WHEN INTERFACES SPEAK: DATA-AWARE GENERATIVE UI HARNESS FOR ACTIVE INTERACTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Most human-agent interaction today remains text-based. However, natural language is often not a perfect medium for complex tasks, as it can lead to cognitive overload, ambiguity, information chaos, and a slow input process; ephemeral generative UIs can present structured information and guide users toward task completion. In this work, we propose **GENUI-Harness**, a multi-agent harness that pairs a Tool Agent for information retrieval and task execution with a GUI Coder Agent that identifies ambiguities and generates front-end code for structured interfaces, helping users complete their tasks. Training such a versatile coder with reinforcement learning is challenging: verifiable rewards for interactive UI generation require costly execution, while LLM-as-a-Judge rewards are prone to reward hacking. We address the first challenge with **Dynamic UX**, a lightweight package that supports dynamic interaction and reward collection within a single sandbox, and the second with **Reward Auditor**, a meta-reward mechanism that monitors reward distributions and automatically distills diagnostic patterns into a shared rubric and scoring specification. To evaluate this task, we introduce **UI-TAU Bench**, a benchmark for active human-agent interaction through generated UI code, built on **10** real-world domain databases constructed from public data sources and based on Tau-Bench tool use settings, with Lite (**300 tasks**) and Full (**1,000 tasks**) splits. Experiments show that GENUI-Harness achieves an average Pass@3 gain of **4.48 percentage points** over smolagents on Lite. Training with GENUI-Harness improves a 4B backbone from a **9.33%** baseline to **58.00%** Pass@3, even outperforming larger frontier models such as Claude Opus 5 (**46.67%**). We further show that GenUI-Harness remains robust not only on ambiguous queries, but also on non-ambiguous ones. In a reviewer survey comparing communication channels, generated UIs reduce the average number of dialogue rounds from **3.4** to **1.2**. These results show that data-aware generative interfaces can support effective task completion and reduce dialogue rounds in the evaluated database-backed workflows.

1 INTRODUCTION

Most current human-agent systems still interact through text (Yao et al., 2023; Schick et al., 2023; Li et al., 2023; Qin et al., 2024; Yao et al., 2025). While natural language is flexible, it is often not a perfect medium for complex, goal-directed tasks: users face cognitive overload when translating structured needs into text, ambiguity from underspecified requirements, information chaos from scattered preferences and database records, and a slow input process when typing complex instructions (Larkin & Simon, 1987; Rao & Daumé III, 2018; Aliannejadi et al., 2019; Zamani et al., 2020). These limitations reflect a deeper mismatch between medium and task in interactive agent settings (Yao et al., 2022; Deng et al., 2023; Zhou et al., 2024; Koh et al., 2024; Drouin et al., 2024; Xie et al., 2024; Trivedi et al., 2024; Liu et al., 2024): progress depends not only on stronger language models but also on better interaction media. Figure 1 illustrates this problem: booking a family trip from Paris to São Paulo requires comparing retrieved flights while specifying travel dates, cabin class, and baggage. In a text exchange, the user must track flight details and answer several preference questions in prose; a generated UI presents flight options alongside preference controls for direct comparison and input.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

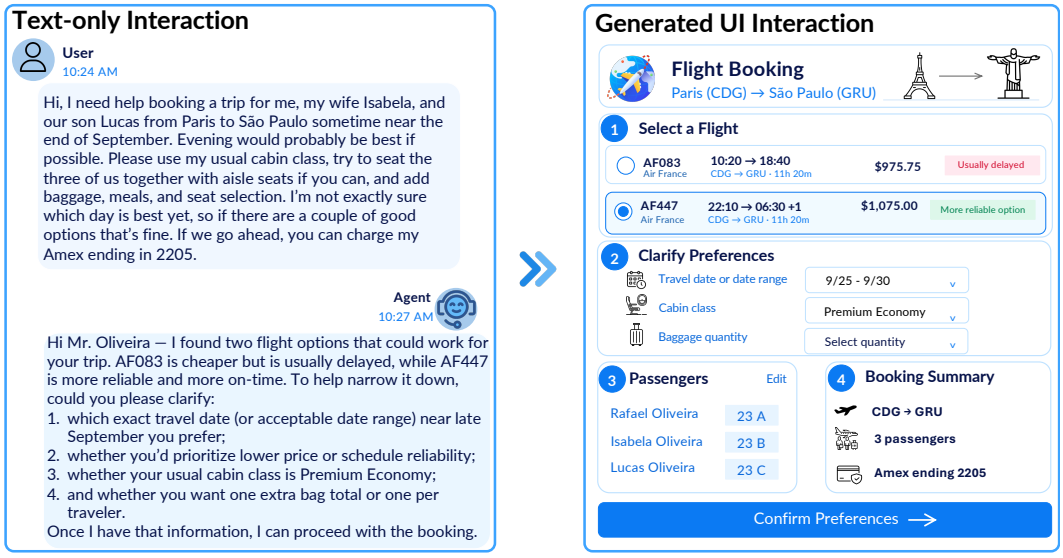

Figure 1: Text-based and generated-UI interaction for the same flight-booking request. (a) Text requires successive clarification of travel preferences. (b) The generated UI presents retrieved options and structured controls for direct review and clarification.

In this paper, we study *data-aware generative UI* as an alternative interaction paradigm. Building on personalized, task-specific, and dynamically synthesized interfaces (Gajos et al., 2010; Vaithilingam & Guo, 2019; Vaithilingam et al., 2024; Nandy et al., 2024; Cao et al., 2025; Chen et al., 2026), an *adaptive and ephemeral* interface provides a communication channel tailored to the current request and relevant database records. It presents structured information and supports direct input, following the principles of direct manipulation and focused clarification (Larkin & Simon, 1987; Shneiderman, 1983; Hutchins et al., 1985; Horvitz, 1999; Xu et al., 2019; Aliannejadi et al., 2021; Zamani et al., 2020). To realize this idea, we propose **GenUI-Harness**, a multi-agent harness with a Tool Agent operating in two stages and a GUI Coder Agent between them. During *pre-agent exploration*, the Tool Agent gathers relevant records and candidate options through read-only tool calls. The GUI Coder Agent uses this evidence to identify ambiguities in task requirements and generates executable front-end TypeScript code that presents the retrieved options and collects the required user input. During *post-agent execution*, the Tool Agent combines the submitted data with the retrieved evidence, obtains any remaining database-dependent information, and executes the requested action through tool calling.

The GUI Coder Agent must identify ambiguous task requirements, express them through appropriate interface controls, and generate code that renders correctly and submits useful data for downstream execution (Wu et al., 2024; Chen et al., 2023). We train the coder with reinforcement learning, for which reward design poses two challenges. First, collecting execution-based rewards requires rendering generated code at scale, with isolated sessions so that one rollout does not affect another. We introduce **Dynamic UX**, a lightweight package for parallel, session-isolated execution and reward collection within a single sandbox. Second, rendering alone does not establish whether an interface exposes the relevant ambiguities or enables successful downstream action. Verifying these properties requires user interaction and task execution, while a less expensive LLM-as-a-Judge reward may assign high scores to interfaces that fail during use and is susceptible to reward hacking (Zheng et al., 2023; Tan et al., 2025; Gao et al., 2023; Skalse et al., 2022). We introduce **Reward Auditor**, a meta-reward mechanism that uses actual task-execution outcomes to diagnose and automatically revise the training reward. It monitors reward distributions and examines individual rollouts to identify failures that receive high rewards and successful interactions that the criteria undervalue.

To evaluate this task, we introduce **UI-Tau Bench**, a benchmark based on the tool-use settings of Tau-Bench (Yao et al., 2025) that evaluates an agent's ability to generate task-specific UIs on the fly and use them for active human-agent interaction. It grounds tasks in **10 real-world domain databases** constructed from public data sources. Its Lite (300 tasks) and Full (1,000 tasks) splits evaluate whether the generated interface supports the user input and downstream action needed to reach the target database state. On Lite, GenUI-Harness attains higher Pass@3 than `smolagents`,

`mini-swe-agent`, and `Pi` on 8, 7, and 8 of the 9 shared backbones, respectively. Training a 4B backbone within our harness increases Pass@3 from 9.33% to 58.00%, even outperforming substantially larger frontier models such as Claude Opus 5 (46.67%). These gains are not limited to ambiguous user requests. We also find that GENUI-Harness remains robust when user goals are already clearly specified, suggesting that generated interfaces are useful not only for clarification, but also for efficient task progression. In a reviewer survey comparing communication channels, generated UIs reduce average dialogue rounds from 3.4 to 1.2, showing that structured submissions can consolidate multiple required inputs into a single exchange.

Our contributions are threefold: (1) the task of *data-aware generative UI* for stateful service tasks; (2) **UI-TAU Bench**, which evaluates active human-agent interaction through generated front-end code; and (3) **GENUI-Harness**, which combines tool grounding, role-specialized UI generation, Dynamic UX, and Reward Auditor. Experiments show strong database-state success for a compact model and GUI-Coder transfer across models.

2 TASK DEFINITION

The data-aware generative UI task requires an agent to construct an executable interface from a user request and database evidence, collect the user input needed for the task, and complete the requested action through tool use. Each instance contains a user request, an initial database state, query and action tools, and domain rules, following the tool-use setting of Tau-Bench (Yao et al., 2025). The agent uses read-only tools to retrieve relevant records from the database, generates a task-specific UI on the fly to obtain missing user information or confirm the intended action, and calls an action tool to update the database while satisfying the user's requirements and the applicable rules. Appendix A.1 gives the formal definition.

3 UI-TAU BENCH

UI-TAU Bench evaluates whether an agent can generate a useful interface on the fly from the current request and database evidence, collect user input, and complete the task through tool use.

Domain Data Collection. The benchmark uses 10 real-world domain databases constructed from public data sources, covering flight booking, hotel booking, e-commerce, and retail (OpenFlights, n.d.; António et al., 2019; Olist, 2018; Chen, 2012). These sources provide domain records in heterogeneous schemas, rather than task environments with a common query interface and defined state transitions. Four doctoral researchers collect and normalize these sources by standardizing field types and identifiers, checking cross-table references, and organizing each domain into a user-facing entity table, resource tables, and a transaction table. Query tools can then retrieve related records, and action tools can apply updates whose effects are checked against a target database state under explicit domain rules. Appendix C documents the data sources and construction details.

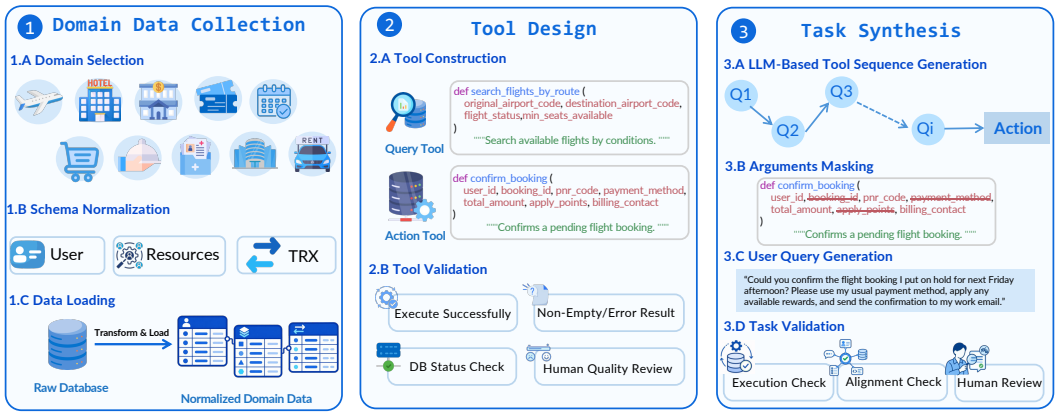

Figure 2: UI-TAU Bench construction pipeline. We collect and normalize the domain databases, construct and validate the query and action tools, and synthesize and validate the benchmark tasks.

Tool Design. Working from the normalized databases, the same annotators first specify the domain rules that govern action requirements, eligibility conditions, and valid state transitions, then define the corresponding query and action tools (Li et al., 2023; Qin et al., 2024). Query tools are atomic, read-only retrieval operations that compose into search chains. Action tools apply rule-compliant updates whose arguments are grounded in query observations or user clarification. Validation applies a separate condition to each tool type. A query tool must execute without error, return a non-empty result, and return records consistent with the arguments it was given. An action tool is invoked with arguments taken from query observations or user clarification and must both satisfy the domain rules and produce the intended database state. A tool that violates any condition is revised against the case that failed and revalidated at most twice. Every surviving tool is then cross-checked by a second annotator. We retain a tool when both annotators agree that it is correct, and discard tools that still fail or whose behavior remains ambiguous. Appendix C.1 details tool construction and validation.

Task Synthesis. We design a task synthesis agent using `Claude Opus 5`. For each candidate, one of these annotators samples a random user and specifies a task goal, such as changing an existing flight booking. Given the selected user's records, the domain database, and the available tools, the agent proposes and executes a query chain for that goal, with later calls using earlier results, and constructs a reference action whose arguments come from retrieved records. We then mask all internal identifiers except the user identifier, together with 2–3 additional arguments for moderate-ambiguity tasks or 4–5 for high-ambiguity tasks. Conditioned on the executed chain, retrieved records, and domain rules, the agent writes a natural-language request that leaves these arguments unspecified. The hidden reference action preserves the complete goal, while the visible request requires the evaluated agent to recover missing information through database exploration, user interaction, or both. Each candidate passes three checks: execution verifies the reference action's database update; alignment checks establish consistency among the request, user profile, records, rules, and reference action; and two independent annotators assess plausibility, fluency, and masked-value leakage. Disagreements are resolved by a third annotator, and unresolved candidates are discarded. Appendix C.1 gives the full protocol.

Table 1: Data statistics

STATISTIC	LITE	FULL
Total instances	**300**	**1,000**
Moderate-ambiguity instances	161	523
High-ambiguity instances	139	477
Unique query tools	303	304
Unique action tools	162	331
Avg. tokens / user task	90.3	90.2
Avg. masked args. / action	4.9	4.8
Inter-annotator agreement (%)	94.1	94.3

Data Statistics and Evaluation Metrics. Table 1 shows the UI-TAU Bench statistics. Appendix C.2 reports per-domain statistics and comparisons with existing benchmarks. We run each task three times and report state exact-match (EM) metrics over N tasks:

$$\text{Pass@3} = \frac{100}{N}\sum_{i=1}^{N}\max_{j\in\{1,2,3\}} z_{i,j}, \qquad \text{Avg@3} = \frac{100}{3N}\sum_{i=1}^{N}\sum_{j=1}^{3} z_{i,j}.$$

Here, $z_{i,j} = 1$ if run j executes successfully and produces exactly the reference database state, and 0 otherwise. Pass@3 measures success in at least one run; Avg@3 averages success across all three.

4 GENUI-HARNESS

Solving each task in UI-TAU Bench requires the agent system to understand the query, explore the database, and interact with the user. GENUI-Harness therefore assigns these requirements to separate roles. A Tool Agent retrieves records and executes the final action, and a GUI Coder Agent turns the retrieved evidence and the unresolved arguments into an interface the user operates directly, so that choices are made among grounded options without exposing database schemas or internal identifiers. Separating the roles also makes the interface component independently trainable, which allows the GUI Coder Agent to be optimized against execution outcomes while the Tool Agent is held fixed, and allows the trained coder to be paired with a different tool-reasoning model. Figure 3 summarizes the workflow, and Appendix A.2 gives the detailed protocols.

Pre-Agent Exploration. Given the query and candidate action schemas, the Tool Agent identifies the intended operation and uses read-only tools in a ReAct loop (Yao et al., 2023; Schick et al., 2023;

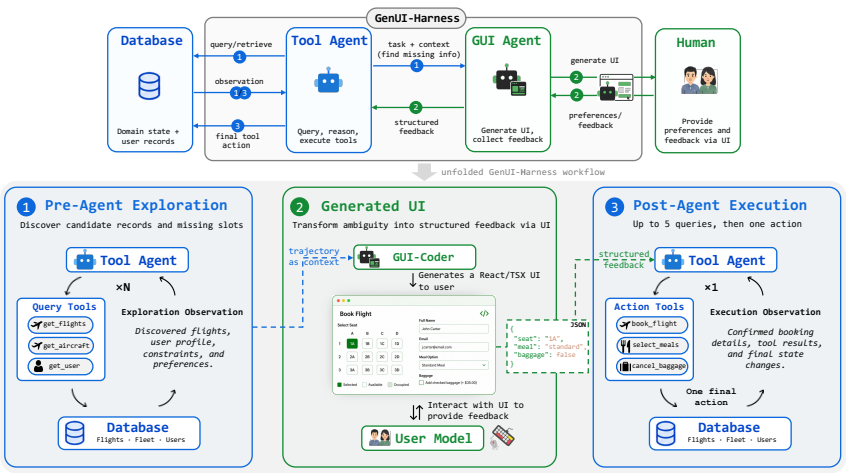

Figure 3: GENUI-Harness links three stages: the Tool Agent retrieves task context, the GUI Coder Agent generates an interface to collect user input, and the Tool Agent uses that input to execute the final database action.

Li et al., 2023) (up to 5 turns) to retrieve records and the user profile. This grounds known arguments and collects candidate options for unresolved user choices. The Tool Agent passes the query and the exploration history, which records its tool calls and their observations, to the GUI Coder Agent.

Generated UI. The GUI Coder Agent reads the query and the exploration history and determines which arguments the retrieved records already fix, which candidate values are available for the remaining ones, and which arguments only the user can decide. For each argument left to the user it selects an interface control suited to that decision, such as a list for choosing among retrieved candidates or an input field for a value the user supplies (Horvitz, 1999; Vaithilingam et al., 2024; Ma et al., 2025). It emits a single Material UI component in React and TypeScript that presents the candidates with readable labels and the attributes a decision depends on. Arguments already resolved are shown as context or prefilled, so the user attends only to what is unresolved. The user simulator operates the rendered interface according to its assigned intent and preferences, and its submission is returned to the Tool Agent as a JSON object whose keys name action arguments and whose values record the user inputs.

Post-Agent Execution. The Tool Agent resumes with the query, exploration history, and submitted user inputs. It can use up to 5 further query turns to validate selections or recover remaining database-dependent arguments. It combines these observations with the user's input to complete the action arguments, then executes the action tool to update the database.

5 TRAINING

We construct training environments from 40 databases synthesized with `Claude Sonnet 4.6` (Anthropic, 2026b), all separate from UI-TAU Bench. On all 40 databases, `DeepSeek V4 Flash` (Xu et al., 2026) generates end-to-end trajectories within GENUI-Harness. We retain successful trajectories whose final database state matches the target and whose interfaces render with valid ARIA snapshots. Trajectories from 25 databases are used to fine-tune the Tool Agent and GUI Coder Agent separately with supervised fine-tuning (SFT). The remaining 15 databases define the RL environments; retained trajectories from these databases supply reference UI code and ARIA snapshots for reward evaluation. Appendix D details construction, filtering, and split verification.

Both agents start from `Qwen3.5-4B` and are fine-tuned separately on their respective training examples, which fixes the trajectory structure and the output format each agent must produce. We call the resulting pair of checkpoints GENUI-4B (SFT). Holding the Tool Agent at its SFT checkpoint, we then train only the GUI Coder Agent on the 15 RL databases with Group Relative Policy Optimization

(GRPO) (Shao et al., 2024), using the audited reward to optimize it toward interfaces that lead to successful task completion (Le et al., 2022; Wu et al., 2024).

5.1 Dynamic UX

Static code checks cannot establish whether a generated UI renders successfully or what controls it exposes. The component must be transpiled, loaded, and executed in a browser to obtain the interface evidence used for reward evaluation. Repeating this process for every rollout adds rendering time before policy updates. A straightforward isolation strategy launches a fresh browser or sandbox per rollout, duplicating browser processes and, with separate sandboxes, servers and dependencies as concurrency grows. Dynamic UX instead shares one rendering runtime and browser process within a single sandbox while assigning each rollout an isolated browser context, so execution results and rewards cannot be affected by other samples.

Dynamic UX initializes the web server, socket router, browser, and component library once, then assigns each rollout a unique session and isolated browser context. It returns execution status and an initial ARIA snapshot for reward collection, while session-based routing prevents state from crossing samples. To evaluate rendering efficiency and scalability, we benchmark Dynamic UX on 1,100 pre-verified components against fresh-process rendering across five concurrency levels. Results show that context-level isolation is consistently more efficient: on average, Dynamic UX increases throughput by 18.2% while reducing median latency by 15.3%, peak resident set size (RSS) by 35.2%, and Chromium process count by 59.4%. A 1,000-session marker test completes without timeout, failure, or content leakage. Appendix D.7 reports the full setup and results.

5.2 Reward Auditor

Successful rendering does not establish that an interface enables task completion. Verifying the latter requires user interaction, downstream action execution, and a check of the resulting database state, making full evaluation costly for every training rollout. A language-model judge provides less expensive feedback, but may reward plausible-looking interfaces that fail during use or undervalue useful alternatives (Zheng et al., 2023; Kim et al., 2024; Lambert et al., 2025; Tan et al., 2025; Gao et al., 2023). Reward Auditor addresses this gap with a meta-reward mechanism that uses task-execution outcomes to diagnose and iteratively revise the reward that trains the GUI Coder Agent.

We construct the initial reward from task requirements and successful reference trajectories. The judge compares the generated code and rendered ARIA snapshot with the user query, reference code, and reference ARIA. Its rubric operationalizes four requirements: functional equivalence, feature completeness, implementation correctness, and task alignment, with scores $q_i^{\mathrm{fe}}, q_i^{\mathrm{fc}}, q_i^{\mathrm{ic}}, q_i^{\mathrm{ta}} \in [0, 5]$. Let g_i indicate that the rollout passes the required code, rendering, reference-evidence, and judge-response checks. The initial execution-gated reward is

$$j_i = \frac{0.35q_i^{\mathrm{fe}} + 0.35q_i^{\mathrm{fc}} + 0.15q_i^{\mathrm{ic}} + 0.15q_i^{\mathrm{ta}}}{5}, \qquad r_i^{(0)} = g_i j_i.$$

This initial rubric provides an interpretable starting point without requiring exact code agreement with the reference. Its specification appears in Appendix D.8.

This initial reward assigns the same zero score to failures in code extraction, structural validation, and rendering. If every output in a prompt group fails these checks, the rewards provide no within-group contrast for GRPO. Conversely, a rendered interface can receive a high judge score even when its controls cannot submit the user's intended values. Auditing therefore needs to address both the judge's criteria and the mapping from execution evidence to scalar rewards.

Reward Auditor uses `Claude Opus 5` (Anthropic, 2026a) to diagnose and revise rewards from task-execution evidence on internal rollouts from the 15 RL databases (Figure 4). The user simulator operates the interface, and the fixed Tool Agent consumes the submission and executes a terminal action.

The objective label z_i is positive only when execution succeeds and the resulting database state exactly matches the target. For each audit round, Reward Auditor compares this label with the reward prediction $\hat{z}_i = \mathbb{I}[r_i \geq 0.5]$ and examines the resulting cases, reward distributions, judge rationales, and complete trajectories to diagnose failure causes and recurring reward–outcome mismatches.

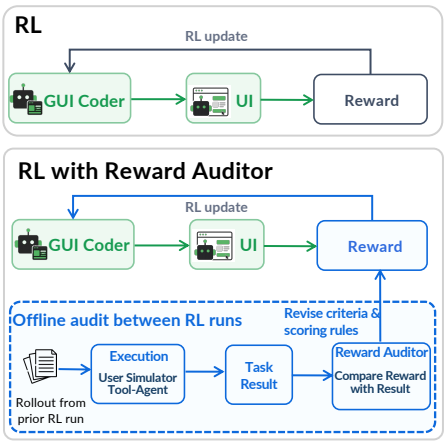

Figure 4: Reward Auditor adds an offline reward-revision loop to RL. Task outcomes on selected internal rollouts guide revisions to judging criteria and scoring rules.

The auditor then modifies the judging criteria and scoring rules and reviews the proposed changes. It replays candidate rewards on stored rollouts and reviews the resulting scores before accepting a revision. Replay holds generated code fixed, making score changes attributable to the reward revision rather than policy behavior. Task-specific diagnoses are consolidated into one shared rubric and scoring specification applied across tasks. The revisions address interaction and submitted-payload requirements, distinguish failure stages, and introduce targeted icon and accessibility checks. We perform two rounds of meta-reward-guided revision, producing the final training reward $r_i^{(2)}$ used for the reported GenUI-4B RL run. In our experiments, Direct RL and both audited variants train independent policies from the same SFT initialization with the Tool Agent fixed.

The final reward separates code-extraction, structural, rendering, and empty-interface failures before applying quality and accessibility scoring. Appendix D.9 gives its complete specification and revision protocol.

6 Experiments

6.1 Setup

We benchmark four agent harnesses on UI-Tau Bench: GenUI-Harness, `smolagents` (Roucher et al., 2025), `mini-swe-agent` (SWE-agent Team, 2025), and `Pi` (Earendil Inc. and Contributors, 2026). All four run the same tasks, databases, query and action tools, user-simulator model, and final-state evaluator, with the same backbone in each row. GenUI-Harness communicates through generated interfaces, whereas the three baselines retain their original text-based interaction. Model and inference details appear in Appendix D.3, and harness adaptations in Appendix D.5. We report Pass@3 and Avg@3 as defined in Section 3.

6.2 Main Results

Overall Performance. Table 2 presents the end-to-end Lite comparison. The three baselines use their native text interaction, providing the text-only control for our generated-UI system while sharing the task environment, tool APIs, user-simulator model, and evaluator; no GUI Coder Agent or browser renderer is inserted into their paths. GenUI-Harness leads Pass@3 on seven of the nine backbones across open-weight and closed-source models. Macro-averaged across these backbones, it improves Pass@3 over `smolagents` by 4.48 percentage points (task-paired bootstrap 95% CI: [2.37, 6.59]); Table 10 reports uncertainty against all three baselines. Because the comparison preserves each harness's native agent loop and communication mode, it measures complete system performance rather than isolating the interface channel alone; the reviewer study in Section 6.3 compares the two communication channels directly. Full-split results are presented in Appendix D.6.

Interaction Efficiency. Figure 5 places GenUI-Harness alone on the favorable lower-right frontier: it achieves higher Avg@3 with 36–43% fewer agent turns among successful episodes. This suggests that the harness avoids redundant reasoning and recovery: the Tool Agent focuses on retrieval and validation, while the generated UI collects multiple grounded choices in one structured interaction.

Table 2: State EM Pass@3 and Avg@3 (%) on the 300-task Lite split across four harnesses (three runs; temperature 0.7). GENUI-Harness uses generated UI; the other harnesses retain native text interaction. Bold marks each row's best result; ties are bolded jointly.

	GENUI-Harness		smolagents		mini-swe-agent		Pi	
Model	**Pass@3**	**Avg@3**	**Pass@3**	**Avg@3**	**Pass@3**	**Avg@3**	**Pass@3**	**Avg@3**
OPEN-WEIGHT MODELS								
`Qwen3.5-4B`	9.33	3.22	10.67	3.90	11.67	4.11	**13.67**	**5.00**
`Qwen3.6-27B`	**54.33**	**32.44**	51.67	30.20	43.33	27.67	43.33	28.00
`Qwen3.6-35B-A3B`	**47.33**	**26.89**	43.67	25.90	36.33	20.11	38.00	21.78
`gpt-oss-120b`	47.67	**28.22**	30.00	14.10	**50.00**	**28.22**	40.33	23.11
`DeepSeek V4 Flash`	**49.67**	**30.22**	49.33	29.70	42.00	27.22	40.67	25.78
`Qwen3.5-397B-A17B`	**56.00**	**33.67**	48.33	32.90	43.33	28.22	42.33	27.89
CLOSED-SOURCE MODELS								
`GPT-5.4`	**40.00**	**25.22**	35.67	18.90	35.67	23.00	35.67	22.11
`Claude Sonnet 4.6`	**51.33**	**35.67**	48.00	31.30	47.67	33.56	40.67	28.44
`Claude Opus 5`	**46.67**	**31.56**	44.67	28.20	40.00	28.00	43.33	28.67

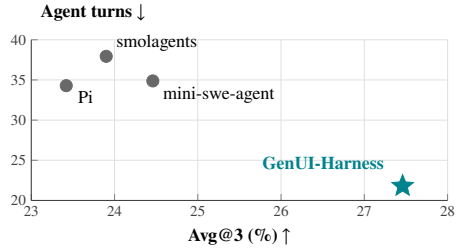

Figure 5: Avg@3 versus agent turns on Lite.

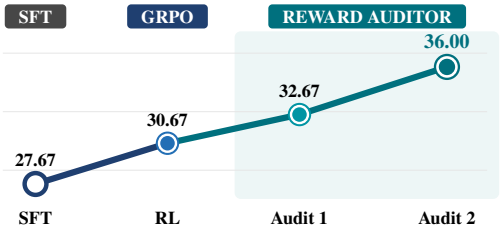

Figure 6: Avg@3 (%) of GUI-Agent on Lite.

Role-Specialized Training and Reward Auditing. We further train a compact `Qwen3.5-4B` within GENUI-Harness through role-specific supervision and audited rewards. SFT teaches the division of labor: the Tool Agent retrieves evidence and executes actions, while the GUI Coder Agent turns unresolved requirements into executable interfaces, increasing Avg@3 from 3.22% to 27.67%. We then isolate the contribution of RL by fixing the SFT Tool Agent and GUI-Coder initialization (Figure 6). Dynamic UX executes generated code and collects rollout rewards for GUI-Coder GRPO at scale. The initial judge-based reward can favor plausible interfaces with broken controls or payload semantics and assign indistinguishable scores to different failure stages. Reward Auditor compares these rewards with exact database-state outcomes, diagnoses distribution-level patterns and individual cases, and automatically revises the shared rubric and scoring rules through replay. Replay holds sampled interfaces fixed, separating changes in reward behavior from changes in policy outputs. The final reward distinguishes code-extraction, structural, rendering, and empty-interface failures before applying quality and accessibility scoring. With the final audited reward, GENUI-4B reaches 58.00% Pass@3 and 36.00% Avg@3, the best result in Figure 6 and above the strongest Table 2 baseline on both metrics.

6.3 ANALYSIS

Error Analysis. Across four models on the 300 Lite tasks, failures are dominated by unsuccessful UI interaction (52.0%) and mismatched action arguments (44.5%). Trace inspection identifies UI defects that prevent users from expressing required selections or submitting correct values; state mismatches require examining both submissions and resulting database changes. For example, a single-select control cannot express a required multi-selection, and an immutable cabin-class field submits the wrong value despite a fully specified user goal. These findings highlight interface operability and argument fidelity as key bottlenecks. Appendix E provides detailed counts, diagnostic procedures, and case studies.

Communication-Channel Analysis. We compare channels on 100 tasks whose interfaces execute and support valid user-simulator interaction in all three runs (pass^3). For each task, three reviewers see the UI and three see a text rendering of the same task state; assignments are randomized, and each reviewer sees only one channel. Generated UIs reduce average dialogue rounds from **3.4** to **1.2**. UI latency is judged acceptable on **85%** of tasks by a majority of UI reviewers. Appendix F details task selection, channel construction, latency measurement, and the evaluation protocol.

Code-Stage Intervention Across Models. To study the generalizability of GUI-Coder, we pair it with five other reasoning models within GENUI-Harness. For each model, we hold its original pre-agent output fixed, replace only code generation with GUI-Coder, collect a new UI, and rerun the same model as the post-agent. Table 3 shows higher Lite Avg@3 for all five models. Because the upstream exploration trace and post-agent backbone are fixed, the intervention changes only interface construction. The consistent improvements therefore indicate that GUI-Coder adapts to distinct reasoning models rather than relying on a particular upstream agent.

Table 3: Avg@3 (%) for code-stage intervention on Lite, reusing the same pre-agent output. Green arrows show absolute gains.

	Avg@3 (%) ↑	
Model	**Original**	**+ GUI-Coder**
`Qwen3.6-35B-A3B`	26.89	**29.33** (↑2.44)
`DeepSeek V4 Flash`	30.22	**38.67** (↑8.45)
`GPT-5.4`	25.22	**30.00** (↑4.78)
`Claude Sonnet 4.6`	35.67	**42.33** (↑6.66)
`Claude Opus 5`	31.56	**40.33** (↑8.77)

Evaluation on Unambiguous Tasks. In some scenarios, the user query states the intent clearly. To analyze GENUI-Harness performance in this setting, we create non-ambiguous Lite variants by revealing the user-facing arguments masked in the original requests while keeping the tools, databases, user simulator, and evaluation protocol fixed. Figure 7 shows that GENUI-4B remains strongest in both settings. We find that the UI provides guided task progression by organizing retrieved information for review and supporting confirmation of the intended action.

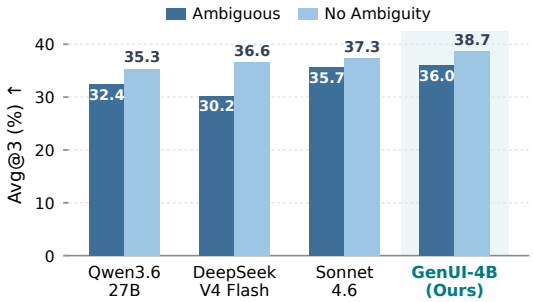

Figure 7: Avg@3 (%) on Lite, with and without ambiguity.

7 Related Work

Tool-using agents and state-based benchmarks study planning, API execution, and task completion, but generally retain language-based interaction (Yao et al., 2023; 2025). Clarification and mixed-initiative systems recover missing intent through questions, structured options, or generated interfaces, yet typically separate interface generation from database-backed action execution (Horvitz, 1999; Rao & Daumé III, 2018; Aliannejadi et al., 2019; Vaithilingam et al., 2024; Chen et al., 2026). Execution-based UI-code methods improve functional correctness through rendering or tests while treating code generation as the endpoint rather than part of an agent loop (Wu et al., 2024; Chen et al., 2023). We instead connect retrieved records, executable interaction, and terminal state evaluation within one task. Appendix B provides a detailed related-work discussion.

8 Conclusion

GENUI-Harness separates database grounding and action execution from task-conditioned interface generation. Dynamic UX scales GUI rollouts, while Reward Auditor revises rewards using task outcomes. On UI-TAU Bench Lite, GENUI-4B achieves 58.00% Pass@3 and 36.00% Avg@3, outperforming all evaluated baselines; GUI-Coder also improves five models. Among successful episodes, GENUI-Harness uses 36–43% fewer agent turns, and generated UIs reduce reviewer dialogue from 3.4 to 1.2 rounds. Together, these results support executable, data-grounded interfaces as an effective interaction layer for database-backed agents.

AI USE STATEMENT

Generative models are used as experimental components for synthetic benchmark construction, trajectory generation, user simulation, reward judging, and automated reward revision, as described in the paper. Generative AI tools also provided feedback on experimental protocols and result interpretation and assisted with preparing and checking the code and data preview. Separately, they were used to polish prose and formatting. The authors reviewed the AI-assisted work and take responsibility for the final text, claims, results, and artifacts.

REPRODUCIBILITY STATEMENT

Appendix C documents the benchmark sources, normalization, tool validation, and task synthesis. Appendix D provides training and inference settings, the Dynamic UX benchmark, and reward specifications. Appendices A, F, and H describe the agent workflow, reviewer evaluation protocol, and prompts. The accompanying anonymous code preview contains inference and Dynamic UX components with preview data.

REFERENCES

Mohammad Aliannejadi, Hamed Zamani, Fabio Crestani, and W. Bruce Croft. Asking clarifying questions in open-domain information-seeking conversations. In *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 475–484. Association for Computing Machinery, 2019. doi: 10.1145/3331184.3331265. URL https://doi.org/10.1145/3331184.3331265.

Mohammad Aliannejadi, Julia Kiseleva, Aleksandr Chuklin, Jeffrey Dalton, and Mikhail Burtsev. Building and evaluating open-domain dialogue corpora with clarifying questions. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 4473–4484. Association for Computational Linguistics, 2021. doi: 10.18653/v1/2021.emnlp-main.367. URL https://aclanthology.org/2021.emnlp-main.367/.

Anthropic. Introducing Claude Opus 5, July 2026a. URL https://www.anthropic.com/news/claude-opus-5.

Anthropic. Claude Sonnet 4.6 system card, February 2026b. URL https://www.anthropic.com/claude-sonnet-4-6-system-card.

Nuno António, Ana de Almeida, and Luís Nunes. Hotel booking demand datasets. *Data in Brief*, 22: 41–49, 2019. doi: 10.1016/j.dib.2018.11.126. URL https://doi.org/10.1016/j.dib.2018.11.126.

Denish Azamuke. Synthetic mobile money transaction dataset. Mendeley Data, Version 2, 2024. URL https://data.mendeley.com/datasets/zhj366m53p/2.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*, 2022.

Tony Beltramelli. pix2code: Generating code from a graphical user interface screenshot. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, pp. 3:1–3:6. Association for Computing Machinery, 2018. doi: 10.1145/3220134.3220135. URL https://doi.org/10.1145/3220134.3220135.

Yining Cao, Peiling Jiang, and Haijun Xia. Generative and malleable user interfaces with generative and evolving task-driven data model. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, 2025. doi: 10.1145/3706598.3713285. URL https://doi.org/10.1145/3706598.3713285.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code generation with generated tests. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=ktrw68Cmu9c.

Daqing Chen. Online retail II. UCI Machine Learning Repository, 2012. URL https://archive.ics.uci.edu/dataset/502/online+retail+ii.

Jiaqi Chen, Yanzhe Zhang, Yutong Zhang, Yijia Shao, and Diyi Yang. Generative interfaces for language models. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 1499–1519. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.74. URL https://aclanthology.org/2026.findings-acl.74/.

DeepSeek-AI. DeepSeek-V4-Flash-0731 model card. Hugging Face, July 2026. URL https://huggingface.co/deepseek-ai/DeepSeek-V4-Flash-0731. Official model card; accessed 2026-08-30.

Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, pp. 845–854. Association for Computing Machinery, 2017. doi: 10.1145/3126594.3126651. URL https://doi.org/10.1145/3126594.3126651.

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 *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=kiYqbO3wqw.

Yang Deng, Xuan Zhang, Wenxuan Zhang, Yifei Yuan, See-Kiong Ng, and Tat-Seng Chua. On the multi-turn instruction following for conversational web agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, pp. 8795–8812. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.477. URL https://aclanthology.org/2024.acl-long.477/.

Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H. Laradji, Manuel Del Verme, Tom Marty, David Vazquez, Nicolas Chapados, and Alexandre Lacoste. WorkArena: How capable are web agents at solving common knowledge work tasks? In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 11642–11662. PMLR, 2024. URL https://proceedings.mlr.press/v235/drouin24a.html.

Earendil Inc. and Contributors. Pi: A minimal agent harness. https://pi.dev/, 2026.

Krzysztof Z. Gajos, Daniel S. Weld, and Jacob O. Wobbrock. Automatically generating personalized user interfaces with Supple. *Artificial Intelligence*, 174(12–13):910–950, 2010. doi: 10.1016/j.artint.2010.05.005. URL https://doi.org/10.1016/j.artint.2010.05.005.

Leo Gao, John Schulman, and Jacob Hilton. Scaling laws for reward model overoptimization. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 10835–10866. PMLR, 2023. URL https://proceedings.mlr.press/v202/gao23h.html.

F. Maxwell Harper and Joseph A. Konstan. The MovieLens datasets: History and context. *ACM Transactions on Interactive Intelligent Systems*, 5(4):19:1–19:19, 2015. doi: 10.1145/2827872. URL https://doi.org/10.1145/2827872.

Eric Horvitz. Principles of mixed-initiative user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 159–166. Association for Computing Machinery, 1999. doi: 10.1145/302979.303030. URL https://doi.org/10.1145/302979.303030.

Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. Direct manipulation interfaces. *Human-Computer Interaction*, 1(4):311–338, 1985. doi: 10.1207/s15327051hci0104_2. URL https://www.tandfonline.com/doi/abs/10.1207/s15327051hci0104_2.

Seungone Kim, Juyoung Suk, Shayne Longpre, Bill Yuchen Lin, Jamin Shin, Sean Welleck, Graham Neubig, Moontae Lee, Kyungjae Lee, and Minjoon Seo. Prometheus 2: An open source language model specialized in evaluating other language models. In *Proceedings of the 2024*

Conference on Empirical Methods in Natural Language Processing, pp. 4334–4353. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.emnlp-main.248. URL https://aclanthology.org/2024.emnlp-main.248/.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Russ Salakhutdinov, and Daniel Fried. VisualWebArena: Evaluating multimodal agents on realistic visual web tasks. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, pp. 881–905. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.50. URL https://aclanthology.org/2024.acl-long.50/.

Nathan Lambert, Valentina Pyatkin, Jacob Morrison, LJ Miranda, Bill Yuchen Lin, Khyathi Chandu, Nouha Dziri, Sachin Kumar, Tom Zick, Yejin Choi, Noah A. Smith, and Hannaneh Hajishirzi. RewardBench: Evaluating reward models for language modeling. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 1755–1797. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-naacl.96. URL https://aclanthology.org/2025.findings-naacl.96/.

Jill H. Larkin and Herbert A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11(1):65–100, 1987. doi: 10.1111/j.1551-6708.1987.tb00863.x. URL https://doi.org/10.1111/j.1551-6708.1987.tb00863.x.

Hugo Laurencon, Leo Tronchon, and Victor Sanh. Unlocking the conversion of web screenshots into HTML code with the WebSight dataset. *arXiv preprint arXiv:2403.09029*, 2024. doi: 10.48550/arXiv.2403.09029. URL https://arxiv.org/abs/2403.09029.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 35, pp. 21314–21328, 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/8636419dea1aa9fbd25fc4248e702da4-Abstract-Conference.html.

Harrison Lee, Samrat Phatale, Hassan Mansoor, Thomas Mesnard, Johan Ferret, Kellie Ren Lu, Colton Bishop, Ethan Hall, Victor Carbune, Abhinav Rastogi, and Sushant Prakash. RLAIF vs. RLHF: Scaling reinforcement learning from human feedback with AI feedback. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 26874–26901. PMLR, 2024. URL https://proceedings.mlr.press/v235/lee24t.html.

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. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 3102–3116. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.187. URL https://aclanthology.org/2023.emnlp-main.187/.

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. In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=zAdUB0aCTQ.

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Haoping Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, Zirui Wang, and Ruoming Pang. ToolSandbox: A stateful, conversational, interactive evaluation benchmark for LLM tool use capabilities. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 1160–1183. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-naacl.65. URL https://aclanthology.org/2025.findings-naacl.65/.

Xing Han Lu, Zdenek Kasner, and Siva Reddy. WebLINX: Real-world website navigation with multi-turn dialogue. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 33007–33056. PMLR, 2024. URL https://proceedings.mlr.press/v235/lu24e.html.

Jiaju Ma, Lei Shi, Kenneth Aleksander Robertsen, and Peggy Chi. Ambigchat: Interactive hierarchical clarification for ambiguous open-domain question answering. In *Proceedings of the 38th Annual ACM Symposium on User Interface Software and Technology*, pp. 1–18, 2025.

Rafael Medellín and Juan Serna. Restaurant & consumer data. UCI Machine Learning Repository, 2011. URL https://archive.ics.uci.edu/dataset/232/restaurant+consumer+data.

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. GAIA: a benchmark for general ai assistants. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=fibxvahvs3.

Sewon Min, Julian Michael, Hannaneh Hajishirzi, and Luke Zettlemoyer. AmbigQA: Answering ambiguous open-domain questions. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*, pp. 5783–5797. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.emnlp-main.466. URL https://aclanthology.org/2020.emnlp-main.466/.

Palash Nandy, Sigurdur Orn Adalgeirsson, Anoop K Sinha, Tanya Kraljic, Mike Cleron, Lei Shi, Angad Singh, Ashish Chaudhary, Ashwin Ganti, Christopher A Melancon, et al. Bespoke: Using llm agents to generate just-in-time interfaces by reasoning about user intent. In *Companion Proceedings of the 26th International Conference on Multimodal Interaction*, pp. 78–81, 2024.

Olist. Brazilian e-commerce public dataset by olist. Kaggle, 2018. URL https://www.kaggle.com/datasets/olistbr/brazilian-ecommerce. CC BY-NC-SA 4.0.

Beatriz Oliveira, Maria Antónia Carravilla, and José Fernando Oliveira. Capacity-pricing model: Car rental instances. Mendeley Data, Version 1, 2017. URL https://data.mendeley.com/datasets/g49smv7nh8/1.

OpenAI. gpt-oss-120b and gpt-oss-20b model card, August 2025. URL https://openai.com/index/gpt-oss-model-card/.

OpenAI. Introducing GPT-5.4, March 2026. URL https://openai.com/index/introducing-gpt-5-4/.

OpenFlights. Airport, airline and route data. OpenFlights Data Repository, n.d. URL https://openflights.org/data.php. Accessed 2026-08-30; distributed under the Open Database License, with component-specific source terms.

Yue Peng, Lanke Xia, Zihan Wang, Jiahao Ye, Ke Ning, and Hongyi Wen. EvoGenUI-Bench: Evaluating LLMs as multi-turn generative UI assistants, 2026. URL https://arxiv.org/abs/2608.29387. Accepted to EMNLP 2026.

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. In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=dHng2O0Jjr.

Qwen Team. Qwen3.5-397B-A17B model card. Hugging Face, 2026a. URL https://huggingface.co/Qwen/Qwen3.5-397B-A17B. Official model card; accessed 2026-09-24.

Qwen Team. Qwen3.5-4B model card. Hugging Face, 2026b. URL https://huggingface.co/Qwen/Qwen3.5-4B. Official model card; accessed 2026-08-30.

Qwen Team. Qwen3.6-27B model card. Hugging Face, 2026c. URL https://huggingface.co/Qwen/Qwen3.6-27B. Official model card; accessed 2026-09-10.

Qwen Team. Qwen3.6-35B-A3B: Agentic coding power, now open to all, April 2026d. URL https://qwen.ai/blog?id=qwen3.6-35b-a3b. Official release article; accessed 2026-08-30.

Sudha Rao and Hal Daumé III. Learning to ask good questions: Ranking clarification questions using neural expected value of perfect information. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, pp. 2737–2746. Association for Computational Linguistics, 2018. doi: 10.18653/v1/P18-1255. URL https://aclanthology.org/P18-1255/.

Aymeric Roucher, Albert Villanova del Moral, Thomas Wolf, Leandro von Werra, and Erik Kaunismäki. smolagents: A smol library to build great agentic systems. https://github.com/huggingface/smolagents, 2025.

Luiz Henrique Salazar. Medical appointments no-show. Mendeley Data, Version 1, 2023. URL https://data.mendeley.com/datasets/wm6w2fvkfj/1.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. doi: 10.52202/075280-2997. URL https://proceedings.neurips.cc/paper/2023/hash/d842425e4bf79ba039352da0f658a906-Abstract-Conference.html.

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. URL https://arxiv.org/abs/2402.03300.

Ben Shneiderman. Direct manipulation: A step beyond programming languages. *Computer*, 16 (8):57–69, 1983. doi: 10.1109/MC.1983.1654471. URL https://doi.org/10.1109/MC.1983.1654471.

Chenglei Si, Yanzhe Zhang, Ryan Li, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. Design2code: Benchmarking multimodal code generation for automated front-end engineering. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics*, pp. 3956–3974. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.naacl-long.199. URL https://aclanthology.org/2025.naacl-long.199/.

Joar Skalse, Nikolaus H. R. Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and characterizing reward gaming. In *Advances in Neural Information Processing Systems*, volume 35, 2022. doi: 10.52202/068431-0687. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/3d719fee332caa23d5038b8a90e81796-Abstract-Conference.html.

SWE-agent Team. mini-swe-agent: The minimal ai software engineering agent. https://github.com/SWE-agent/mini-swe-agent, 2025.

Sijun Tan, Siyuan Zhuang, Kyle Montgomery, William Y. Tang, Alejandro Cuadron, Chenguang Wang, Raluca Ada Popa, and Ion Stoica. JudgeBench: A benchmark for evaluating LLM-based judges. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=G0dksFayVq.

Ticketmaster. Discovery API v2. Ticketmaster Developer Portal, n.d. URL https://developer.ticketmaster.com/products-and-docs/apis/discovery-api/v2/. Accessed 2026-08-30; subject to Ticketmaster Developer Terms of Use.

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 *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, pp. 16022–16076. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.850. URL https://aclanthology.org/2024.acl-long.850/.

Priyan Vaithilingam and Philip J. Guo. Bespoke: Interactively synthesizing custom GUIs from command-line applications by demonstration. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, pp. 563–576. Association for Computing Machinery, 2019. doi: 10.1145/3332165.3347944. URL https://doi.org/10.1145/3332165.3347944.

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*. Association for Computing Machinery, 2024. doi: 10.1145/3613904.3642639. URL https://doi.org/10.1145/3613904.3642639.

W3C. Accessible rich internet applications (WAI-ARIA) 1.2. W3C Recommendation, June 2023. URL https://www.w3.org/TR/wai-aria-1.2/. Section 2, Important Terms.

Jason Wu, Eldon Schoop, Alan Leung, Titus Barik, Jeffrey Bigham, and Jeffrey Nichols. UICoder: Finetuning large language models to generate user interface code through automated feedback. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics*, pp. 7511–7525. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.naacl-long.417. URL https://aclanthology.org/2024.naacl-long.417/.

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*, volume 37, 2024. doi: 10.52202/079017-1650. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5d413e48f84dc61244b6be550f1cd8f5-Abstract-Datasets_and_Benchmarks_Track.html.

Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

Jingjing Xu, Yuechen Wang, Duyu Tang, Nan Duan, Pengcheng Yang, Qi Zeng, Ming Zhou, and Xu Sun. Asking clarification questions in knowledge-based question answering. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*, pp. 1618–1629. Association for Computational Linguistics, 2019. doi: 10.18653/v1/D19-1172. URL https://aclanthology.org/D19-1172/.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. In *Thirty-sixth Conference on Neural Information Processing Systems*, 2022. doi: 10.52202/068431-1508. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/82ad13ec01f9fe44c01cb91814fd7b8c-Abstract-Conference.html.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=roNSXZpUDN.

Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. Sikuli: Using GUI screenshots for search and automation. In *Proceedings of the 22nd Annual ACM Symposium on User Interface Software and Technology*, pp. 183–192. Association for Computing Machinery, 2009. doi: 10.1145/1622176.1622213. URL https://doi.org/10.1145/1622176.1622213.

Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Xian Li, Sainbayar Sukhbaatar, Jing Xu, and Jason Weston. Self-rewarding language models. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 57905–57923. PMLR, 2024. URL https://proceedings.mlr.press/v235/yuan24d.html.

Hamed Zamani, Gord Lueck, Everest Chen, Rodolfo Quispe, Flint Luu, and Nick Craswell. MIMICS: A large-scale data collection for search clarification. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management*, pp. 3189–3196. Association for Computing Machinery, 2020. doi: 10.1145/3340531.3412772. URL https://doi.org/10.1145/3340531.3412772.

Michael JQ Zhang and Eunsol Choi. Clarify when necessary: Resolving ambiguity through interaction with LMs. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 5541–5558. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-naacl.306. URL https://aclanthology.org/2025.findings-naacl.306/.

Tong Zhang, Peixin Qin, Yang Deng, Chen Huang, Wenqiang Lei, Junhong Liu, Dingnan Jin, Hongru Liang, and Tat-Seng Chua. CLAMBER: A benchmark of identifying and clarifying ambiguous information needs in large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, pp. 10746–10766. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.578. URL https://aclanthology.org/2024.acl-long.578/.

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 *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023. doi: 10.52202/075280-2020. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/91f18a1287b398d378ef22505bf41832-Abstract-Datasets_and_Benchmarks.html.

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 *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=rmiwIL98uQ.

A Task Formulation and Agent Workflow

A.1 Formal Task Definition and Evaluation

A task consists of a user query, an initial domain database, query and action tools with domain rules, and a specific user goal. The query may express only part of that goal. The agent observes the query, tool schemas, retrieved records, and subsequent user input. Read-only query calls gather evidence; the generated interface obtains user input; one terminal action updates the database.

Each task specifies an initial database state s^0_{db} and a reference action $a^\star = (\texttt{name}^\star, \mathbf{v}^\star)$, where $\mathbf{v}^\star = \{(k_j, v_j)\}_{j=1}^n$ contains the grounded value of every required argument. Executing $a^\star$ from s^0_{db} produces the target database state $s^\star_{\mathrm{db}}$ used for evaluation. For task i and run $j \in \{1, 2, 3\}$, the evaluator resets an isolated database copy to $s^0_{\mathrm{db},i}$, executes the agent's terminal action, and observes the resulting state $\hat{s}_{\mathrm{db},i,j}$. Let $e_{i,j} = 1$ indicate successful execution. The state-EM outcome is

$$z_{i,j} = \mathbb{I}\big[e_{i,j} = 1 \ \wedge \ \hat{s}_{\mathrm{db},i,j} = s^\star_{\mathrm{db},i}\big] .$$

Over N tasks, Pass@3 is $\frac{100}{N}\sum_i \max_j z_{i,j}$ and Avg@3 is $\frac{100}{3N}\sum_i \sum_j z_{i,j}$. The entire resulting database state must match the target. This credits any action realization that induces the verified state transition, while failed execution and every missing, extra, or incorrect state change count as failures.

A.2 Detailed Agent Workflow

The Tool Agent performs up to five read-only query turns before UI generation. The GUI Coder Agent receives the initial query and exploration history, first reflects on the remaining information needs, and then produces a single React/TypeScript component. The interface displays retrieved options and collects user input through `submitData`. The Tool Agent resumes with the query, exploration history, and submitted JSON, uses up to five further query turns to validate or resolve selections, and emits one terminal action.

The user simulator receives the generated TSX, the initial ARIA snapshot, and the complete user goal encoded as the reference action. It generates Playwright code to operate the interface according to that goal. The simulator is fixed across all harnesses and nine backbones and does not determine success. In generated-UI runs, it cannot invoke task tools or directly modify the database; it must act through the rendered interface, after which the fixed Tool Agent consumes its submission and the evaluator independently checks the terminal database state. The reference action is private to the simulator and is not an additional input to the Tool Agent or GUI Coder Agent. The same simulator also operates interfaces produced by unrelated model families, while the reviewer study in Appendix F confirms that people can operate the selected interfaces under the same task information. We therefore use it as a controlled, capable-user simulator without claiming that it reproduces the full distribution of human interaction behavior. Appendix H gives the role templates, and Appendix D.3 specifies serving and retry settings.

B Related Work

Tool-Using and Interactive Agents. Tool-using agents combine language reasoning with API calls to retrieve information and execute actions (Yao et al., 2023; Schick et al., 2023; Li et al., 2023; Qin et al., 2024). Their evaluation covers general assistance and stateful environments (Liu et al., 2024; Mialon et al., 2024; Trivedi et al., 2024), as well as interaction with existing web and desktop interfaces (Yao et al., 2022; Deng et al., 2023; Zhou et al., 2024; Koh et al., 2024; Drouin et al., 2024; Xie et al., 2024). Conversational benchmarks further study how agents respond to evolving user requests during navigation and tool use (Lu et al., 2024; Deng et al., 2024; Lu et al., 2025). Tau-Bench is particularly close to our setting: it combines simulated users, domain-specific tools, and database-state evaluation, with clarification conducted through textual dialogue (Yao et al., 2025). We retain database-backed task execution but change how user choices are elicited: the agent generates an executable interface from retrieved records and the action schema, allowing users to inspect options and supply the choices needed for the final action.

Clarification and Generative Interfaces. Mixed-initiative systems coordinate automated inference with user control, while clarification research studies how agents identify and resolve underspec-

ified requests (Horvitz, 1999; Rao & Daumé III, 2018; Aliannejadi et al., 2019; Xu et al., 2019; Min et al., 2020; Zhang et al., 2024; Zhang & Choi, 2025). MIMICS operationalizes clarification through question-and-option panes, and AmbigChat uses interactive widgets for hierarchical disambiguation (Zamani et al., 2020; Ma et al., 2025). In HCI, direct manipulation and automatically generated interfaces have long been used to expose task structure and reduce the gap between intent and action (Shneiderman, 1983; Hutchins et al., 1985; Gajos et al., 2010). More recent systems synthesize task-specific or malleable interfaces from demonstrations, natural-language requests, or evolving data models (Vaithilingam & Guo, 2019; Vaithilingam et al., 2024; Nandy et al., 2024; Cao et al., 2025; Chen et al., 2026). Contemporaneous EvoGenUI-Bench, accepted to EMNLP 2026, evaluates whether one executable interface preserves requirements and tool grounding across five successive revision turns (Peng et al., 2026); our benchmark instead evaluates task-specific interfaces that elicit unresolved intent and complete a terminal database action. Our setting is distinguished by database-grounded service interaction: retrieved records and an executable action schema determine what the interface must clarify, and correctness is verified by executing the terminal action and exactly checking the resulting database state.

UI Code Generation and Outcome-Based Feedback. Prior work includes screenshot-based GUI automation and datasets (Yeh et al., 2009; Deka et al., 2017), screenshot-to-code reconstruction (Beltramelli, 2018; Laurencon et al., 2024; Si et al., 2025), and UI-code fine-tuning with compiler and multimodal feedback (Wu et al., 2024). CodeRL and CodeT demonstrate the value of execution and generated tests when reference similarity is insufficient for functional correctness (Le et al., 2022; Chen et al., 2023). Model-generated feedback has also been used for scalable alignment and self-improvement (Bai et al., 2022; Lee et al., 2024; Yuan et al., 2024), but judge and reward-model reliability remains an active concern (Zheng et al., 2023; Kim et al., 2024; Lambert et al., 2025; Tan et al., 2025). In particular, optimizing imperfect proxies can induce overoptimization or reward gaming (Gao et al., 2023; Skalse et al., 2022). We combine online execution-gated feedback with a meta-reward process that uses observed task outcomes to diagnose and iteratively revise the judging criteria and scoring rules.

C UI-TAU BENCH CONSTRUCTION DETAILS

This appendix provides additional details on schema design, database population, domain rules, tool generation and validation, task synthesis, and quality control for UI-TAU Bench.

C.1 DETAILED CONSTRUCTION PIPELINE

The construction pipeline follows the three top-level stages shown in Figure 2, with each stage producing a validated artifact required by the next. During Tool Design, corresponding domain rules specify action requirements, service constraints, and valid state transitions before the tools are constructed and validated. This ordering prevents errors in collected data, rules, or tool behavior from propagating into the final benchmark instances. All pseudorandom choices in data construction use seed 42.

Stage 1: Domain Data Collection. The benchmark uses 10 real-world domain databases constructed from the public data sources documented in Table 4. The same annotators collect and preprocess these sources by standardizing field types and identifiers and checking cross-table references, then organize each domain into a user-facing entity table, resource tables, and a transaction table. The resulting databases are used for tool and task construction.

Stage 2: Tool Design. The same annotators first specify corresponding domain rules that govern action requirements, eligibility conditions, service constraints, and valid state transitions. They then define query and action tools over the normalized database. Query tools are atomic, read-only operations designed to compose into diverse search chains. Their combined coverage includes broad overview operations, such as listing the records available to a user-facing entity, and targeted retrieval operations, such as finding a resource by a date, location, status, or identifier. Action tools implement rule-compliant database writes across multiple service scenarios. Their database-valued arguments are grounded in actual query observations so that an agent can obtain the required values through

Table 4: Public data sources for the 10 evaluation domains. The table summarizes the domain information provided by each source and its license or access terms.

Domain	Public data source	Domain entities or workflows	License / terms
Flight booking	OpenFlights airport, airline, and route data (OpenFlights, n.d.)	Airports, airlines, routes, equipment, and route-level connectivity	ODbL; component terms vary
Hotel booking	Hotel Booking Demand datasets (António et al., 2019)	Hotel bookings, arrival dates, room assignment, cancellation status, and special requests	CC BY 4.0
Bank account	Synthetic Mobile Money Transaction Dataset (Azamuke, 2024)	Account identifiers, transfers, transaction amounts, and balance transitions	CC BY 4.0
Restaurant reservation	UCI Restaurant & Consumer Data (Medellín & Serna, 2011)	User profiles and preferences, restaurants, accepted payments, hours, and ratings	CC BY 4.0
E-commerce	Olist Brazilian E-Commerce Dataset (Olist, 2018)	Customers, sellers, products, orders, payments, delivery status, and reviews	CC BY-NC-SA 4.0
Movie ticket	MovieLens 25M (Harper & Konstan, 2015)	Pseudonymous users, movies, ratings, tags, and preference signals	GroupLens research-use terms
Event ticketing	Ticketmaster Discovery API v2 (Ticketmaster, n.d.)	Events, venues, classifications, dates, locations, and availability filters	Ticketmaster developer terms
Medical appointment	Medical Appointments No-Show (Salazar, 2023)	Patient attributes, specialties, appointment time, attendance status, and no-show reasons	CC BY 4.0
Retail	UCI Online Retail II (Chen, 2012)	Customers, products and SKUs, invoices, order lines, prices, and cancellations	CC BY 4.0
Car rental	Capacity-Pricing Model car-rental instances (Oliveira et al., 2017)	Rental demand, vehicle groups, fleet capacity, and time- and price-dependent decisions	CC BY 4.0

the available tool interface. Each tool is paired with a schema that describes its purpose, arguments, constraints, and return values.

Tool Validation and Repair. We validate query and action tools before using them for task synthesis. Each query tool is executed over the user-facing entity records, and each action tool is tested with inputs grounded in real query observations; action validation checks both compliance with the domain rules and the resulting database state change. An execution error or empty query result initiates a repair-and-revalidation loop. We re-execute the repaired implementation against the failed validation case and repeat this process at most twice. A candidate that remains unsuccessful after the second repair is dropped. We subsequently execute every proposed tool sequence end to end: a sequence is rejected if any tool raises an execution error or if it contains more than two empty query results. The surviving sequence must terminate in an executable reference action whose resulting database state exactly matches the target state. This separates artifact-level tool generation from task-level retention and ensures that only verified execution paths contribute to the benchmark.

Stage 3: Task Synthesis. One of these annotators first samples a random user and specifies a service goal, such as changing an existing flight booking. Given the selected user's records, the domain database, and the available query tools, a task synthesis agent instantiated with `Claude Opus 5` proposes a chain of query calls for that goal. Each call draws its arguments from the database rather than from invented values, and later calls depend on the results of earlier ones, so that a chain traverses the schema the way an agent would, from the user record to an existing transaction and then to the resources that transaction involves. A chain contains 8.7 query calls on average. The synthesis agent executes the proposed chain against the domain database and then selects an action tool appropriate to the retrieved records. Each argument of that action is filled with a value observed during execution,

so that every argument of the reference action is grounded in a retrieved record. We then remove information from the description the evaluated agent receives. Every internal identifier required by the action, such as a booking or record identifier, is masked, so the evaluated agent must retrieve it from the database. The user identifier remains visible so that the evaluated agent can determine which account the task concerns. Two ambiguity levels determine how many further arguments are masked. Moderate-ambiguity tasks mask 2 to 3 additional arguments selected at random and high-ambiguity tasks mask 4 to 5. Masked values are recoverable only through database exploration, user interaction, or both. Conditioned on the query chain, its execution results, and the domain rules, the synthesis agent generates the request the evaluated agent receives. The request states the user goal in natural language and leaves the masked arguments unspecified, so that the resulting ambiguity is what the evaluated agent must resolve. A candidate task is retained only if it passes three checks. An execution check re-runs the reference action and confirms that it produces the intended database state. An alignment check confirms that the user profile, the request, the retrieved records, the domain rules, and the reference action are mutually consistent. Two independent annotators confirm that the request is realistic for the domain and fluent and that no masked value appears; disagreements are resolved by a third annotator. Table 1 reports inter-annotator agreement.

C.2 Benchmark Statistics and Comparison

Table 6 reports the database composition and task counts for each evaluation domain. The Lite and Full splits maintain comparable query length, ambiguity density, and annotation agreement, while Full broadens task and action-tool coverage (Table 1). Table 5 compares the interaction setting and evaluation target of closely related benchmarks. Tau-Bench and ToolSandbox elicit user information through dialogue. AppWorld evaluates generated programs that operate application APIs, while WebLINX evaluates agents that navigate existing websites under conversational instructions. In UI-Tau Bench, generated front-end code instead creates the interface through which the user provides task information; tool execution then connects that input to an objectively evaluated database update.

Table 5: Comparison with related benchmarks, using the settings and scale reported in the cited papers. Tasks, scenarios, and demonstrations are distinct dataset units. Generated UI denotes front-end code produced for the user to operate, rather than code used by the agent to call APIs or navigate an existing interface.

Benchmark	**Reported scale**	**Interaction setting**	**Evaluation target**
Tau-Bench (Yao et al., 2025)	165 tasks; 2 domains	Agent converses with a simulated user and calls domain APIs	Final database state versus annotated goal
ToolSandbox (Lu et al., 2025)	1,032 scenarios; 34 tools	Agent converses with a simulated user and executes stateful tools	Intermediate and final milestones; forbidden events
AppWorld (Trivedi et al., 2024)	750 tasks; 9 apps; 457 APIs	Agent writes and executes programs to operate app APIs	State-based unit tests for task completion and unintended changes
WebLINX (Lu et al., 2024)	Approximately 2,300 demonstrations; over 150 websites	Agent follows dialogue instructions to navigate existing websites	Next-action prediction against expert demonstrations
UI-Tau Bench	300 Lite and 1,000 Full tasks; 10 domains	Agent generates an executable UI for user input, then acts through tools	Final database-state exact match after UI interaction and action execution

Table 6: UI-TAU Bench domain statistics. Every domain provides a user-facing entity table, resource tables, and a transaction table. Columns and records count fields and rows across these tables; the table also reports the retained tasks in each split.

Domain	Domain-specific tables	Columns	Records	Lite	Full
Flight booking	flights, bookings, airports	56	156	30	100
Hotel booking	hotels, room inventory, bookings	58	70	30	100
Bank account	accounts, transactions, loans	67	70	30	100
Restaurant reservation	restaurants, reservations, reviews	63	70	30	100
E-commerce	products, orders, product reviews	52	70	30	100
Movie ticket	movies, theaters, bookings	60	70	30	100
Event ticketing	events, venues, ticket purchases	63	70	30	100
Medical appointment	appointments, physicians, clinics	65	70	30	100
Retail	products, orders, store locations	44	70	30	100
Car rental	vehicles, branches, rental bookings	48	70	30	100
Total		**576**	**786**	**300**	**1,000**

Tool Semantics Review. Two of these annotators apply the validation criteria from Section 3 to every surviving tool, not to a sample. Each tool is checked independently by a second annotator against the domain rules and the resulting database state change, and a tool is retained only when both annotators agree it is correct. A tool the two annotators cannot agree on, or whose behavior remains ambiguous after the repair loop described above, is discarded rather than retained.

Task Quality Review. The same annotators apply this check to every candidate task that passes the execution and alignment checks from Section 3, again not to a sample. For each candidate, two annotators rate it independently, judging whether the request is realistic for the domain, fluent, and free of any masked value. A disagreement between these two initial annotators is resolved by a third annotator through discussion, and a candidate whose ambiguity this discussion cannot resolve is dropped rather than retained. Table 1 reports agreement between the initial independent ratings separately for Lite (94.1%) and Full (94.3%).

D TRAINING DETAILS

D.1 TRAINING DATA CONSTRUCTION

The training pool contains 40 domain databases, all disjoint from the 10 evaluation domains. Before optimization, we partition them into 25 SFT databases and 15 RL databases, with no database shared across the two sets. We use `DeepSeek V4 Flash` to generate end-to-end trajectories within GENUI-Harness and retain a trajectory only when its terminal action executes and reaches the target database state. Each retained SFT task contributes one interface-generation example for the GUI Coder Agent and one or more pre- or post-agent examples for the Tool Agent. The RL split supplies task prompts for GUI-Coder GRPO and Reward Auditor sampling; the SFT Tool Agent remains fixed during RL. Table 7 summarizes the resulting data.

Table 7: Training-domain split and data volume (train/validation). Each task contributes one GUI-Coder example or RL prompt; an SFT task can contribute multiple Tool Agent examples.

Split	Databases	Tasks	Tool Agent data	GUI-Coder data
SFT	25	12,376 / 254	32,409 / 669	12,376 / 254
RL	15	8,116 / 169	–	8,116 / 169

ARIA snapshots and split verification. An ARIA snapshot records the rendered interface's accessible roles, names, and structure (W3C, 2023). We use it as execution evidence for the reward judge. The 40 training databases and 10 evaluation domains are constructed from separate domain lists. We check for exact overlap in user-task text, serialized reference actions, and executable

tool-name sequences; none is found. For recurring generic tool names, we additionally inspect arguments and implementations to check for reused tools or databases.

D.2 Model and Optimization Details

Models and role specialization. The Tool Agent and GUI Coder Agent use separate checkpoints initialized from `Qwen3.5-4B` (Qwen Team, 2026b). Both are first trained with full-parameter supervised fine tuning. In the GenUI-4B configurations, the Tool Agent is then kept fixed while only the GUI Coder Agent undergoes RL. Starting from the GUI Coder Agent SFT checkpoint, we train three independent RL policies for Direct RL, Audit Round 1, and Audit Round 2 (final). The final audited policy is GUI-Coder. The GenUI-4B designation refers to the active model size per call: although the system stores two role-specific checkpoints, each Tool Agent or GUI Coder Agent invocation activates only one 4B model. Table 8 summarizes the SFT settings.

Table 8: Supervised fine-tuning configuration for the two role-specific agents.

Setting	Tool Agent	GUI Coder Agent
Training / validation examples	32,409 / 669	12,376 / 254
Parameter update	Full fine-tuning	Full fine-tuning
Optimizer	AdamW	AdamW
Peak learning rate	1×10^{-5}	1×10^{-5}
Weight decay / gradient clip	0.05 / 1.0	0.05 / 1.0
Global batch size	32	32
Maximum sequence length	12,288	12,288
Schedule	Cosine; 20 warmup steps minimum LR 1×10^{-6}	Cosine; 10 warmup steps minimum LR 1×10^{-6}
Configured epochs	3	3
Data seed	42	42

GRPO optimization. We optimize only the GUI Coder Agent with GRPO (Shao et al., 2024), keeping the Tool Agent fixed at its best SFT checkpoint. The reported runs use 1,500 training and 41 validation prompts from the larger RL pool in Table 7. GRPO samples a group of interfaces for each prompt and uses their relative rewards to compute advantages, avoiding a separate value model. Let $\hat{A}_i$ denote the group-relative advantage of output y_i and $\rho_{i,t} = \pi_\theta(y_{i,t} \mid x, y_{i,<t}) / \pi_{\text{old}}(y_{i,t} \mid x, y_{i,<t})$ the token-level importance ratio. The clipped policy objective, with the KL coefficient set to zero, is

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}\left[\frac{1}{G}\sum_{i=1}^{G}\frac{1}{|y_i|}\sum_{t=1}^{|y_i|}\min\Big(\rho_{i,t}\hat{A}_i,\ \text{clip}(\rho_{i,t}, 1-\epsilon, 1+\epsilon)\hat{A}_i\Big)\right],$$

where ϵ controls the clipping range and G is the number of outputs sampled per prompt. Table 9 summarizes the shared training configuration.

Table 9: Shared GRPO configuration for the three independent GUI Coder Agent runs.

Setting	Value
Initialization	GUI Coder Agent SFT checkpoint
Training / validation prompts	1,500 / 41
Optimizer	AdamW; learning rate 5×10^{-6}; weight decay 0
Update budget	500 updates; 32 rollouts per update
Grouped sampling	8 rollouts per prompt; oversampling factor 2.0
Sequence limit	16,384 tokens
Policy objective	GRPO clipped surrogate; KL coefficient 0
LoRA	Rank 16; alpha 32
Sampling	Temperature 1.0; top-p 1.0; min-p 0.05
Validation	Every 40 updates; 4 rollouts per validation prompt
Seed	42
Hardware	8×NVIDIA H200 GPUs
Framework	`prime-rl` 0.7.0; `vLLM` 0.24.0; PyTorch 2.11.0+cu128

D.3 Evaluation Inference Configuration

The open-weight baselines are `Qwen3.5-4B`, `Qwen3.6-35B-A3B`, and `Qwen3.6-27B` (Qwen Team, 2026b;d;c), `gpt-oss-120b` (OpenAI, 2025), `DeepSeek V4 Flash` (DeepSeek-AI, 2026), and `Qwen3.5-397B-A17B` (Qwen Team, 2026a). The closed-source baselines are `GPT-5.4` (OpenAI, 2026), `Claude Sonnet 4.6` (Anthropic, 2026b), and `Claude Opus 5` (Anthropic, 2026a).

The Lite harness comparison in Table 2 and the training ablation in Figure 6 use temperature 0.7 for the evaluated model. All evaluation stages use a 32K-token output budget. Local `vLLM` requests leave top-p unset and therefore use the serving default, whereas closed-source API requests and the user-simulator call explicitly use top-$p = 1.0$. Open-weight models and our role-specific checkpoints are served through `vLLM`; the closed-source models are accessed through a hosted API under the aliases `openai-gpt-5.4`, `claude-sonnet-4-6`, and `claude-opus-5`. All three closed-source API baselines use the default thinking effort. The fixed user simulator is served as `DeepSeek-V4-Flash-0731`. These are the exact identifiers exposed to the evaluation pipeline; the serving APIs do not expose dated snapshots for these aliases.

Each structured generation permits at most three attempts when the required output cannot be extracted; transport-level API requests permit up to 30 attempts. These are internal retries within one evaluation run, distinct from the three complete runs used for Pass@3 and Avg@3. Browser rendering and user-simulator execution each have a 60-second timeout. Each complete run starts from an isolated copy of the task's initial database state. Table 2 reports all results to two decimal places.

D.4 Bootstrap Confidence Intervals for Lite Results

To assess whether the aggregate advantages in Table 2 are robust to task sampling, we perform a paired bootstrap over the 300 Lite tasks. Using random seed 42, in each of 10,000 replicates we sample task identifiers with replacement, retain all three runs for every sampled task, and apply the same sampled identifiers to every harness and backbone. We compute the paired difference for each backbone and then macro-average across the fixed set of nine evaluated backbones. Table 10 reports percentile 95% confidence intervals. All intervals remain above zero for both metrics, showing that the aggregate gains of GenUI-Harness are consistent across bootstrap resamples.

Table 10: Macro-average paired differences across the nine shared backbones on Lite. Intervals are 95% task-paired bootstrap confidence intervals over 10,000 resamples.

Baseline	Δ Pass@3 (pp), 95% CI	Δ Avg@3 (pp), 95% CI
`smolagents`	+4.48 [2.37, 6.59]	+3.56 [2.15, 5.00]
`mini-swe-agent`	+5.81 [3.37, 8.26]	+3.00 [1.46, 4.58]
`Pi`	+7.15 [4.30, 9.96]	+4.04 [2.30, 5.81]

D.5 Baseline Harness Adaptation

The three baselines retain their original agent loops and text-based user interaction; we do not insert the GUI Coder Agent or renderer into them. They share the benchmark tasks, database, Python implementations of the query and action tools, user-simulator model and hidden goal, and final-state evaluator. When a baseline requests information, the simulator receives its textual message and interaction history and replies in natural language. This gives an end-to-end text-only comparison while keeping the task environment and success criterion fixed.

We adapt only the tool and termination boundaries required by each framework. In `smolagents`, the benchmark tools are registered through its native agent API. In `mini-swe-agent`, we replace the default shell-only action surface with the benchmark's JSON-schema tool set and add an explicit finish action. In `Pi`, its `pi-agent-core` Node.js loop accesses the same Python tools through a localhost HTTP bridge. Where a framework does not natively impose the benchmark's termination conditions, an external check applies the configured stopping rule.

Some role-specific checkpoints emit tagged tool calls rather than native function calls. For these checkpoints, a format adapter renders the expected prompt and maps tagged outputs into each framework's internal tool-call representation. This conversion does not modify generated arguments, tool observations, database state, or grading; beyond these compatibility layers, we retain each framework's own agent loop.

D.6 Full Split Results

The Full split contains 1,000 tasks, each requiring end-to-end tool use, user interaction, and state evaluation; GenUI-Harness additionally generates and executes an interface. Due to the resulting computational cost, we run each task once and restrict this four-harness comparison to open-weight models. Table 11 reports single-run Success Rate, equivalently Pass@1. Under the same Full protocol, our role-specialized GenUI-4B achieves **40.90%** Success Rate.

Table 11: State EM Pass@1 (single-run Success Rate, %) on the Full split of UI-Tau Bench (1,000 tasks) across four agent harnesses in their native interaction modes. Bold marks the best result in each row.

Model	GenUI-Harness	smolagents	mini-swe-agent	Pi
Open-weight models				
`Qwen3.5-4B`	4.11	3.80	4.70	**5.30**
`Qwen3.6-27B`	**37.50**	34.50	21.00	27.00
`Qwen3.6-35B-A3B`	**35.30**	27.10	24.50	21.90
`gpt-oss-120b`	**32.20**	14.50	30.00	25.50
`DeepSeek V4 Flash`	**33.00**	31.80	29.40	28.80
`Qwen3.5-397B-A17B`	**37.70**	36.80	28.00	29.70

D.7 Dynamic UX Systems Benchmark

We compare two process-management strategies on 1,100 pre-verified successful TSX components, sampling 100 from each of 11 evaluated model configurations with seed 42. The fresh-process implementation launches and tears down a Chromium browser for every rollout. Dynamic UX launches one persistent browser instance and creates a fresh, isolated `browser.new_context()` for every rollout. Both use the same per-render identifier, websocket messages, component-loaded signal, and dedicated websocket-server–Vite pair; only the browser lifecycle differs.

We run both implementations at concurrency 1, 4, 8, 16, and 32 in the same 16-core Rocky Linux 9.8 sandbox. Each configuration is repeated three times with newly shuffled sample orders drawn from a pseudorandom stream initialized with seed 42. The first *concurrency* completed requests in each run are treated as warm-up and excluded from latency and throughput, but remain in failure and timeout counts. A 200-ms background sampler records RSS and the number of Chromium processes in the harness process tree. Table 12 averages throughput and latency across repeats; peak RSS and process count are the maxima observed across the three repeats.

At concurrency 1, the two architectures perform similarly. As concurrency increases, repeatedly launching browser process trees becomes increasingly expensive. Dynamic UX reaches its highest throughput at concurrency 16, where it improves throughput by 37.5%, reduces median and P95 latency by 29.2% and 41.3%, reduces peak RSS by 48.1%, and uses 76.0% fewer Chromium processes than fresh-process rendering. At concurrency 32, both architectures approach saturation; Dynamic UX still provides 11.3% higher throughput with 50.9% lower peak RSS and 79.7% fewer Chromium processes. Separately, we render 1,000 sessions containing unique marker strings through Dynamic UX at concurrency 32. All sessions complete without timeout or rendering failure, and no page contains another session's marker.

D.8 Initial Reward Specification

The initial rubric translates the requirements of a task-completing interface into four criteria. Functional equivalence assesses whether the candidate supports the task-relevant functions of the successful

Table 12: Matched-concurrency rendering benchmark. Baseline launches one browser per rollout; Dynamic UX shares one browser across isolated contexts. Bold marks the better value; green arrows show relative change from Baseline.

C	System	Renders/min ↑	Median (ms) ↓	P95 (ms) ↓	Peak RSS (MB) ↓	Chromium processes ↓
1	Baseline	25.48	2,237.5	2,349.1	894.7	7
	Dynamic UX	**26.47** (↑3.89%)	**2,226.5** (↓0.49%)	**2,328.8** (↓0.86%)	**876.1** (↓2.08%)	**6** (↓14.29%)
4	Baseline	94.30	2,471.8	2,863.4	2,306.8	24
	Dynamic UX	**101.23** (↑7.35%)	**2,271.2** (↓8.12%)	**2,382.5** (↓16.79%)	**1,556.3** (↓32.53%)	**10** (↓58.33%)
8	Baseline	144.18	3,188.8	4,076.9	4,388.7	48
	Dynamic UX	**189.03** (↑31.11%)	**2,410.1** (↓24.42%)	**2,548.1** (↓37.50%)	**2,525.0** (↓42.47%)	**15** (↓68.75%)
16	Baseline	188.11	4,870.2	6,467.4	8,158.7	96
	Dynamic UX	**258.64** (↑37.49%)	**3,445.7** (↓29.25%)	**3,798.8** (↓41.26%)	**4,235.0** (↓48.09%)	**23** (↓76.04%)
32	Baseline	214.08	8,724.7	10,218.6	15,440.0	192
	Dynamic UX	**238.32** (↑11.32%)	**7,467.2** (↓14.41%)	**8,846.3** (↓13.43%)	**7,585.8** (↓50.87%)	**39** (↓79.69%)

reference; feature completeness checks whether the required information and controls are present; implementation correctness concerns the generated code and component behavior; and task alignment checks consistency with the user request. The reference is drawn from a successful trajectory, providing a concrete example of task completion rather than a code string that the candidate must reproduce. The online judge, `DeepSeek V4 Flash` at temperature 0, receives the query, reference and candidate code, and their initial rendered ARIA snapshots. It returns scores on these four criteria and supporting rationales. Section 5.2 gives the normalized weighted score.

Direct RL uses this initial judge score after TSX extraction, task/reference-evidence checks, structural validation, rendering, and valid judge-response checks. A failed check receives zero reward. Reference ARIA is cached before training, whereas candidate code is rendered for each rollout. This online path does not execute the full user-interaction and post-agent trajectory for every rollout; those executions provide the objective evidence used by Reward Auditor. Collecting one such outcome requires browser interaction, post-agent inference, action execution, and terminal-state evaluation. Reward Auditor applies this evidence to a sampled audit panel and amortizes the limited execution budget by converting its diagnostics into a shared reward specification used across RL rollouts.

D.9 Meta-Reward Revision Protocol

Execution-based diagnosis. Reward Auditor diagnoses and revises rewards from internal rollouts on the 15 RL training databases. `DeepSeek V4 Flash` (Xu et al., 2026) generates a Playwright program to interact with the rendered interface; after browser execution, the fixed Tool Agent consumes the submission and predicts the terminal action. The objective outcome is positive only when the action executes and the resulting database state exactly matches the target. Reward Auditor compares this outcome with $\hat{z}_i = \mathbb{I}[r_i \geq 0.5]$, where r_i is the complete reward, not the raw judge score. The auditor examines the resulting true/false positive and negative cases, aggregate reward distributions, and complete trajectories, including retrieved evidence, generated code, user submissions, final actions, and judge rationales, to diagnose failure causes and recurring errors in the evaluation criteria or score assignment.

Automated revision, review, and replay. In each round, Reward Auditor modifies the current reward specification from reward-distribution summaries, diagnosed error patterns, and individual cases, and reviews the proposed changes. It replays candidate specifications on the same stored outputs and reviews their effects on successful and unsuccessful traces before accepting a revision. Diagnosis, revision, and review require no human intervention. Holding the generated outputs fixed isolates score changes due to the reward specification. Development checks include replay of 1,618

Table 13: Problems targeted by meta-reward-guided reward revision. Revisions combine judging criteria with deterministic scoring rules.

Diagnosed problem	Reward revision	Intended effect
Plausible interfaces receive high scores despite interaction or payload defects	Revise judging criteria for actionable controls, submission wiring, payload semantics, and task fidelity	Make task-blocking defects explicit in the reward
Different failure stages collapse to the same zero score	Assign stage-aware low-reward bands	Distinguish progress within unsuccessful rollouts
Generated code uses unavailable icon exports	Apply a deterministic unsupported-icon gate	Prevent the quality score from overriding the detected defect
Label forms or associations impede control access	Apply source-based accessibility penalties	Include interaction-relevant defects in score assignment

Direct RL rollouts from 372 groups for the deterministic scoring rules and comparison of candidate judges on 1,005 paired frozen candidates.

Iterative reward revision. Audit Round 1 and Audit Round 2 are successive meta-reward-guided revisions of the training feedback. The revisions address both what the judge evaluates and how its judgments and execution evidence determine the final scalar. Table 13 summarizes the targeted problems. After automated review and replay, task-specific diagnoses are consolidated into one shared rubric and scoring specification. The accepted reward is applied across tasks and remains fixed for its RL run. Direct RL and both audited variants train independent policies from the same GUI Coder Agent SFT checkpoint, with the Tool Agent fixed; reward versions are iterated, not policy checkpoints continued along one training trajectory.

Final reward specification. The final reward combines the four-dimensional quality score with ordered failure checks and a source-based accessibility multiplier. Using the judge scores defined in Section 5.2, we compute

$$Q_i = \frac{0.35q_i^{\mathrm{fe}} + 0.35q_i^{\mathrm{fc}} + 0.15q_i^{\mathrm{ic}} + 0.15q_i^{\mathrm{ta}}}{5}.$$

This uses the initial quality-score weights without additional rescaling. For each soft check c, let $s_{i,c} \in [0,1]$ be the fraction of eligible controls or labels exhibiting that defect. The accessibility penalty and multiplier are

$$P_i = \frac{0.110s_{i,\mathrm{plain}} + 0.055s_{i,\mathrm{long}} + 0.056s_{i,\mathrm{link}}}{0.221}, \qquad A_i = 1 - 0.5\min(1, P_i).$$

The three nonzero-weight checks detect labels that are not plain strings, overlong labels (more than 5 words or 39 characters), and `Select` controls without a `labelId` linked to an `InputLabel`. Their weights are 0.110, 0.055, and 0.056, respectively; the label-collision check has weight zero and does not change P_i. Severities are proportions rather than binary flags: three defective labels among nine eligible labels give severity $1/3$. These checks form one shared scoring specification developed through the internal reward-audit process.

The following rules are evaluated from top to bottom, and the first matching branch returns the reward:

$$r_i^{(2)} = \begin{cases} 0.00, & \text{masked rollout,} \\ 0.00, & \text{TSX extraction fails,} \\ 0.02, & \text{format check fails,} \\ 0.05, & \text{code is incomplete,} \\ 0.05 + 0.05(1 - s_{i,\mathrm{icon}}), & \text{an unavailable icon is used,} \\ 0.10, & \text{rendering fails,} \\ 0.15, & \text{rendering succeeds but ARIA has fewer than 400 characters,} \\ 0.20 + 0.80Q_iA_i, & \text{otherwise.} \end{cases}$$

A rollout is masked if the judge reports an error, rendering reports a transient error, or reference evidence (including the reference ARIA snapshot) is missing. Here $s_{i,\mathrm{icon}}$ measures the proportion

of icon imports that are used but unavailable in the component package. The icon check is a gate and does not enter the soft accessibility penalty or proceed to quality scoring. Reported transient rendering errors are masked at zero; the 0.10 branch applies to unmasked rendering failures. The short-ARIA branch is a low-content heuristic. The multiplier satisfies $A_i \in [0.5, 1]$, so the quality-scoring branch lies in $[0.20, 1.00]$, above all failure-stage scores. For example, a quality score of 0.80 with no accessibility penalty gives 0.84; a perfect quality score with all three soft-check severities equal to one gives 0.60. In contrast, the initial reward assigns zero to every failed check and otherwise returns Q_i.

Intermediate reward revision. Audit Round 1 introduced the judge caps in Table 14. It is an intermediate reward version; the final reward is the ordered specification above.

Table 14: Explicit judge caps in the Audit Round 1 revision. FE, FC, IC, and TA denote functional equivalence, feature completeness, implementation correctness, and task alignment.

Detected defect	Maximum judge score
Empty/error interface or no actionable control	All four dimensions $= 0$
Missing or unreachable `submitData`	FE, TA ≤ 1
Missing required controls	FC $\leq 4, 3, 2$ when one, two, or at least half are missing
Wrong, constant, or omitted submitted values	FE, TA ≤ 3
Broken control wiring	IC ≤ 3; TA ≤ 3 when task-blocking
Harmful invented facts	TA ≤ 2
Contradicted task evidence or required-language violation	TA ≤ 3

The Audit Round 1 parser also lowers an unsupported perfect score from 5 to 4.5 and applies deterministic dimension caps to nonempty critical, payload, interaction, and visible-content defect lists. These rules illustrate how diagnosed defects become explicit evaluation criteria. Subsequent revision is guided by the remaining reward–outcome mismatches, with both false positives and false negatives considered rather than requiring every criterion to become stricter.

E Detailed Error Analysis

Evaluation cohort. We analyze one run on each of the 300 Lite tasks for `DeepSeek V4 Flash`, `Claude Sonnet 4.6`, `Claude Opus 5`, and GenUI-4B, giving 1,200 model-task runs. These single-run diagnostics are separate from the three-run aggregates in Table 2. Table 15 partitions the recorded outcomes. Across 800 failures, 416 (52.0%) have no valid UI interaction, 356 (44.5%) are classified as correct-tool calls with mismatched arguments or resulting state, and 28 (3.5%) are rejected during execution.

Table 15: Single-run outcome partition on the 300 Lite tasks under GenUI-Harness. Columns are mutually exclusive and sum to 300 for every configuration.

Configuration	No valid interact.	Wrong tool	Wrong args	Exec. rejected	Success
`DeepSeek V4 Flash`	124	0	71	7	98
`Claude Sonnet 4.6`	87	0	102	10	101
`Claude Opus 5`	82	0	124	1	93
GenUI-4B (Ours)	123	0	59	10	**108**

Diagnostic evidence. We inspect generated TSX, rendered ARIA snapshots, user-interaction traces, submitted payloads, and final actions to distinguish failure stages from underlying causes. For detailed trace review, we sample eight no-valid-interaction failures and eight argument-mismatch failures per model (64 cases in total; seed 42). These cases support the mechanisms summarized in Table 16; their frequencies are not extrapolated to all failures. The "no valid interaction" category records where execution stops, not whether the underlying cause is the interface or the simulator. We attribute a failure to generated code when the trace exposes a concrete restriction or incorrect binding that prevents the required interaction or submission.

Table 16: Observed failure mechanisms and the evidence used to identify them.

Mechanism	Trace evidence
Render failure	Compiler or runtime errors prevent an interactive page from loading.
Inaccessible control	A control's rendered accessible name or label binding does not support the attempted browser interaction.
Insufficient input controls	The available controls cannot express the required selection or value.
Incorrect submitted value	The payload diverges from the task requirement or displayed choice, with no control to correct it.

Representative cases. In a hotel-preference task, a single-select radio group cannot represent the requested selection of two hotels. In a flight-booking task, an immutable cabin-class field submits "Economy" although the task requires "Premium Economy." These restrictions arise from the generated controls and submission logic: knowing the intended selection is insufficient when the UI cannot represent or submit it. The examples show how generated code can obstruct interaction even when the user's goal is fully specified.

Interpreting state mismatches. Trace inspection also finds differences involving generated identifiers, list ordering, and free-text values. These remain failures under the reported strict database-state comparison. A state mismatch alone does not establish whether its cause lies in the UI payload, downstream tool reasoning, or state-comparison behavior; that attribution requires the corresponding execution trace and domain semantics.

F Communication-Channel Survey Protocol

Survey design and task selection. We compare text-based and generated-UI communication on the same task content to assess interaction effort and perceived latency. This comparison focuses on how reviewers communicate their preferences once an executable interface is available. The survey involves six human reviewers and 100 tasks from UI-TAU Bench Lite. All reviewers provided informed consent. Reviewers were compensated with gift cards at a rate of US$30 per hour. Participation is anonymous, and we collect no names or other identifying information. All user profiles and task records shown in the survey are synthetic, and the tasks concern routine service workflows such as booking and reservation changes rather than sensitive personal decisions. Reviewers only complete the assigned interaction and assess its usability or latency; no real customer data or private information is involved. Interface content comes from four frontier models: `GPT-5.4`, `Claude Opus 5`, `Claude Sonnet 4.6`, and `DeepSeek V4 Flash`. For each model, a task must execute the generated interface successfully and produce a valid user-simulator interaction, including a usable submission, in all three runs. We intersect the four models' eligible task sets, sample 100 tasks from this common pool, and fix one source model for each selected task. Let $v_{i,m,j} = 1$ when run j of model m on task i meets both validity conditions and 0 otherwise. With $\mathcal{M}$ denoting the four source models, eligibility requires

$$\prod_{m \in \mathcal{M}} \prod_{j=1}^{3} v_{i,m,j} = 1.$$

We denote the all-three requirement for each model by pass^3: the superscript indicates that every run must pass the interface-validity checks. This differs from Pass@3, which requires at least one successful run. Here, validity concerns interface execution and interaction, rather than final database-state exact match. The common eligibility rule keeps the channel comparison from being confounded by rendering or submission failures and applies equally to all four content sources. The resulting comparison concerns communication on consistently usable interfaces; benchmark state-EM results separately include unsuccessful executions and interactions.

Assignment and paired conditions. For each task, the six reviewers are randomly assigned with seed 42 to two equal groups: three interact through text and three through the generated UI. Each reviewer encounters every task in only one condition, giving 300 task–reviewer interactions per

condition. This design pairs conditions at the task level and prevents reviewers from carrying answers learned in one version into the other version of the same task.

Both conditions use the same user query, retrieved options, and task state from one fixed source model per task. In the UI condition, reviewers inspect the interface and provide or revise values through its controls. A fixed `DeepSeek V4 Flash` converter produces the text condition from the corresponding interface's ARIA snapshot, preserving concrete values, prefilled preferences, and unresolved fields; reviewers respond by typing. The conversion holds the available task information fixed so that the comparison concerns its presentation through text or UI, rather than differences in independently retrieved content. Neither condition exposes the hidden reference action or internal execution decisions.

Real-user interaction and stopping criterion. Reviewers can ask follow-up questions or request confirmation in either condition, with responses provided by our trained agent. An interaction ends when the agent successfully executes an action that produces the target database state. This evaluation permits the exchanges needed to complete the task rather than imposing the benchmark's fixed pre-agent–UI–post-agent sequence.

Interaction rounds. One round consists of a user message or interface submission and the corresponding agent response. Multiple field edits or clicks within one UI submission do not count as separate dialogue rounds. We count exchanges through successful task completion and average across the 300 task–reviewer interactions in each condition. The resulting means are 3.4 rounds for text and 1.2 for generated UI.

UI latency and acceptability. Interfaces are generated live during the survey. UI latency is measured from the start of interface generation through rendering and display. The three reviewers assigned to the UI condition independently assess latency acceptability for that task. A task passes when at least two of them judge its latency acceptable; the reported rate is the number of passing tasks divided by 100. This criterion is met for 85/100 tasks (85%). These outcomes describe the selected tasks with consistently valid interfaces.

G LIMITATIONS

Benchmark evaluation uses an LLM-based user simulator operating through Playwright. The simulator receives generated source and the reference goal to construct browser actions; although it must act through the rendered interface and cannot invoke task actions directly, this information may make it more capable than a user who observes only the interface. Source-blind evaluation from human-observable state is an important direction for future work. The reviewer survey covers a selected subset of consistently usable interfaces. Code execution and accessibility-tree evidence may not fully capture visual usability, including overlapping layouts or confusing arrangements. Strict database-state equality can also reject semantically equivalent outcomes that differ in generated identifiers, ordering, or free text. Adaptive interface regeneration after further clarification is an important direction for future work.

H PROMPT TEMPLATES

This appendix gives the prompt templates for the agent and simulator roles in GENUI-Harness, lightly copy-edited for readability. The Tool Agent uses two templates, one for pre-agent exploration and one for post-agent execution, matching the two stages of Section 4. Fields written as `[[name]]` are placeholders substituted before each call. `[[agent_history]]` carries the running transcript of thought, action, and observation triples, `[[domain_db]]` and `[[tool_sets]]` the sampled domain records and available tool schemas, and `[[reasoning_for_ui]]` the output of an intermediate reflection step that selects which retrieved values the interface must expose. Lines exceeding the text width are wrapped for printing. We additionally give the initial four-criterion reward judge template. Appendix D.9 specifies the final reward composition, including ordered failure checks and accessibility penalties. The meta-reward revision step uses `Claude Opus 5` to analyze reward distributions and individual errors, revise and review the criteria, and assess replay results automatically. Appendix D.9 describes this revision protocol.

Tool Agent pre-agent exploration template

```
You are a helpful tool agent. You are good at performing tasks with tools.
You will be given a task, your interaction history with the environment, and a set of tools.
      Think step by step about how to use the query tools to retrieve the information from
     the domain database that you need in order to call an action tool for the user.
You have only one chance to ask the user all the questions needed to complete the task.
First think about what you need to do, then decide which tool to use, then perform the
     action by writing its arguments.

Specifically, you should:
1. Think about the task and what is needed to complete it, and fill in the Thought field.
2. Decide which query tool to use and fill in the Action field with the tool name and the
     arguments that do not come from the databases.
3. Use this information-gathering step to prepare better clarification questions for the
     user.

# Notice:
- Always think step by step, and always use the tools to perform the action.
- Always fill in the Thought and Action fields, and never skip a step.
- Perform one step at a time, and wrap each tool call in <tool></tool> tags.
- Because of permission constraints, you may only use query-type tools on the domain
     database.
- To save space, do NOT mention user_database or domain_database in the Action field. Doing
      so breaks the program.
- The tool in the Action field must be given in full, with nothing omitted except database
     contents.
- You must NOT use tools outside the tool sets provided in the Task field. Doing so causes
     an error.
- Do not hallucinate arguments. Take them from the sample databases and from other tool
     responses.
- Only query tools are needed at this phase. Action tools are shown so you can see which
     arguments they require.
- You must wrap your thought in <thought></thought> tags and your response in ```
     agent_state ... ```.

You will be given a task, a set of tools, and the relevant databases. The task is described
      in the Task field.

# Here are sample domain and user databases for precise tool calling:
## Sample domain database:
[[domain_db]]

# Tool sets:
[[tool_sets]]

# User Id: [[user_id]]
# User Task: [[task]]

# Agent History:
[[agent_history]]

Your response must be in the following format, with all tags kept:

```agent_state
# Step [[step_num]]:
Thought:
<thought>// Your thought here</thought>
Action:
<tool>
// Your tool call here. One example is:
{
 "type": "query",
 "function": {
 "name": "get_comic_arc_info",
 "arguments": {
 "title": "Spider-Man",
 "issue": 15
 }
 }
}
</tool>
```

When this is the last step, and you are confident that the information gathered is
     sufficient and that you know what to ask the user, predict [terminate] as the tool
     action:

```agent_state
```
```

```
# Step [[step_num]]:
Thought:
<thought>// Your thought here</thought>
Action:
<tool>
[terminate]
</tool>
```

You cannot leave the Action field empty. You must provide a valid tool call with arguments. The more you gather now, the more you can succeed later, since the user would otherwise be overwhelmed by too much information.
Generate the next step of thought and action based on the above information and the Notice above:

Tool Agent post-agent execution template

You are a helpful tool agent. You are good at performing tasks with tools and database queries.
You will be given a task, your interaction history with the environment, the clarification obtained from the user, the query tools, and the action tools.
You have already gathered some information and asked the user for clarification.
Think about the task, review the exploration recorded in the history, and then perform an action tool with its arguments.

If you are going to continue gathering information, you should:
1. Think about the task and what is needed to complete it, and fill in the Thought field.
2. Decide which query tool to use and fill in the Action field with the tool name and its arguments.
3. Use this step to consolidate the existing information that the action tool will need.

Notice:
- Always think step by step, and always use the tools to perform the action.
- Always fill in the Thought and Action fields, and never skip a step.
- Perform one step at a time. Wrap a query tool call in <tool></tool> tags, and wrap the final action tool in <terminate></terminate> tags.
- Because of permission constraints, you may only use query-type tools on the domain database, and action tools for the user task.
- To save space, do NOT mention user_database or domain_database in the Action field. Doing so breaks the program.
- The tool in the Action field must be given in full, with nothing omitted except database contents.
- You must NOT use tools outside the tool sets provided in the Task field, and you must not invent external tools. Doing so causes an error.
- Do not hallucinate arguments. Take them from the sample databases and from other tool responses.
- You must wrap your thought in <thought></thought> tags and your response in ``` agent_state ... ```.

You will be given a task, a set of tools, and the relevant databases. The task is described in the Task field.

Here are sample domain and user databases for precise tool calling:
Sample domain database:
[[domain_db]]

Tool sets:
[[tool_sets]]

User Id: [[user_id]]
User Task: [[task]]

Agent History:
[[agent_history]]

Your response must be in the following format, with all tags kept:

```agent_state
# Step [[step_num]]:
Thought:
<thought>// Your thought here</thought>
Action:
<tool>
// Your tool call here. One example is:
{
 "type": "query",
 "function": {
```

```
 "name": "get_comic_arc_info",
 "arguments": {
 "title": "Spider-Man",
 "issue": 15
 }
 }
}
</tool>
```

```
When the information gathered is sufficient and you are ready to complete the task, emit
     the final action tool wrapped in <terminate></terminate> tags:

```agent_state
# Step [[step_num]]:
Thought:
<thought>// Your thought here</thought>
Action:
<terminate>
{
 "name": // predicted action tool name
 "type": "action",
 "arguments": {
 // arguments for the action tool
 }
}
</terminate>
```

You cannot leave the Action field empty. You must provide a valid tool call with arguments.
[[final_step_alert]]
Generate the next step of thought and action based on the above information and the Notice
     above:
```

GUI Coder Agent interface generation template

```
# Role
You are an AI assistant with expertise in performing actions for users as well as coding.
     In particular, you are a capable React and TypeScript programmer able to generate rich,
      dynamic, and interactive UIs.

# Task
You must generate a Material UI React component that gathers from the user the information
     needed to complete a task. Carefully consider the user task, the interaction history
     with the environment, and the additional information that the user must provide. Then
     generate a dynamic interactive UI using TypeScript, Material UI, and React so that the
      user can supply the missing information required to complete the task.
You have only this **one chance** to interact with the user.

# Notice
- You may only use Material UI components or CSS-styled HTML elements to build the UI.
- You must use a single file named `Component.tsx` that exports the component to be
     displayed, e.g. `export function Component() { ... }`.
- No pictures or images are allowed, only interactive elements.
- The layout should be clear and user-friendly, with explicit instructions and labels. You
     are encouraged to use diverse elements.
- The component is hosted inside an existing application, so return a self-contained
     component that can be imported and used. The host controls the component's background
     and position.
- The layout must be responsive, since the screen dimensions are unknown. Assume roughly
     1280x720.
- You may select important data from the interaction history and display it to make the
     user's task easier.
- You may split the implementation into several components, but you must export only a
     single function `Component`.

# Gathering feedback from the user
- Select and use the right data from the interaction history. Do not expect the user to
     provide all the data you need.
- To save the user input you must call the `submitData` function, which accepts a JSON
     object and sends it to the server. It has the interface:
```typescript
async function submitData(data: object): Promise<void>;
```

To use the function you must import it as
```typescript
```
```

````
import { submitData } from "./api"
```

# Example output
Return your response as a single `tsx` code block, such as:
```tsx
import React from "react";
import { submitData } from "./api";
// ... additional imports if needed

function MyButton() {
...
}

function MyField() {
...
}

export function Component() {
 return <>
 <MyField/>
 <MyButton/>
 </>
}
```

# User Task: [[task]]

# Interaction History:
[[agent_history]]

# Reasoning:
[[reasoning_for_ui]]
Now generate the code following the reasoning above and the Notice:
````

User Simulator interface operation template

````
# Role
You are role-playing as a human USER interacting with an AI agent to complete a specific
     task. Your goal is to generate realistic and natural responses that a user might give
     in this scenario.
The AI agent interacts with you through a UI, so you will be asked to fill in information
     in that UI.

# Your task
Given the source code of a UI, its ARIA snapshot, and a goal, generate Playwright code that
      interacts with the UI to achieve your goal.

# Information about the UI
## UI code:
```tsx
[[tsx_code]]
```

## ARIA snapshot:
```aria_snapshot
[[aria_snapshot]]
```

# Your specific task tool call (INVISIBLE TO THE AGENT):
```tool_calling
<tool>
[[original_action_tool_calling]]
</tool>
```

# Notice:
- Write a single function named interact_with_page that uses Playwright to interact with
     the UI according to your task. It must have exactly this name, and you must aim to
     complete the task fully.
- The function must accept one argument of type `Page` and be enclosed in ```python ... ```
      tags.
- The tool call above is your end goal. Understand its details and write code that
     plausibly reaches that goal through the UI.
- Import every module and function you need from Playwright, such as `Page`. Without the
     imports the code will not run.
````
````

```
- The Playwright version is 1.53.0. Every function and method you use must be compatible
    with it.
- You do not need to wait for elements to load, since the UI is already loaded.

# Example output

'''python
import asyncio
from playwright.async_api import Page
...

async def interact_with_page(page: Page):
 # Your interaction code here. Keep the function name and format exactly the same.
 await ...
'''

Generate your Playwright code function here:
```

Initial reward judge scoring template

```
You are an expert code reviewer evaluating TypeScript code implementations.

## REVIEW CONTEXT

You are provided with:
1. **USER TASK**: The original task description or requirement that the code should
    accomplish
2. **GROUND TRUTH CODE**: A reference implementation that correctly solves the task
3. **PREDICTED CODE**: The implementation to be evaluated
4. **EXECUTION RESULT**: Both codes have been executed and rendered successfully, with
    accessibility trees provided

## REVIEW TASK

Your goal is to evaluate how well the PREDICTED CODE accomplishes the USER TASK compared to
     the GROUND TRUTH CODE.

Evaluate the predicted code on these dimensions:

### 1. FUNCTIONAL EQUIVALENCE (Most Important)
Does the predicted code implement the same core functionality as the ground truth?
- Are the same user interactions supported (buttons, forms, inputs)?
- Does it handle the same events and actions?
- Does it produce similar outputs or UI states?
- Does it call the same or equivalent APIs/functions with comparable inputs/outputs?

### 2. FEATURE COMPLETENESS
Are all key features from the ground truth present in the predicted code?
- Check for major UI components (forms, buttons, modals, lists, etc.)
- Check for state management patterns and derived state
- Check for validation logic and user guidance
- Check for API integrations
- Check for conditional rendering and user feedback
- Check for accessibility features (labels, roles, keyboard flow) that are visible in the
    ARIA snapshot

### 3. IMPLEMENTATION CORRECTNESS
Based on the execution results, does the predicted code work correctly?
- The ARIA snapshot shows what UI elements are actually rendered
- Verify that key components from the ground truth appear in the predicted code's output
- Check that the predicted code successfully renders without errors
- If the predicted code includes multi-step or conditional flows, confirm the initial step/
    state is correctly rendered and actionable

### 4. TASK ALIGNMENT
Does the predicted code solve the user's original task?
- Even if implementation differs from GT, does it achieve the same goal?
- Are there any critical missing pieces that prevent task completion or create incorrect/
    unsafe behavior?

## EVALUATION PRINCIPLES

**FOCUS ON FUNCTIONALITY, NOT IMPLEMENTATION DETAILS**
- Two different implementations can be functionally equivalent
- Don't penalize for different variable names, styling approaches, or code structure
- DO penalize for missing features, broken functionality, or incomplete implementations
```

```
**USE THE EXECUTION RESULTS**
- The ARIA snapshots show what actually renders
- Use this to verify that key UI elements are present and functional
- If a component is in the code but not in the ARIA snapshot, it may not be working

**BE FAIR BUT STRICT**
- Give credit for correct implementations, even if they differ from GT
- Don't give credit for partially implemented features that won't work
- Missing critical features should significantly impact the score

**SCORING GUIDELINES**

For each dimension, assign an INTEGER score from 0 to 5:
- **5**: Excellent - Functionally equivalent to GT, all key features present and working
    correctly
- **4**: Good - Most features present and working, only minor differences or missing minor
    features
- **3**: Adequate - Core functionality present but missing several features or has notable
    issues
- **2**: Poor - Some functionality present but major features missing or not working
    correctly
- **1**: Very Poor - Minimal functionality, missing most features or fundamentally
    incorrect approach
- **0**: Failed - No meaningful implementation or completely wrong approach

Use the FULL RANGE of scores. Do not cluster all scores at 3-4. If code is excellent, give
    5. If code is broken, give 0.

## USER TASK

{user_task}

## GROUND TRUTH CODE

'''typescript
{ground_truth_code}
'''

## GROUND TRUTH EXECUTION RESULT

The ground truth code was executed successfully. Here is its accessibility tree:

'''
{gt_aria_snapshot}
'''

## PREDICTED CODE

'''typescript
{predicted_code}
'''

## PREDICTED CODE EXECUTION RESULT

The predicted code was executed successfully. Here is its accessibility tree:

'''
{pred_aria_snapshot}
'''

## OUTPUT FORMAT

Return ONLY a valid JSON object with this exact schema (no markdown, no extra text):

{{
  "functional_equivalence": <integer 0-5>,
  "feature_completeness": <integer 0-5>,
  "implementation_correctness": <integer 0-5>,
  "task_alignment": <integer 0-5>,
  "reasoning": "<concise explanation of the scores, mentioning key features present/missing,
       comparing ARIA snapshots, and noting any critical differences>"
}}

Example (do not copy literally):
{{"functional_equivalence": 5, "feature_completeness": 4, "implementation_correctness": 5,
    "task_alignment": 4, "reasoning": "Predicted code implements all core functionality:
    form with email/phone inputs, validation on submission, API call to submitData
    endpoint, success/error message display. ARIA snapshots show both codes render similar
     UI structure with form inputs and submit button. Minor difference: GT includes a
```

```
    loading spinner during submission (visible in GT ARIA as 'status: loading...') which
    is missing in predicted code, hence feature_completeness is 4 instead of 5. Otherwise
    functionally equivalent."}}

CRITICAL REQUIREMENTS:
- Return valid JSON only - no markdown code fences, no explanatory text before or after
- All dimension scores MUST be integers in the range [0, 5]
- Use the full range of scores - don't cluster around 3-4
```