, and evaluator endpoints are described in Sections 4 and 5.

Use of Large Language Models

LLMs assisted manuscript editing. Inside the benchmark, multi-agent pipelines authored and adversarially reviewed intentions and assertions (Section 4), and the behavior evaluator ends in an LLM trajectory judgment (Section 5). All reported metrics derive from deterministic assertions only; the sole LLM-derived statistic is the agreement probe, a check on the runner rather than a benchmark result.

References

Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song Wang. SWE-Bench+: Enhanced coding benchmark for llms, 2024. URL https://arxiv.org/abs/2410.06992.

Sagnik Anupam, Davis Brown, Shuo Li, Eric Wong, Hamed Hassani, and Osbert Bastani. BrowserArena: Evaluating LLM agents on real-world web navigation tasks, 2025. URL https://arxiv.org/abs/2510.02418.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. In *arXiv preprint arXiv:2108.07732*, 2021. URL https://arxiv.org/abs/2108.07732.

Yining Cao, Peiling Jiang, and Haijun Xia. Generative and malleable user interfaces with generative and evolving task-driven data model, 2025. URL https://arxiv.org/abs/2503.04084.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q. Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. MultiPL-E: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 2023. URL https://arxiv.org/abs/2208.08227.

Jiaqi Chen, Yanzhe Zhang, Yutong Zhang, Yijia Shao, and Diyi Yang. Generative interfaces for language models, 2025. URL https://arxiv.org/abs/2508.19227.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2Web: Towards a generalist agent for the web. In *Advances in Neural Information Processing Systems*, 2023. URL https://arxiv.org/abs/2306.06070.

Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H. Laradji, Manuel Del Verme, Tom Marty, Leo Boisvert, Megh Thakkar, Quentin Cappart, David Vazquez, Nicolas Chapados, and Alexandre Lacoste. WorkArena: How capable are web agents at solving common knowledge work tasks?, 2024. URL https://arxiv.org/abs/2403.07718.

Divyansh Garg, Shaun VanWeelden, Diego Caples, Andis Draguns, Nikil Ravi, Pranav Putta, Naman Garg, Tomas Abraham, Michael Lara, Federico Lopez, James Liu, Atharva Gundawar, Prannay Hebbar, Youngchul Joo, Jindong Gu, Charles London, Christian Schroeder de Witt, and Sumeet Motwani. REAL: Benchmarking autonomous agents on deterministic simulations of real websites, 2025. URL https://arxiv.org/abs/2504.11543.

Timnit Gebru, Jamie Morgenstern, Briana Vecchione, Jennifer Wortman Vaughan, Hanna Wallach, Hal Daumé III, and Kate Crawford. Datasheets for datasets. *Communications of the ACM*, 64(12): 86–92, 2021. doi: 10.1145/3458723. URL https://doi.org/10.1145/3458723.

Robert Geirhos, Jörn-Henrik Jacobsen, Claudio Michaelis, Richard Zemel, Wieland Brendel, Matthias Bethge, and Felix A. Wichmann. Shortcut learning in deep neural networks. *Nature Machine Intelligence*, 2:665–673, 2020. doi: 10.1038/s42256-020-00257-z. URL https://doi.org/10.1038/s42256-020-00257-z.

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, 2024. URL https://arxiv.org/abs/2401.13919.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with APPS. In *Advances in Neural Information Processing Systems Datasets and Benchmarks Track*, 2021. URL https://arxiv.org/abs/2105.09938.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida I. Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code, 2024. URL https://arxiv.org/abs/2403.07974.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2310.06770.

René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis*, 2014. doi: 10.1145/2610384.2628055. URL https://doi.org/10.1145/2610384.2628055.

Raghav Kapoor, Yash Parag Butala, Melisa Russak, Jing Yu Koh, Kiran Kamble, Waseem Alshikh, and Ruslan Salakhutdinov. OmniACT: A dataset and benchmark for enabling multimodal generalist autonomous agents for desktop and web, 2024. URL https://arxiv.org/abs/2402.17553.

Rafael-Michael Karampatsis and Charles Sutton. How often do single-statement bugs occur? the ManySStuBs4J dataset of single-statement bugs. In *Proceedings of the International Conference on Mining Software Repositories*, 2020. URL https://arxiv.org/abs/1905.13334.

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, 2024. URL https://arxiv.org/abs/2401.13649.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida I. Wang, and Tao Yu. DS-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, 2023. URL https://arxiv.org/abs/2211.11501.

Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, 2015. doi: 10.1109/TSE.2015.2454513. URL https://doi.org/10.1109/TSE.2015.2454513.

Ido Levy, Ben Wiesel, Sami Marreed, Alon Oved, Avi Yaeli, Nir Mashkif, and Segev Shlomov. ST-WebAgentBench: A benchmark for evaluating safety and trustworthiness in web agents, 2024. URL https://arxiv.org/abs/2410.06703.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. API-Bank: A comprehensive benchmark for tool-augmented LLMs, 2023. URL https://arxiv.org/abs/2304.08244.

Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, et al. Holistic evaluation of language models. *Transactions on Machine Learning Research*, 2023. URL https://arxiv.org/abs/2211.09110.

Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. QuixBugs: A multi-lingual program repair benchmark set based on the Quixey challenge. In *Proceedings Companion of the ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, 2017. doi: 10.1145/3135932.3135941. URL https://doi.org/10.1145/3135932.3135941.

Kevin Qinghong Lin, Siyuan Hu, Linjie Li, Zhengyuan Yang, Lijuan Wang, Philip Torr, and Mike Zheng Shou. Computer-use agents as judges for generative user interface, 2025. URL https://arxiv.org/abs/2511.15567.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. AgentBench: Evaluating LLMs as agents, 2023. URL https://arxiv.org/abs/2308.03688.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Advances in Neural Information Processing Systems Datasets and Benchmarks Track*, 2021. URL https://arxiv.org/abs/2102.04664.

Zimu Lu, Yunqiao Yang, Houxing Ren, Haotian Hou, Han Xiao, Ke Wang, Weikang Shi, Aojun Zhou, Mingjie Zhan, and Hongsheng Li. WebGen-Bench: Evaluating LLMs on generating interactive and functional websites from scratch, 2025. URL https://arxiv.org/abs/2505.03733.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. BEARS: An extensible Java bug benchmark for automatic program repair studies. In *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*, 2019. doi: 10.1109/SANER.2019.8667991. URL https://doi.org/10.1109/SANER.2019.8667991.

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. GAIA: A benchmark for general AI assistants, 2023. URL https://arxiv.org/abs/2311.12983.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, 2023. doi: 10.1145/3586183.3606763. URL https://doi.org/10.1145/3586183.3606763.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating large language models to master 16000+ real-world APIs, 2023. URL https://arxiv.org/abs/2307.16789.

Christopher Rawles, Alice Li, Daniel Rodriguez, Oriana Riva, and Timothy Lillicrap. Android in the wild: A large-scale dataset for Android device control. In *Advances in Neural Information Processing Systems*, 2023. URL https://arxiv.org/abs/2307.10088.

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 Tyamagundlu, Timothy Lillicrap, and Oriana Riva. AndroidWorld: A dynamic benchmarking environment for autonomous agents, 2024. URL https://arxiv.org/abs/2405.14573.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. CodeBLEU: A method for automatic evaluation of code synthesis, 2020. URL https://arxiv.org/abs/2009.10297.

Tianlin Shi, Andrej Karpathy, Linxi Fan, Jonathan Hernandez, and Percy Liang. World of bits: An open-domain platform for web-based agents. In *International Conference on Machine Learning*, 2017. URL https://proceedings.mlr.press/v70/shi17a.html.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Cote, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. ALFWorld: Aligning text and embodied environments for interactive learning. In *International Conference on Learning Representations*, 2021. URL https://arxiv.org/abs/2010.03768.

Qiushi Sun, Kanzhi Cheng, Zichen Ding, Chuanyang Jin, Yian Wang, Fangzhi Xu, Zhenyu Wu, Chengyou Jia, Liheng Chen, Zhoumianze Liu, Ben Kao, Guohao Li, Junxian He, Yu Qiao, and Zhiyong Wu. OS-Genesis: Automating GUI agent trajectory construction via reverse task synthesis. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, pp. 5555–5579, 2025. doi: 10.18653/v1/2025.acl-long.277. URL https://aclanthology.org/2025.acl-long.277/.

Yiyou Sun, Xinyang Han, Weichen Zhang, Yuanbo Pang, Tianyu Wang, Yuhan Cao, Yixiao Huang, Chris Duroiu, Haoyun Zhang, Jeffrey Lin, Weishu Zhang, Tyler Zeng, Ying Yan, Bo Liu, Hanson Wen, Mingyang Xu, Xiaoyuan Liu, Zimeng Chen, Weiyan Shi, Amanda Dsouza, Vincent Sunn Chen, Dawn Song, et al. Agents' last exam, 2026. URL https://arxiv.org/abs/2606.05405.

Shin Hwei Tan, Jooyong Yi, Yulis, Sergey Mechtaev, and Abhik Roychoudhury. Codeflaws: A programming competition benchmark for evaluating automated program repair tools. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pp. 180–182, 2017. doi: 10.1109/ICSE-C.2017.76. URL https://doi.org/10.1109/ICSE-C.2017.76.

Muxin Tian, Zhe Wang, Blair Yang, Zhenwei Tang, Kunlun Zhu, Honghua Dong, Hanchen Li, Xinni Xie, Guangjing Wang, and Jiaxuan You. SWE-Bench Mobile: Can large language model agents develop industry-level mobile applications?, 2026. URL https://arxiv.org/abs/2602.09540.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *Annual Meeting of the Association for Computational Linguistics*, 2024. URL https://arxiv.org/abs/2407.18901.

Priyan Vaithilingam, Elena L. Glassman, Jeevana Priya Inala, and Chenglong Wang. DynaVis: Dynamically synthesized ui widgets for visualization editing. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 2024. doi: 10.1145/3613904.3642639. URL https://doi.org/10.1145/3613904.3642639.

Ruoyao Wang, Peter Jansen, Marc-Alexandre Cote, and Prithviraj Ammanabrolu. ScienceWorld: Is your agent smarter than a 5th grader? In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2022. doi: 10.18653/v1/2022.emnlp-main.775. URL https://aclanthology.org/2022.emnlp-main.775/.

Weixuan Wang, Dongge Han, Daniel Madrigal Diaz, Jin Xu, Victor Rühle, and Saravan Rajmohan. OdysseyBench: Evaluating LLM agents on long-horizon complex office application workflows, 2025. URL https://arxiv.org/abs/2508.09124.

Zilong Wang, Yuedong Cui, Li Zhong, Zimin Zhang, Da Yin, Bill Yuchen Lin, and Jingbo Shang. OfficeBench: Benchmarking language agents across multiple applications for office automation, 2024. URL https://arxiv.org/abs/2407.19056.

Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. BugsInPy: A database of existing bugs in Python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020. doi: 10.1145/3368089.3417943. URL https://doi.org/10.1145/3368089.3417943.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Jing Hua Toh, 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*, 2024. URL https://arxiv.org/abs/2404.07972.

Yiheng Xu, Zekun Wang, Junli Wang, Dunjie Lu, Tianbao Xie, Amrita Saha, Doyen Sahoo, Tao Yu, and Caiming Xiong. Aguvis: Unified pure vision agents for autonomous GUI interaction, 2024. URL https://arxiv.org/abs/2412.04454.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. ProgramBench: Can language models rebuild programs from scratch?, 2026. URL https://arxiv.org/abs/2605.03546.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. WebShop: Towards scalable real-world web interaction with grounded language agents. In *Advances in Neural Information Processing Systems*, 2022. URL https://arxiv.org/abs/2207.01206.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains, 2024. URL https://arxiv.org/abs/2406.12045.

Ori Yoran, Samuel Joseph Amouyal, Chaitanya Malaviya, Ben Bogin, Ofir Press, and Jonathan Berant. AssistantBench: Can web agents solve realistic and time-consuming tasks? In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2024. URL https://aclanthology.org/2024.emnlp-main.505/.

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*, 2024. URL https://arxiv.org/abs/2307.13854.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Yifan Liu, Muhammad Asaduzzaman, Zhenchang Xing, Xin Xia, and David Lo. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions, 2024. URL https://arxiv.org/abs/2406.15877.

A Dataset Details

This appendix records the dataset statistics summarized in the main text. The generator-visible case fields are intention and context; the app-domain and assertion fields below are evaluator-side metadata.

Statistic	Value
Cases	500
Single-app domains	9
single_app / multi_tool / shared_world	302 / 130 / 68
Episodes	2386
Deterministic assertions	10480
Dev / public / sealed	147 / 172 / 181

Table 3: LiveUIBench statistics: case, family, episode, and deterministic-assertion counts with the dev/public/sealed split sizes, generated mechanically from the built case file.

Authoring, review, and audit protocol. Intentions were authored by a multi-agent pipeline (one author per app domain, grounded in the live AppWorld capability catalog) and adversarially reviewed for recurring-workflow shape and concrete seed entities. Assertions were authored by a second pipeline and adversarially repaired so that every grounding value is a concrete entity from the case context — never a generic word — and every named API exists in the domain's catalog. Case ids, split counts, episode shape, and assertion types are validated mechanically, and domain labels are never exposed to the generator. Two live-world audits then sampled cases under a fixed no-repair harness configuration and repaired defects through a tracked override layer rather than by editing cases in place; the second audit asked specifically whether a *correct* app could still fail each case, growing the seedable world fixtures to 367 files and quarantining unsatisfiable cases out of the suite.

Control	Construction	Contract failed by design
Direct answer	Returns text with no surface	Visible task state
Static shell	Renders plausible controls bound to nothing	Action routing
Route-only	Declares agent routes but never writes visible state	Grounded visible update
Hidden state	Persists agent results in backend memory the user cannot see	Grounded visible update, reopen persistence
Schema-first	Emits a well-formed state schema without visible workflow output	Visible task state, grounded visible update
Multi-agent	Coordinates sub-agents but surfaces results partially and loses revision state on reopen	Grounded visible update, reopen persistence

Table 4: The six scripted controls. Each passes some assertion layers cleanly while failing a targeted contract.

Split	Cases
dev	147
public_test	172
sealed_test	181
total	500

Table 5: LiveUIBench split sizes generated from the case file.

App domain	Cases
multi_tool	130
shared_world	68
amazon	42
gmail	42
spotify	34
phone	32
venmo	32
splitwise	32
file_system	31
simple_note	29
todoist	28

Table 6: App-domain distribution. Domain labels support auditing and are not exposed to the generator.

Assertion type	Count
visible_result_grounded	2805
appworld_api_called	2115
appworld_db_changed	1418
route_enters_orchestrator	1254
state_persists	1187
input_available	567
action_available	567
route_exists	567
total	10480

Table 7: Assertion-type counts. Every assertion is checked deterministically against execution artifacts.

Family	Evaluated object	GUI eval	Behavior eval	Success
Code repair	Patch / code	–	Tests	Resolved issue
ProgramBench	Rebuilt program	–	Behavioral probes	Full program pass
GUI agents	Existing software	–	Action episodes	Task completion
AppWorld	Existing app world	–	API/GUI episodes	State-verified
ALE / exams	Work product	–	Outcome checks	Full pass
Generative interfaces	Generated UI	Preference	–	Human preference
WebGen-Bench	Generated website	–	LLM agent tests	Per-test accuracy
AUI-Gym	Generated agent UI	–	CUA judge	Task solvability
LIVEUIBENCH	Workflow surface	Admission gate	Episode sequence	FWR

Table 8: Positioning of LIVEUIBENCH against adjacent evaluation families: the agent creates the environment that future user interactions will operate.

B EVALUATOR ARCHITECTURE DETAILS

The main text reports all scores from the deterministic runner and uses the online behavior evaluator only as validation (§5). This appendix records the architecture behind that validation. Figure 4 shows the two evaluator endpoints over one generated surface. The GUI evaluator's admission rules cover semantic input slots, visible result blocks, action bindings, readable text, and cross-platform readiness; blocking findings prevent admission. The behavior evaluator's sessions run against a freshly reset world seeded per case; fixture identities are given to the generator and driver, and transient provider faults are retried but never scored. Both write artifact-backed reports, and neither mutates the package. Figure 5 shows how one orchestration loop coordinates the two stages — GUI checks before save, behavioral episodes after render — with every finding attributed to a specific layer and handed back to the generator as an optimization signal.

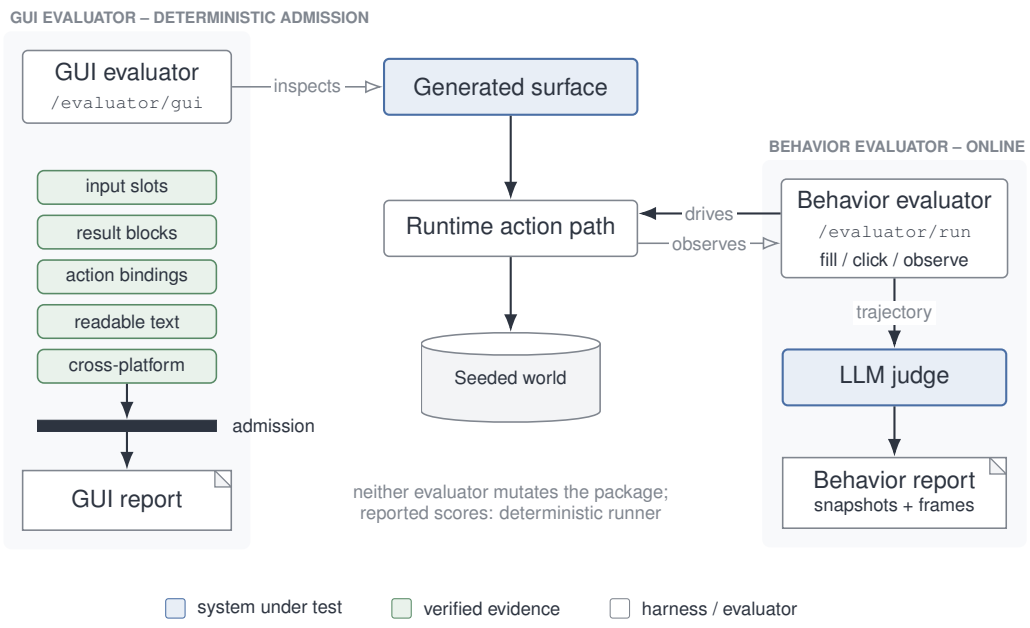

Figure 4: Two evaluators over one generated surface. The GUI evaluator (`/evaluator/gui`) applies deterministic admission rules to the package and its rendered surface model — semantic input slots, visible result blocks, action bindings, readable text, and cross-platform readiness — and blocking findings withhold admission. The behavior evaluator (`/evaluator/run`) drives long-horizon exploratory sessions online, one user-level fill/click/observe action at a time through the same runtime action path against a freshly reset, per-case seeded world, then reaches an LLM judgment over the full trajectory and archives step-by-step snapshots and rendered frames. Both write artifact-backed reports and neither mutates the package; all reported scores come from the deterministic runner, while the behavior evaluator verdicts from the user's side of the screen.

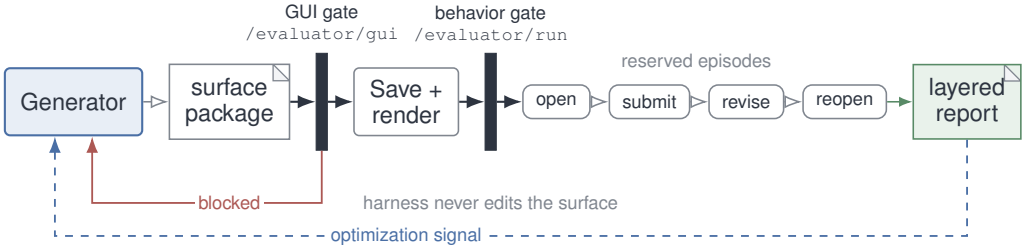

Figure 5: Evaluation orchestration. One loop coordinates the two evaluation stages: the GUI gate (`/evaluator/gui`) runs deterministic admission checks on the surface package before save, and the behavior gate (`/evaluator/run`) drives the reserved open-submit-revise-reopen episodes after render. Episode evidence compiles into a layer-attributed report; blocking admission findings and layer-attributed diagnostics flow back to the generator as an optimization signal. The harness never edits the surface, so "could not render a plausible surface" stays distinguishable from "rendered a surface that does not behave."

C GUIOS Runtime Details

This appendix records the mechanism inventory behind the design principles of §6.

Spec package layout. Each generated app is serialized as `manifest.json` (app identity), `ui.json` (the A2UI surface description), `actions.json` (control-to-workflow bindings), `data.schema.json` (app-scoped state), `agents.json` (the app's backend workflow), and

`assets/` (a sanitized SVG icon). Because the package is declarative and validated, it is diffable, versionable, and rollback-able; a single spec renders across Web, iOS, and Android through one universal shell (Expo + React Native Web) with no per-platform reimplementation.

Gateway endpoints. The Agent Gateway exposes four endpoints: `/agent/run` (turn a whiteboard sentence into an app), `/agent/action` (execute a control's workflow), `/agent/patch-app` (apply a natural-language modification as a spec patch), and `/agent/query-state` (report whiteboard-level state). The front end is a renderer and a router; all behavior lives behind the gateway.

Pipeline stages. A *Router Agent* classifies each whiteboard sentence as `create_app`, `modify_app`, `answer_question`, or `remove_app`. For app creation, *Requirement* and *Business-Slot* agents synthesize the app name and tagline, the semantic input slots the user will fill, and the lifecycle action buttons, and they select the target AppWorld services the workflow must drive. A *UI Designer Agent* lays out the surface from the fixed A2UI vocabulary — `Card`, `Column`, `Row`, `Text`, `TextField`, `Button`, `List` — under a strict binding whitelist. Downstream *Data*, *Asset*, *Safety*, and *Layout* agents produce the state schema, a sanitized SVG icon (drawn by an Icon agent under a strict tag whitelist), high-risk-action interception, and cross-platform layout.

Validation gate. The designed surface must pass validation before it is admitted. Writing the surface as a graph $G = (V, E)$ whose vertices are components and whose edges are binding references, admission requires: (i) every $v \in V$ comes from the A2UI vocabulary and every $e \in E$ satisfies the binding whitelist; (ii) G is acyclic; (iii) every component is referenced exactly once (the reference relation restricted to components is a bijection onto V minus the root); and (iv) V contains a result `Text`, an input control, and an entities `List`. The package also forbids unsafe components outright — no `script`, `iframe`, `raw_html`, `custom_js`, `native_module`, or `shell_command` — and every action must bind to a backend workflow with declared capability permissions. This is runtime-side enforcement of the *visible task state* and *action routing* contracts before any surface reaches a user, exactly the properties the harness's GUI admission checks gate on from the outside.

Write Verifier rules. Every write batch passes a *Write Verifier Agent* that checks amount traceability, verbatim entity names, replace-not-append semantics, one object per call, and that no UI-component id leaks into a real object name. The verifier may only edit a call's arguments; it may never swap the API the workflow selected.

Honest-failure gate. An action wired to AppWorld permissions that makes zero real calls returns an explicit failure (HTTP 502) rather than fabricating a local success; the runtime never falls back to inventing content.

Entity-first, per-turn visible state. The executor returns `entities[{name, id, detail}]` and per-turn `stats[]`. Entities render as "current content" cards and are wholesale-replaced each turn rather than appended, with derived-state grouping (by entity-group prefix) and derived counters, so the visible result always reflects the latest grounded database state rather than an accumulating transcript.

Pre-registered runtime ablations. The measurement plan of §7 ablates only the grounding executor. Two further ablations are scoped for follow-up runs rather than claimed here: disabling the Write Verifier (does argument-level verification measurably reduce grounding errors?) and capping the refine loop at one shot (does deficit-driven retrying add over a single grounded attempt?). Their absence from the current plan is a scoping decision, not an oversight.

D Release Governance

The benchmark release separates public development from sealed evaluation: the development and public-test splits ship with cases, fixtures, and the full harness to support debugging and reproducibility, while the sealed-test split is withheld and sealed-split claims require hosted execution with archived reports. Repository-side release gates (schema validation, leakage checks against the sealed

split, report-format checks, runtime smoke tests, and the paper-readiness gate of Appendix E) are enforced in CI and documented with the release.

E RUN-ARTIFACT PROTOCOL

Every reported run must satisfy the following protocol before its numbers may appear in the main text. (1) The run writes `report.json` and `run-summary.json` under a unique artifact directory, logging model identity, API base, command, split, case count, latency, temperature, token budget, and cost. (2) Every table in the paper is generated from those artifacts by checked-in scripts, never edited by hand. (3) A claim-traceability ledger maps each main-text claim to its source artifact, and a run manifest lists every run with its command so controls and diagnostics can be rerun through the same adapter boundary. (4) Sealed-split numbers additionally require hosted execution with archived reports. Runtime configuration: the harness resolves the LLM configuration per request, streams per-step progress over SSE, retries only transient transport faults (never scored), enforces a bounded orchestrator timeout, and resets AppWorld to a pristine world before every run; the executor performs writes first and verifies by read. The ledger and manifest formats are checked automatically by the repository's paper-readiness gate.

F EXAMPLE BEHAVIOR TRAJECTORIES

The online behavior evaluator archives a per-step snapshot of the rendered surface (fields with live values, action buttons, and the executed operation), which the harness renders into trajectory frames. Figures 6 and 7 show two archived trajectories. In the first, a fitness check-in surface completes a submit–revise sequence: each click routes into the app-scoped workflow and the visible plan updates. In the second, a purchase-review backlog surface accepts input and routes correctly, but the trajectory judgment fails the session because the visible state never reflects the real Amazon order data the intention requires — the same real-API grounding gap the deterministic assertions localize.

Figure 8 isolates the grounding bottleneck by contrasting two apps generated from unrelated cross-domain intentions. On the left, an Amazon review-backlog surface routes a submit into the app-scoped workflow, which selects and invokes the real `create_product_review` API; the returned entity (*"Created a 5-star review … with review id 15968"*) is written back into the visible result block, so the state the user sees is grounded in a real database write. On the right, a Venmo recurring-rent surface exposes equally well-structured semantic slots (per-tenant rooms, amounts, paid/unpaid status, late fee), but the executed action degenerates to echoing the app title rather than issuing the real per-tenant payment requests the intention names — no grounded entity ever reaches the surface. The two frames share the same generated-app machinery; the passing workflow differs from the failing one only in whether the backend agent completed real, visible API grounding — precisely the property LIVEUIBENCH's grounding contracts measure.

Figure 6: Trajectory frames from a passing episode on a fitness check-in surface (fill → first submit → revision submit): each submit routes into the app-scoped workflow and the visible plan updates. The banner records the executed operation and its outcome; the card shows the live field values the user sees at that step.

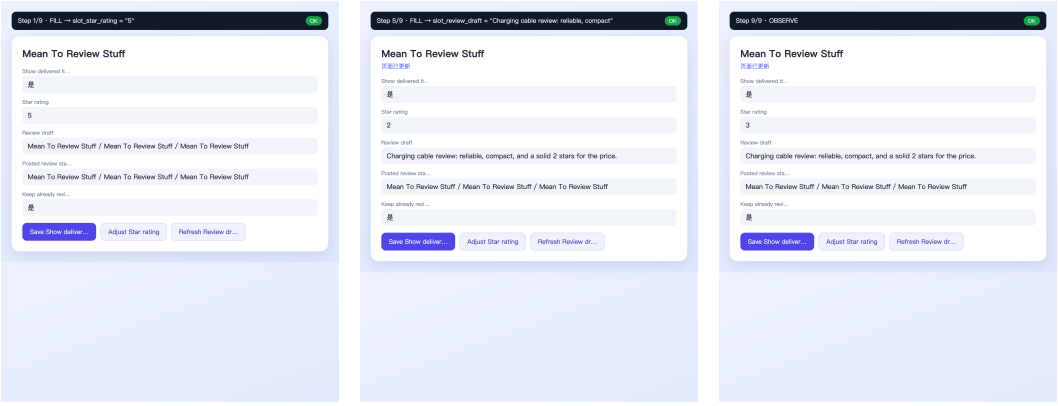

Figure 7: Trajectory frames from a purchase-review backlog surface. Controls fill and route correctly, but the visible state never surfaces the real order entities the intention names, so the episode fails — the behavior-level view of the real-API-grounding bottleneck.

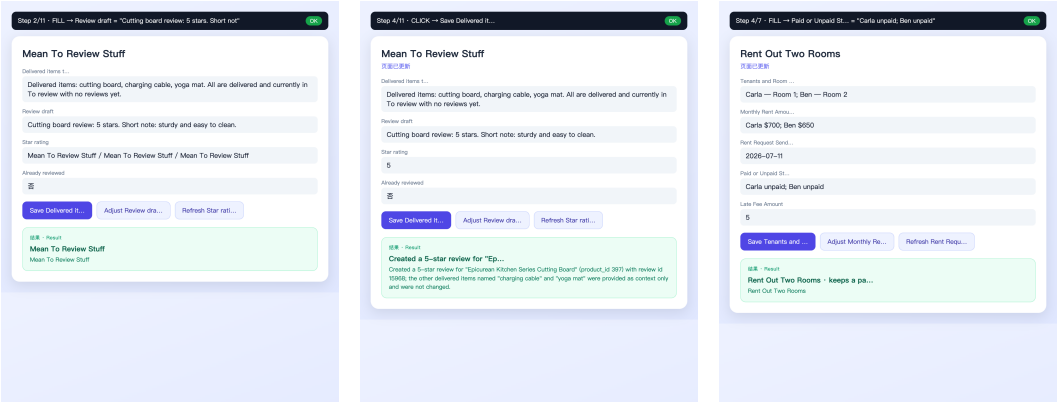

Figure 8: Cross-domain grounding contrast. *Left/center:* an Amazon review-backlog surface opens, then a submit routes into the app-scoped workflow, which invokes the real `create_product_review` API and writes the returned review entity (*review id 15968*) into the visible result block — a grounded, passing step. *Right:* a Venmo recurring-rent surface with equally structured semantic slots, whose action degenerates to echoing the app title instead of issuing real per-tenant payment requests, so no grounded entity reaches the surface and the episode fails. Same generated-app machinery; real-API grounding is the discriminator.

G Case Studies from the Behavioral Audit

The two-audit trail (Section 4) was driven by repeated live behavioral sampling over the audited suite under a fixed no-repair harness configuration. This appendix reconstructs five representative sessions at the API-call level from their archived run artifacts (per-step surface snapshots, embedded AppWorld request/response payloads, and the trajectory judgment). Together they illustrate the two claims the main text makes about the audited suite: repaired cases attribute failures cleanly to model capability rather than to missing world data, and the dominant open failure family — multi-party ledger state discipline — decomposes into distinct, reproducible arithmetic-encoding errors rather than one monolithic weakness. Each session's pass/fail verdict below is the archived trajectory judgment; these are audit-time observations, not benchmark scores.

G.1 Divisor anchoring: the payer drops out of the denominator

Case `splitwise-camping-gear-food`: five campers split gear rentals and a food haul. The generated app created the group correctly (five member emails) and then recorded the $125 food haul with the call

```
record_expense({ group_id: 274, description: "food haul",
  amount: 125, paid_amount: 125, payer_email: "oracle@guios.dev",
  debtor_emails: [ivan@, rue@, hal@, zoe@] })   % payer omitted
```

Splitwise divides `paid_amount` across `debtor_emails`; omitting the payer yields four debtors at $31.25 instead of five $25 shares. The surface then *reported its own error* — the visible summary stated "requested equal split: $25.00 per person; currently recorded non-payer share: $31.25 per person" — yet no correcting call followed. Notably, the *same session's second expense* avoided the bug by passing explicit `debt_amounts: [22.5, 22.5, 22.5, 22.5]`: the model can encode the arithmetic correctly, but does not do so consistently — the instability the main text's ledger taxonomy describes. The episode also left the long-horizon story incomplete (of the two required settle-ups, only one `record_payment` was issued). Archived trajectory judgment: fail.

G.2 Wrong service, plausible surface: nine obligations never created

Case `splitwise-fantasy-league-pot`: a ten-person league buy-in tracker; nine manager identities are seeded into the world by the case fixture. The generated surface was well-formed

— league name, ten-manager roster, $25 buy-in, a "Collect buy-ins" action — but the workflow bound to the *wrong backend*: all 64 real calls in the session are Venmo reads (`search_users`, `show_transactions`, ...); not one Splitwise call was issued, and no obligation-creating write of any kind occurred. Because the seeded managers exist in Splitwise, the Venmo searches returned strangers, and the app truthfully reported it "could not create the buy-in requests ... because those recipient accounts do not exist." After three "Collect buy-ins" clicks, the final surface still showed zero transactions while claiming "17 real operations completed." This is the scale-amplified end of the ledger family: the more counterparties a case requires, the more a single early binding error compounds into a fully unresolved workflow. Archived trajectory judgment: fail.

G.3 Duplicate-write inflation: a rebalance that raises the balance

Case `splitwise-thanksgiving-potluck`: the host logs each relative's dish spending; when the uncle later contributes a $60 pie-and-wine purchase, his owed amount should shrink. The archived call sequence shows the app instead *re-recorded the $140 turkey expense four additional times* (expense ids 5040–5043, 5045) while attempting split edits, then correctly recorded the uncle's $60 contribution (id 5044) — whose effect was swamped by the duplicates. The uncle's visible balance moved $35 → $163.34 → $183.34, the opposite direction of the expected rebalance, and the harness's per-step snapshots show the surface presenting each inflated state as a success. One episode step also failed outright (a fill targeted an "Add expense" control that was never rendered). This is the same state-discipline failure as Appendix G.1 in a different encoding: not a wrong divisor but a non-idempotent update loop. Archived trajectory judgment: fail.

G.4 Repaired-case attribution: data present, aggregation still fails

Case `gmail-newsletter-triage-inbox` (post-repair): the audit added a world fixture seeding nine unread newsletter threads from three senders, so a failure can no longer be blamed on an empty world. In the archived session the app called the inbox-listing API eight times — *every time with an empty request*, no query and no pagination — so every call returned the same default first page of five threads, and only one thread was ever surfaced, labeled, and reported ("I found and labeled 1 unread newsletter thread..."). The nine-thread sweep, the two-keeper selection, and the archive step were structurally unreachable from that first page. This is the attribution pattern the repaired benchmark is designed to expose: the world verifiably contains the data (the per-step snapshot itself reads "inbox threads: 9"), the surface renders and routes, and the failure localizes precisely to multi-item aggregation — a model-capability gap, not a benchmark defect. Archived trajectory judgment: fail.

G.5 A passing session under the same configuration

Case `gmail-family-forwarding-hub` (post-repair, same harness configuration): the fixture seeds three inbox threads and two relatives. The session forwarded the appointment thread to the mother (`forward_email_thread` → sent thread 47830), revised the recipient and forwarded the reunion thread to the uncle (sent thread 47832), and logged the routing decision as a real note (`create_note` → id 3084); after teardown and reopen, both forwarded records and the routing note were still visible, and the judge separately confirmed the third (never-routed) thread remained untouched in the inbox — the app did not over-complete. Archived trajectory judgment: pass. Contrasted with Appendices G.1–G.4, this session shows the audited suite discriminating within one model and one configuration: multi-write workflows with per-entity state discipline fail, while read-route-write workflows over seeded entities complete end-to-end — evidence that the repaired cases are satisfiable and that the remaining failures measure capability.

H Example Case and Episode Schema

In the weekly commute-playlist case, the intention asks for an app that builds a fresh commute playlist each week and remembers last week's choices. The open episode checks that input controls and an executable route exist. The submit episode ("vibe upbeat indie, seed artists Phoenix... for *Monday Commute*") asserts that the event enters the orchestrator, that `create_playlist` is invoked, that the `spotify` database changes, and that the visible result names *Monday Commute*. The revise

episode adds “no songs over 4 minutes” and asserts a further database change and an updated visible result. The reopen episode asserts *Monday Commute* is still visible after teardown.

The following schema-only example illustrates the episode format; it is not a sealed-test case.

```
{
  "id": "spotify-weekly-commute-playlist__submit",
  "kind": "submit",
  "userMessage": "Vibe is upbeat indie, seed artists Phoenix and
      Two Door Cinema Club, about 20 songs for Monday Commute.",
  "assertions": [
    { "type": "route_enters_orchestrator", "value": true },
    { "type": "appworld_api_called", "value": "create_playlist" },
    { "type": "appworld_db_changed", "value": "spotify" },
    { "type": "visible_result_grounded", "value": "Monday Commute" }
  ]
}
```