SLICE2FIX: DISCOVERING ERROR SLICES FOR TARGETED GUI AGENT REPAIR

Anonymous authors
Paper under double-blind review

ABSTRACT

Graphical-user-interface (GUI) agents are increasingly used to operate mobile applications, yet they still fail whole groups of tasks that share a common property. Existing ways of learning from execution feedback stay at the level of single tasks or written reports: they repair the tasks that failed, keep the tasks the agent already solves, or summarise the failures into weakness reports, and none of them turns the property behind the failures into new tasks without relying on a stronger model to solve them. To address these limitations, we propose SLICE2FIX, which repairs a GUI agent by its systematic failures rather than by its failed tasks. Its core idea is to define a systematic failure as an *error slice*, a group of tasks described by a combination of controllable task features, so that the object that names a failure also specifies the tasks that repair it, and a reference executor written for each task family solves them, closing the loop from diagnosis to repair. Across three strong open-source end-to-end GUI agents at the 8B scale, slice-directed repair raises success on the repaired slices by 20 to 29 percentage points. With the same amount of data, untargeted tasks from the same generator, applications and operations gain 13.1 points there on GUI-Owl-1.5-8B, against 20.0 for slice-directed repair. The repaired agents also improve on AndroidWorld, including templates that appear in no training data. We also release FEATBENCH, a feature-labelled benchmark of 974 mobile tasks.

1 INTRODUCTION

Graphical-user-interface (GUI) agents operate software from natural-language instructions (Qin et al., 2025; Ye et al., 2025), but their execution remains unreliable: an agent that completes most tasks of an application can still fail on others (Rawles et al., 2025). Closing such gaps requires training trajectories that cover the situations in which the agent fails, which existing datasets cover thinly (Sun et al., 2025; Liu et al., 2026b). In practice, the next training data therefore come from the agent's own execution feedback: which tasks it fails, and what to collect for them.

Existing methods turn this feedback into training data in three ways. *Failure-case repair* obtains corrected demonstrations for the tasks the agent failed (Qin et al., 2025), and so covers only the failures that happened to be observed. *Success-only rejection sampling fine-tuning (RFT)* samples several trajectories per task and keeps the successful ones (Yuan et al., 2023), and so yields nothing on tasks the agent never solves, as Figure 1b and c illustrate. More recently, weakness-driven methods take the tasks that a stronger teacher solves but the agent fails, summarise a verifier's diagnoses of these failures into weakness reports, and write new tasks from them (Kim et al., 2026). Their repair data are the teacher's rollouts, yet open models several times larger fail these tasks about as often, so the teacher must be far larger. All three act on individual tasks or their descriptions, and none turns the property the failed tasks share into tasks solvable without a stronger model.

The failed tasks often do share such a property. In AndroidWorld, an agent completes an expense-entry task if the required category is visible on screen but fails it consistently if the category can only be reached by horizontal scrolling (Rawles et al., 2025). We call such a group a *systematic failure*: a whole set of tasks that share one property and on which the agent fails far more often than on its other tasks. For example, asked to delete an entry given by its name and amount, an agent succeeds when no other entry carries that name, but once entries with the same name are present it matches by name alone and deletes the wrong one, as Figure 3b shows.

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.

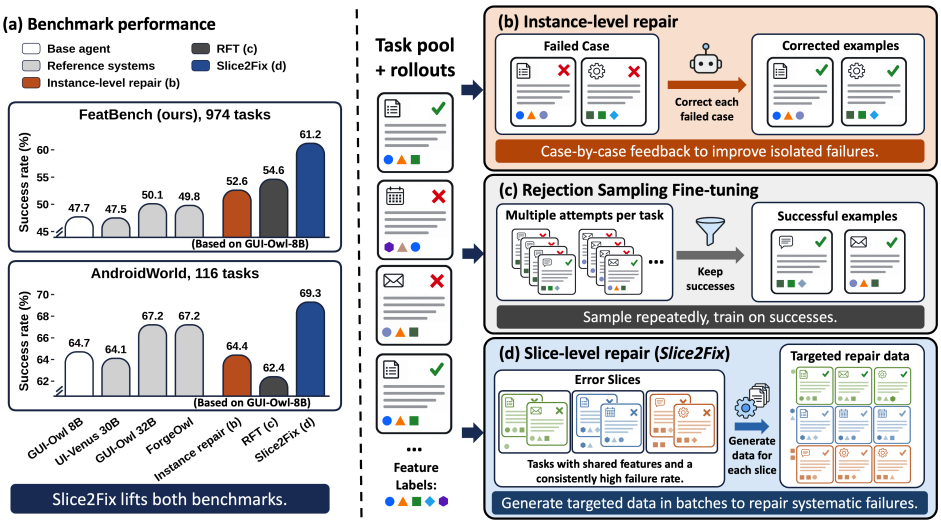


Figure 1: **Benchmark performance and repair strategies.** (a) We compare the success rates of GUI-Owl-1.5-8B on FEATBENCH and AndroidWorld under each repair strategy. SLICE2FIX lifts both benchmarks, while the other strategies gain less on FEATBENCH and none on AndroidWorld. (b) Instance-level repair obtains corrected examples for failed tasks. (c) Rejection sampling fine-tuning (RFT) retains successful trajectories from repeated attempts. (d) SLICE2FIX discovers error slices and generates targeted repair data for each slice.

Repairing an agent by its systematic failures raises two challenges: the failures must be found in the agent's execution records, and each must be turned into new tasks with correct demonstrations. Finding them is harder in a GUI agent than in a vision model. Error-slice discovery finds and repairs the subsets on which a vision model fails by searching over the features of single images with fixed labels (Eyuboglu et al., 2022; Chen et al., 2023; Zhang et al., 2023). A GUI agent's failure is instead the outcome of a long multimodal trajectory and offers no per-sample features that both describe a failure and specify new tasks. To address these challenges, we propose SLICE2FIX, shown in Figure 1d, which defines a systematic failure as an *error slice*: a group of tasks that share labels of controllable task features.

The labels serve both ends of the loop. Grouping the agent's execution records by them exposes the slices on which it fails far more often than elsewhere, without any model diagnosing a trajectory. And because a slice is a setting of the features, new tasks in it can be generated in any number and solved by a reference executor written once per task family, so the repair data are no longer bounded by the failures that were observed. The same setting with one feature switched off gives matched control tasks for the slice. Every task is built from designed features, as in behavioural testing (Ribeiro et al., 2020), and carries their labels by construction. The combinations of them that form an agent's systematic failures are mined from its own execution records.

We summarize our contributions as follows:

- **We propose SLICE2FIX, which makes an agent's systematic failures, not its individual failed tasks, the unit of repair.** Each systematic failure is represented as an *error slice*, the tasks that share a combination of labels of controllable task features. Because these features are settings of the task generator, a slice that describes a failure also specifies the tasks that repair it.
- **We show that the failures of the agents we study are largely systematic.** On three strong open-source end-to-end GUI agents at the 8B scale, 12 slices hold 59.6% of all failures on a feature-controlled pool of mobile tasks and are shared by all three agents.
- **We show that repairing by slice outperforms repairing by instance on FEATBENCH.** Across the three agents, slice-directed repair raises success on new tasks under the repaired slices by 20.0 to 28.7 percentage points and on the whole of FEATBENCH by 13.4, 6.8

and 11.7, the best of the three data strategies. It also improves AndroidWorld for all three agents and AndroidLab for two of them, with the same 448 repair trajectories.

- **We release FEATBENCH, a feature-controlled benchmark of 974 mobile tasks.** Every task carries the feature labels it was built under and a reference trajectory, and it includes structurally matched control tasks that switch a slice's feature off.

2 RELATED WORK

GUI agents. Native end-to-end GUI agents (Qin et al., 2025; Wang et al., 2025a; Ye et al., 2025; Xu et al., 2026; Zhou et al., 2025; Gu et al., 2025; Venus Team et al., 2026; Zhou et al., 2026) are trained on recorded demonstrations and refined with reinforcement learning against environment reward (Wang et al., 2025a; Bai et al., 2024; Qi et al., 2025), and because demonstrations are costly, another line synthesises tasks and trajectories from exploration, tutorials or the agent's own interactions (Sun et al., 2025; Xu et al., 2025b; Su et al., 2025). Even the strongest open models still fail a sizeable share of tasks on interactive benchmarks with programmatic success checks (Rawles et al., 2025; Xu et al., 2025a).

Data loops from execution feedback. A growing body of work turns an agent's own execution feedback into its next training data. The simplest loop is rejection sampling fine-tuning (RFT): the agent attempts each task several times and is fine-tuned on its successful trajectories (Yuan et al., 2023). Failure-case repair instead supplies correct behaviour for the tasks the agent failed: an expert relabels the failing steps (Ross et al., 2011), reflective traces correct the agent's own mistakes (Qin et al., 2025), or recovery trajectories are synthesised from the error states the agent reaches (Bu et al., 2026). Failed trajectories can also be kept as negative samples during fine-tuning (Wang et al., 2025b). MobileForge (Liu et al., 2026b) adapts an agent to mobile applications without annotation, generating tasks and evaluating rollouts in real applications and turning trajectory outcomes, step-level process feedback and corrective hints into policy-optimisation updates. More recently, LearnWeak (Kim et al., 2026) goes beyond single instances: on tasks that a teacher solves and the student fails, it summarises a verifier's diagnoses into written weakness reports, generates new tasks conditioned on them, and trains the student on the teacher's trajectories.

Error-slice discovery. Existing error-slice discovery methods, developed mainly for vision models, find coherent subsets of data on which a model fails far more often than elsewhere by searching over metadata features (Chung et al., 2019; Sagadeeva & Boehm, 2021), clustering errors in an embedding space (Eyuboglu et al., 2022), or diagnosing them through language and interactive inspection (Zhang et al., 2023; Cabrera et al., 2023). HiBug further uses named slices to select or generate repair data (Chen et al., 2023; 2025b), and recent work turns the profiled weaknesses of language models and agents into training data (Khan et al., 2025; Zeng et al., 2025; Kang et al., 2026; Wang et al., 2026). For GUI agents, DataScope varies the number of targets and of structurally matched distractors inside hand-designed templates (Liu et al., 2026a), much as behavioural testing probes one capability at a time (Ribeiro et al., 2020). It measures features chosen in advance rather than mining their combinations, and the analysis stops before the diagnosis becomes training data.

In summary, existing data loops for GUI agents work either task by task, covering only the failures that were observed, or through prose reports of weaknesses, which rely on a stronger teacher to solve the new tasks, while error-slice discovery defines weaknesses over data that already exist and relies on per-sample features that a GUI agent's long, multimodal trajectories do not provide. SLICE2FIX brings slices to GUI agents by defining a systematic failure as an error slice over the controllable features of a task generator, which can be mined, checked against control tasks, and expanded into repair data.

3 METHOD

Given the executions of GUI agents on tasks with programmatic success checks, SLICE2FIX finds their systematic failures and turns them into training data that repairs the property they share.

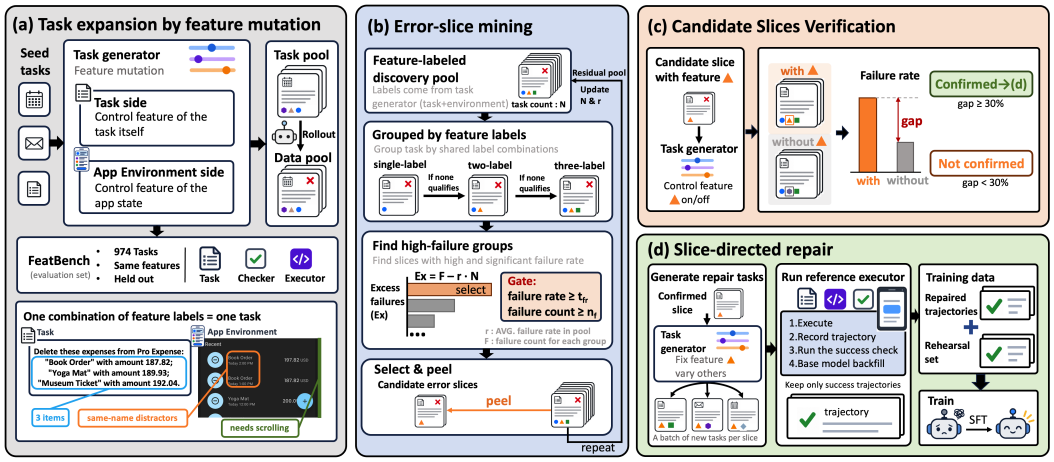


Figure 2: **Overview of SLICE2FIX.** (a) A task generator expands seed tasks by mutating features, so every task carries its labels. (b) Mining finds the label groups with the most excess failures, peeling each one before searching again. (c) A candidate slice is checked on new tasks with its feature on and off. The 30% shown is illustrative, and the thresholds are given in Appendix C. (d) Slices are turned into batches of new tasks with reference trajectories for fine-tuning.

3.1 OVERVIEW

The overall pipeline of SLICE2FIX is shown in Figure 2. First, we group the factors that affect whether a task succeeds into task-side and environment-side factors and design adjustable features for each. *Feature mutation* then expands seed tasks into a large pool of tasks that carry their labels by construction, on which we roll out the agents. Next, we repeatedly pick out and remove the label group that fails far more often than the rest of the pool, which gives candidate error slices that can further be checked against control tasks with their feature switched off. Finally, we generate new tasks under the slices, record a correct trajectory on each with a reference executor, and fine-tune the agent on them together with a rehearsal set of its own successes.

Notation. Let $\Phi = \{\phi_1, \ldots, \phi_m\}$ be the task features, each ϕ_j taking labels in a set $\mathcal{V}_j$, so that every task t carries a label vector $\boldsymbol{\phi}(t) = (\phi_1(t), \ldots, \phi_m(t)) \in \mathcal{V}_1 \times \cdots \times \mathcal{V}_m$. The discovery pool consists of a set of tasks $\mathcal{T}$ and the trajectories $\mathcal{D} \subseteq \mathcal{T} \times \mathcal{A}$ that a set of agents $\mathcal{A}$ produces on them, each judged by the task's success check as $y_{t,a} \in \{0, 1\}$. For $S \subseteq \mathcal{T}$, we write $N(S) = |\{(t, a) \in \mathcal{D} : t \in S\}|$ and $F(S) = |\{(t, a) \in \mathcal{D} : t \in S,\ y_{t,a} = 0\}|$ for its numbers of trajectories and failures, and $f(S) = F(S)/N(S)$ for its failure rate. A label combination $L = \{(\phi_j, v_j)\}_{j \in J}$ fixes a label $v_j \in \mathcal{V}_j$ on each feature in a set $J \subseteq \{1, \ldots, m\}$. Its *slice* is the set of tasks that carry all of these labels,

$$S_L \ = \ \{\, t \in \mathcal{T} : \phi_j(t) = v_j \ \text{ for all } j \in J \,\}, \tag{1}$$

and $|L| = |J|$ is its size. Finally, $G(L)$ denotes the generator's distribution over new tasks that carry the labels in L while their other features vary.

3.2 FEATURE-CONTROLLED TASK EXPANSION

The first stage of SLICE2FIX builds the data that mining reads: a large pool of tasks with known factors that can be changed one at a time, so that a slice holds enough failures to pass the gates and its failures can be traced to the factor it names. Existing interactive benchmarks provide neither: they hold at most a few hundred task templates (116 in AndroidWorld, 138 in AndroidLab), and each task changes several properties at once, so a failure cannot be pinned on any one of them. We therefore group the factors that affect whether a task succeeds into two sides and design adjustable *features* for each: task-side features describe the request, such as the number of targets or the source of the values it needs, and environment-side features describe the application state, such as the similarity of the distractors or the display of an attribute as an icon only. Each feature $\phi_j \in \Phi$ takes one of a few *labels* in $\mathcal{V}_j$, all listed in Table 4. Starting from a small set of seed tasks, we apply *feature mutation*, which assigns new labels to their features, and the generator of each task family (one operation in

one application), built on the environment utilities of AndroidWorld, writes each setting as a task, deriving its initial state and success check from the same labels. A task's labels are therefore exactly the settings the generator wrote it from: every task t carries its label vector $\phi(t)$ by construction, without any annotation, and any one feature can be switched off with everything else held fixed.

Because every task is generated with a known structure, we implement one *reference executor* per task family that solves its tasks through the interface without calling a model. It validates every generated task, so that every failure in the pool can be attributed to the agent, and records the correct trajectories used for repair. Instance-level repair and SLICE2FIX record their demonstrations with the same executors, whose implementation Appendix F describes, and any other source of correct trajectories could serve the pipeline as well.

We run the agents $\mathcal{A}$ to be repaired on the expanded tasks $\mathcal{T}$ and judge each resulting trajectory with the task's own success check. The tasks and their judged trajectories $\mathcal{D}$ form the *discovery pool*. Ours holds $|\mathcal{T}| = 1{,}025$ tasks attempted by $|\mathcal{A}| = 3$ strong open-source 8B GUI agents, with $N(\mathcal{T}) = 3{,}066$ trajectories of which $F(\mathcal{T}) = 1{,}294$ fail. We write our evaluation set, FEATBENCH, with the same generator from new seeds and hold it out from mining and from all training. The discovery pool spans 20 task families in 10 applications with 623 distinct feature settings, and FEATBENCH spans 9 families in 6 applications with 763.

3.3 MINING ERROR SLICES

With every trajectory in the discovery pool carrying the labels of its task, we look for the label combinations L such that the agents fail systematically on their slices S_L as defined in Eq. 1. The candidates are many and overlap heavily, so the search has to pick out a few that each carry many failures at a high rate without counting the same failure twice. Candidates can further be checked on new tasks. The set $\mathcal{A}$ may contain a single agent or several. We pool three agents to find the failures they share, and the same procedure applies to one agent's records alone.

We call our search *harvest and peel*. It starts from the feature-labelled discovery pool and groups its tasks by their labels, beginning with single labels. Writing R for the pool that remains, initially the whole discovery pool, and $r = f(R)$ for its failure rate, we first rank each group S of tasks still in R. Ranking by failure rate would put a handful of tasks failing at 100% ahead of a large group failing at 80%, so we rank by *excess failures*,

$$\mathrm{Ex}(S) \;=\; F(S) \;-\; r\,N(S), \tag{2}$$

the failures a group carries beyond the pool's own rate for a set of its size. Excess alone would in turn favour large groups that fail only moderately, and a small group can fail often by chance, so a group is eligible only if it passes two gates,

$$f(S) \;\geq\; t_{\mathrm{fr}}, \qquad F(S) \;\geq\; n_f, \tag{3}$$

a high failure rate and enough failures. We select the top-ranked eligible group as a slice.

Overlapping groups would then count the same failures many times, so we *peel* the selected slice from the pool, $R \leftarrow R \setminus S$, recompute r and search again. Slices are thus disjoint and their coverage adds up, and since peeling removes whole tasks, a slice takes all three agents' attempts on its tasks with it.

Finally, the number of label combinations grows quickly with their size, and most pairs or triples would only restate a failure that one of their labels already explains. We therefore move on to pairs of labels only once no single-label group passes the gates, then to triples, each round working on the pool the previous round left, so that a larger combination is selected only for failures that no smaller one explains. The search runs up to three labels. Extending it to four or five labels returns the same slices on our pool. Algorithm 1 in Appendix B gives the full procedure and the full list of slices.

A check on new tasks filters out slices whose failures come from a hard application or task family rather than from the slice's feature. To check a candidate, we generate a batch of tasks in the slice and a control batch with the feature switched off, which together make up the held-out FEATBENCH, and confirm the slice if the gap in failure rate between the two batches exceeds the 95th percentile of the gap under no effect, with the thresholds given in Appendix C. The check uses the base agents' results on these tasks, and Appendix E shows that the gains hold when the slices are selected on one half of FEATBENCH and measured on the other. A slice that the check rejects is not used for repair.

3.4 SLICE-DIRECTED REPAIR

Existing loops draw repair data from the tasks the agent happened to fail, so the data are bounded by the failures that were observed and aimed at those tasks rather than at the property they share.

A slice, however, is more than a list of failed tasks. Its labels are settings of the generator, so the slice is also a specification of tasks that exhibit the failure, and we construct the repair data from the slice itself in the three steps of Figure 2d. First, for each slice S_L the generator draws a batch of repair tasks $\mathcal{T}_L^{\text{rep}} \sim G(L)$, which keep the labels in L fixed and vary the other features over their labels, with seeds disjoint from every test task. The result is a batch of repair tasks that all exhibit the property the slice names while the rest of the task changes from one task to the next, so the data target the property itself, and the batch can be made as large as needed. Second, the reference executor of each task's family runs every repair task and records each step, the screenshot and the action, keeping a trajectory only if the task's success check accepts its final state. We record the demonstrations with executors rather than distil them from a larger open agent because the failures are shared across scales: GUI-Owl-1.5-32B-Instruct and UI-Venus-1.5-30B-A3B succeed on the target tasks of Table 1 about as often as GUI-Owl-1.5-8B (44.7% and 43.6%, against 43.6%). The text accompanying each action is a templated one-sentence description for GUI-Owl. For MAI-UI and UI-Venus, which have a reasoning field in their output formats, it is written by the base agent itself as a post-hoc rationale after it is shown the action the executor took, so the action is never the base agent's own choice. Third, we fine-tune each agent on these repair trajectories together with a *rehearsal set* of trajectories it solved by itself, mixed in to prevent forgetting as in continual learning (Rebuffi et al., 2017; Chaudhry et al., 2019; Rolnick et al., 2019), with its composition given in Appendix E.

4 EXPERIMENTS

4.1 SETUP

Models. We repair three strong open-source end-to-end GUI agents at the 8B scale, GUI-Owl-1.5-8B-Instruct (Xu et al., 2026), MAI-UI-8B (Zhou et al., 2025) and UI-Venus-1.5-8B (Venus Team et al., 2026). For reference, we also evaluate GUI-Owl-1.5-32B-Instruct, UI-Venus-1.5-30B-A3B and ForgeOwl-8B (Liu et al., 2026b) without training.

Benchmarks and metrics. We report the task success rate on FEATBENCH, whose 974 tasks are held out from mining and training and form the batches of the check, AndroidWorld (Rawles et al., 2025) (116 tasks) and AndroidLab (Xu et al., 2025a) (138 tasks), each under its own success check. The target tasks of FEATBENCH are its 275 tasks under the slices that the check confirms. AndroidWorld and AndroidLab are averaged over three runs, and all differences are in percentage points.

Baselines. *Instance-level repair* records one correct trajectory, with the same reference executors, on the failed tasks of the discovery pool, and *success-only RFT* (Yuan et al., 2023) keeps the base agent's successful trajectories from repeated attempts on the training tasks. All methods add their data to the agent's rehearsal set and share the fine-tuning recipe.

Implementation. Mining uses $t_{\text{fr}} = 75\%$ and $n_f = 10$ in Eq. 3. Every model is fine-tuned once with LoRA for one epoch, and evaluation uses greedy decoding. Appendix E gives the full hyperparameters, step budgets and hardware.

4.2 DISCOVERED ERROR SLICES

Failures concentrate in a few shared slices. Mining returns the 12 slices of Table 6, most defined by a single label, and Figure 3a shows one each with one, two and three labels. They hold 28% of the pool tasks but 60% of its failures, where random label groups of the same sizes hold only 40%. The failure rate inside them is 89%, against 24% outside. Each agent fails far more often inside every slice than over the pool, so the slices describe properties that current 8B agents share, not quirks of one model.

Slices isolate the property that fails. As Figure 3b shows, the near-duplicate slice isolates a single property on a single template: the agent fails only when near-duplicates are present. A high failure rate can also come from a hard task family, and the check on new tasks filters out such slices: the

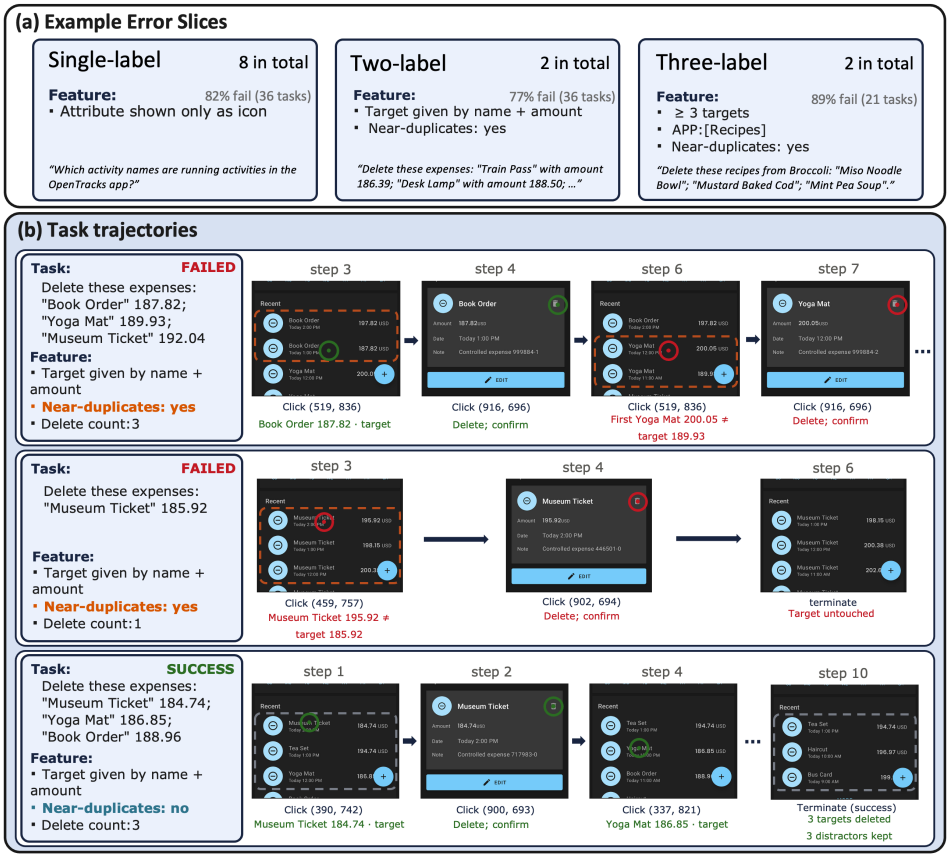


Figure 3: **Error slices.** (a) We show three error slices defined by one, two and three feature labels, together with their failure rates over the three base agents in the discovery pool. (b) On three tasks from the template of the two-label slice, GUI-Owl-1.5-8B deletes the wrong entries when near-duplicates are present and succeeds when they are absent.

date-range slice in Tasks fails on all 36 of its pool attempts, but exact-date queries in the same app fail almost as often on new tasks (87 of 90 against 90 of 90), so the difficulty lies in finding items by date, not in the range, and neither this slice nor the other one the check rejects receives repair data.

The slices also mark failures on tasks we did not write. Labelling AndroidWorld with the same features, blind to outcomes as described in Appendix D, the tasks carrying the labels of the five repaired slices it covers fail 52% to 95% of the time for the 7–8B agents, against 34% to 37% for the other tasks, while tasks under a rejected slice fail no more often (34% against 38%). GUI-Owl-1.5-32B-Thinking shows the same pattern on its own.

4.3 Main Results

Slice2Fix outperforms instance-level data loops on FeatBench. As shown in Table 1, on the target tasks it raises success by 20.0% and 24.0% on GUI-Owl-1.5-8B and MAI-UI-8B, against 12.6% and 9.9% for instance-level repair and 14.9% and -2.2% for success-only RFT, and on UI-Venus-1.5-8B it matches instance-level repair (28.7% against 29.5%). On FeatBench as a whole it gives the best result for every agent (61.2, 59.3 and 48.8), although on UI-Venus-1.5-8B the margin over instance-level repair is one point.

Repair does not come at the cost of other tasks. The other FeatBench tasks do not decline for any agent ($+10.8\%$, $+0.1\%$ and $+4.9\%$), so the repair data do not trade general skills for the targeted ones. On the external benchmarks the gains are smaller: Slice2Fix improves AndroidWorld for all three agents ($+4.6\%$, $+1.1\%$ and $+3.1\%$) and AndroidLab for two ($+5.1\%$ and $+5.8\%$), leaving

Table 1: **Success rates (%) of three 8B GUI agents repaired by SLICE2FIX and two baselines.** FEATBENCH is run once, and AndroidWorld and AndroidLab are averaged over three runs. The number of training trajectories includes the rehearsal set. Subscripts give the change from the base model, and bold marks the best result in each column per base model.

Model	#Training traj.	FEATBENCH			AndroidWorld	AndroidLab
		Target (275)	Other (699)	All (974)	All (116)	All (138)
Reference						
ForgeOwl-8B	≈8.6k	55.3	47.6	49.8	67.2	34.1
UI-Venus-1.5-30B-A3B	–	43.6	49.0	47.5	64.1	39.9
GUI-Owl-1.5-32B-Instruct	–	44.7	52.2	50.1	67.2	36.2
GUI-Owl-1.5-8B						
Base	0	43.6	49.4	47.7	64.7	28.3
Instance-level repair	772	56.2 +12.6	51.1 +1.8	52.6 +4.8	64.4 −0.3	31.9 +3.6
Success-only RFT	1,189	58.5 +14.9	53.1 +3.7	54.6 +6.9	62.4 −2.3	30.4 +2.2
SLICE2FIX (ours)	858	**63.6** +20.0	**60.2** +10.8	**61.2** +13.4	**69.3** +4.6	**33.3** +5.1
MAI-UI-8B						
Base	0	31.6	60.7	52.5	55.7	46.4
Instance-level repair	633	41.5 +9.9	59.2 −1.4	54.4 +1.9	**59.5** +3.7	46.4 +0.0
Success-only RFT	1,108	29.5 −2.2	60.7 +0.0	51.8 −0.6	57.5 +1.7	**48.3** +1.9
SLICE2FIX (ours)	834	**55.7** +24.0	**60.7** +0.1	**59.3** +6.8	56.9 +1.1	46.4 +0.0
UI-Venus-1.5-8B						
Base	0	5.1	49.7	37.1	64.9	32.6
Instance-level repair	669	**34.6** +29.5	53.1 +3.4	47.8 +10.7	66.4 +1.4	**42.8** +10.2
Success-only RFT	784	8.0 +2.9	54.2 +4.5	41.2 +4.1	67.0 +2.0	39.1 +6.5
SLICE2FIX (ours)	759	33.8 +28.7	**54.6** +4.9	**48.8** +11.7	**68.0** +3.1	38.4 +5.8

Table 2: **Slice mining under different gates (a) and search strategies (b).** In (a), one gate changes at a time while the other stays at our setting (*). In (b), random draws, for each of our slices, label groups of the same sizes (mean of 20 draws), top failure rate ranks groups by failure rate alone, and w/o peeling does not remove covered tasks.

	(a) Mining gates									(b) Search strategies			
	t_{fr}					n_f				Random	Top failure rate	w/o peeling	SLICE2FIX
	60%	70%	75%*	80%	90%	5	10*	20	30				
Slices found	13	13	12	15	10	13	12	11	8	12	14	15	12
Failures covered (%)	79.1	63.8	59.6	56.2	40.5	60.1	59.6	56.8	51.3	39.6	16.7	39.9	59.6
Failure rate in slices (%)	68.0	84.2	89.2	91.4	96.5	89.1	89.2	89.1	89.6	64.0	100.0	77.7	89.2

it unchanged for MAI-UI-8B. None of these gains is significant on its own, but their direction is consistent: in paired comparisons on AndroidWorld, more tasks improve than decline for every agent (15 against 8, 13 against 9 and 14 against 7). Pooled over the three agents, AndroidWorld tasks with near-duplicates or a new named target gain more than tasks without any repaired label ($+11.1\%$ and $+6.3\%$ against $+2.8\%$), and the other three repaired slices gain at most 1.7 points more. All methods share the same rehearsal set, which alone does not raise AndroidWorld in Table 3. On the 39 AndroidWorld templates that appear in no training data, SLICE2FIX improves all three agents ($+6.0\%$, $+5.1\%$ and $+2.4\%$) and is the best of the three strategies for two of them, as Table 8 shows. The AndroidWorld gains are thus not explained by templates seen in training. AndroidLab, from which no training task of any kind is drawn, shows the same direction.

4.4 SENSITIVITY TO THE MINING GATES

As shown in Table 2(a), we vary the two gates of mining, t_{fr} and n_f, one at a time. Raising t_{fr} to 90% lifts the failure rate inside the slices from 89.2% to 96.5% but cuts their coverage from 59.6% to 40.5%. Lowering it to 60% raises coverage to 79.1% but lets the failure rate fall to 68.0%. The gate n_f mainly decides whether small slices are kept: too low a value admits a slice confined to a

Table 3: **Effect of the training data on GUI-Owl-1.5-8B.** Fractions of the repair data are subsampled within each of the 25 repair batches, rounding up, and untargeted tasks come from the same generator with random features. Success rates (%) are on the 475-task subset of FEATBENCH, AndroidWorld and AndroidLab, and subscripts give the change from the base model.

Training data	#Training traj.	FEATBENCH			AndroidWorld	AndroidLab
		Target (275)	Other (200)	All (475)	All (116)	All (138)
None (base agent)	0	43.6	49.5	46.1	64.7	28.3
Rehearsal set only	410	49.5 +5.8	52.0 +2.5	50.5 +4.4	62.9 −1.8	31.2 +2.9
Repair data only	448	80.4 +36.7	71.0 +21.5	76.4 +30.3	42.2 −22.5	34.8 +6.5
Rehearsal set + 448 untargeted tasks	858	56.7 +13.1	56.5 +7.0	56.6 +10.5	67.2 +2.5	32.6 +4.3
Rehearsal set + $\frac{1}{4}$ repair data	534	53.8 +10.2	54.0 +4.5	53.9 +7.8	66.4 +1.7	34.1 +5.8
Rehearsal set + $\frac{1}{2}$ repair data	634	59.3 +15.6	53.5 +4.0	56.8 +10.7	68.1 +3.4	34.8 +6.5
Rehearsal set + all repair data (SLICE2FIX)	858	63.6 +20.0	58.0 +8.5	61.3 +15.2	69.3 +4.6	33.3 +5.1

single task family, and too high a value drops real small slices. Our setting, $t_{\text{fr}} = 75\%$ and $n_f = 10$, thus balances the two measures.

4.5 ABLATION

Targeted versus untargeted data. The gains of SLICE2FIX could come from targeting the slices, or merely from adding generated tasks with correct trajectories. To separate the two, we replace the repair data with 448 untargeted tasks from the same generator, applications and operations, with their features sampled at random instead of fixed to a slice, and report the results in Table 3. With the same amount of data, they raise target success by 13.1%, against 20.0% for SLICE2FIX ($p = 0.005$, McNemar test), while on the other tasks and the external benchmarks SLICE2FIX stays at or above them (69.3 against 67.2 on AndroidWorld, 33.3 against 32.6 on AndroidLab) without a significant difference. The gap follows how many training tasks fall under each slice: sampled at random, only 40 of the 261 untargeted deletion tasks carry the near-duplicate labels, and on the slices of an attribute shown only as an icon and of counts or sums of the matches, where the untargeted data hold fewer tasks (31 and 23 instead of 70 and 36), the gains shrink from 13.9% and 19.4% to 2.8% and 0.0%. Knowing which applications and operations fail thus explains part of the gain, and generating tasks under the labels of each slice supplies the rest.

Slice search. As shown in Table 2(b), both the excess-failure ranking and peeling are needed. Without peeling, the search keeps returning variants of the same label and covers only 39.9% of the failures, about as much as random slices of the same size (39.6%). Ranking by failure rate alone favours small groups that always fail and covers only 16.7%. With both, the slices cover 59.6% of the failures at a failure rate of 89.2%.

Amount of repair data. As shown in Table 3, target success grows steadily with the amount of repair data: over the rehearsal set alone (+5.8%), the first 124 repair trajectories add 4.4%, the next 100 a further 5.5% and the last 224 another 4.4%, so half of the repair data already gains more than instance-level repair with 362 trajectories (12.6%), while the other tasks and AndroidWorld vary by at most 4.5%.

Role of the rehearsal set. Trained on the repair data alone, the agent reaches 76.4% on the FEATBENCH subset but falls from 64.7% to 42.2% on AndroidWorld. Adding the rehearsal set keeps AndroidWorld above the base agent while retaining more than half of the gain on the target tasks.

5 CONCLUSION

SLICE2FIX turns the recurring failures of a GUI agent into training data. Because an error slice is written in the features of the task generator, the description that tells us where an agent fails also tells the generator which tasks to produce for it. On three 8B GUI agents, a few such slices account for most of the failures, and data generated from them raise success on the target tasks by 20 to 29 percentage points. Given the same budget, untargeted tasks from the same applications and operations gain less on these targets, which points to the slices themselves as the source of the additional gain.

AI USE STATEMENT

In this work, we used generative AI tools to polish and translate the writing, and to assist in implementing parts of the experimental pipeline, including the task generators, the reference executors and the evaluation and analysis scripts, in designing experiments and in analysing results. We did not use them to formulate or prove mathematical claims, which are not applicable to this work. The repair tasks are written by the task generator and their actions recorded by the executors, and the rehearsal set and the RFT data are the agents' own trajectories. Beyond these trajectories, the only model-generated text in the training data is the reasoning field of MAI-UI-8B and UI-Venus-1.5-8B, which each base agent writes for the executor's actions as part of the method. All reported results come from running the agents in emulators and scoring them with each benchmark's own success check. All AI-assisted work was reviewed and tested by the authors, who take full responsibility for the content. Models that are part of the method or the evaluation, such as the agents under study and the model judge that AndroidLab uses for some tasks, are described in the corresponding sections.

ETHICS STATEMENT

This work complies with the ICLR Code of Ethics. All experiments run in Android emulators on open-source applications whose contents are synthetic, so no personal data, real accounts or human subjects are involved. The repaired agents act only inside these emulators. As with any GUI agent, deploying such models on real devices could lead to unintended actions, and we release them for research use.

REPRODUCIBILITY STATEMENT

Appendix A describes the task generator and its features, Appendix B the mining procedure with its pseudocode, Appendix C the confirmation protocol, and Appendix E the training data, hyperparameters, evaluation settings and hardware. FEATBENCH with its success checks and base-agent results, the discovery pool and the code for mining and evaluation are available at `https://anonymous.4open.science/r/Slice2Fix/`. We will release the task generator, the reference executors, the reference trajectories and judge test cases of FEATBENCH, the code for confirmation, the repair and rehearsal training data, and the fine-tuned adapters with the paper. All base models are publicly available, the external benchmarks are evaluated with their official harnesses, and all evaluations use greedy decoding.

REFERENCES

Hao Bai, Yifei Zhou, Mert Cemri, Jiayi Pan, Alane Suhr, Sergey Levine, and Aviral Kumar. DigiRL: Training in-the-wild device-control agents with autonomous reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Tianpeng Bu, Xin Liu, Qihua Chen, Hao Jiang, Shurui Li, Hongtao Duan, Lu Jiang, Lulu Hu, Bin Yang, and Minying Zhang. Recovering policy-induced errors: Benchmarking and trajectory synthesis for robust GUI agents. In *International Conference on Machine Learning (ICML)*, 2026. arXiv:2605.29447.

Ángel Alexander Cabrera, Erica Fu, Donald Bertucci, Kenneth Holstein, Ameet Talwalkar, Jason I. Hong, and Adam Perer. Zeno: An interactive framework for behavioral evaluation of machine learning. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI)*, 2023.

Arslan Chaudhry, Marcus Rohrbach, Mohamed Elhoseiny, Thalaiyasingam Ajanthan, Puneet K. Dokania, Philip H. S. Torr, and Marc'Aurelio Ranzato. On tiny episodic memories in continual learning. *arXiv preprint arXiv:1902.10486*, 2019.

Jingxuan Chen, Derek Yuen, Bin Xie, Yuhao Yang, Gongwei Chen, Zhihao Wu, Li Yixing, Xurui Zhou, Weiwen Liu, Shuai Wang, Kaiwen Zhou, Rui Shao, Liqiang Nie, Yasheng Wang, Jianye Hao, Jun Wang, and Kun Shao. SPA-Bench: A comprehensive benchmark for SmartPhone agent evaluation. In *International Conference on Learning Representations (ICLR)*, 2025a. arXiv:2410.15164.

Muxi Chen, Yu Li, and Qiang Xu. HiBug: On human-interpretable model debug. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, 2023.

Muxi Chen, Chenchen Zhao, and Qiang Xu. HiBug2: Efficient and interpretable error slice discovery for comprehensive model debugging. In *International Conference on Learning Representations (ICLR)*, 2025b.

Yeounoh Chung, Tim Kraska, Neoklis Polyzotis, Ki Hyun Tae, and Steven Euijong Whang. Slice finder: Automated data slicing for model validation. In *IEEE 35th International Conference on Data Engineering (ICDE)*, pp. 1550–1553, 2019. doi: 10.1109/ICDE.2019.00139.

Sabri Eyuboglu, Maya Varma, Khaled Saab, Jean-Benoit Delbrouck, Christopher Lee-Messer, Jared Dunnmon, James Zou, and Christopher Ré. Domino: Discovering systematic errors with cross-modal embeddings. In *International Conference on Learning Representations (ICLR)*, 2022.

Zhangxuan Gu, Zhengwen Zeng, Zhenyu Xu, Xingran Zhou, Shuheng Shen, Yunfei Liu, Beitong Zhou, Changhua Meng, Tianyu Xia, Weizhi Chen, Yue Wen, Jingya Dou, Fei Tang, Jinzhen Lin, Yulin Liu, Zhenlin Guo, Yichen Gong, Heng Jia, Changlong Gao, Yuan Guo, Yong Deng, Zhenyu Guo, Liang Chen, and Weiqiang Wang. UI-Venus technical report: Building high-performance UI agents with RFT. *arXiv preprint arXiv:2508.10833*, 2025.

Hangoo Kang, Tarun Suresh, Jon Saad-Falcon, and Azalia Mirhoseini. TRACE: Capability-targeted agentic training. *arXiv preprint arXiv:2604.05336*, 2026.

Zaid Khan, Elias Stengel-Eskin, Jaemin Cho, and Mohit Bansal. DataEnvGym: Data generation agents in teacher environments with student feedback. In *International Conference on Learning Representations (ICLR)*, 2025.

Suji Kim, Kangsan Kim, and Sung Ju Hwang. Learn from weaknesses: Automated domain specialization for small computer-use agents. *arXiv preprint arXiv:2605.28775*, 2026.

Quyu Kong, Xu Zhang, Zhenyu Yang, Nolan Gao, Chen Liu, Panrong Tong, Chenglin Cai, Hanzhang Zhou, Jianan Zhang, Liangyu Chen, Zhidan Liu, Steven Hoi, and Yue Wang. MobileWorld: Benchmarking autonomous mobile agents in agent-user interactive and MCP-augmented environments. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers) (ACL)*, pp. 6142–6167, 2026.

Juyong Lee, Taywon Min, Minyong An, Dongyoon Hahm, Haeone Lee, Changyeon Kim, and Kimin Lee. Benchmarking mobile device control agents across diverse configurations. In *Conference on Lifelong Learning Agents (CoLLAs)*, 2025. arXiv:2404.16660.

Chen Liu, Ling Chen, Hanzhang Zhou, Xu Zhang, Quyu Kong, Panrong Tong, Wenhao Wang, Xin Yu, Steven Hoi, and Yue Wang. What memory do GUI agents really need? From passive records to active task-driving states. *arXiv preprint arXiv:2606.31612*, 2026a.

Guangyi Liu, Pengxiang Zhao, Gao Wu, Yiwen Yin, Mading Li, Liang Liu, Congxiao Liu, Zhang Qi, Mengyan Wang, Liang Guo, Jiangning Zhang, and Yong Liu. MobileForge: Annotation-free adaptation for mobile GUI agents with hierarchical feedback-guided policy optimization. *arXiv preprint arXiv:2606.19930*, 2026b.

Zehan Qi, Xiao Liu, Iat Long Iong, Hanyu Lai, Xueqiao Sun, Wenyi Zhao, Yu Yang, Xinyue Yang, Jiadai Sun, Shuntian Yao, Tianjie Zhang, Wei Xu, Jie Tang, and Yuxiao Dong. WebRL: Training LLM web agents via self-evolving online curriculum reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2025.

Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, Wanjun Zhong, Kuanye Li, Jiale Yang, Yu Miao, Woyu Lin, Longxiang Liu, Xu Jiang, Qianli Ma, Jingyu Li, Xiaojun Xiao, Kai Cai, Chuang Li, Yaowei Zheng, Chaolin Jin, Chen Li, Xiao Zhou, Minchao Wang, Haoli Chen, Zhaojian Li, Haihua Yang, Haifeng Liu, Feng Lin, Tao Peng, Xin Liu, and Guang Shi. UI-TARS: Pioneering automated GUI interaction with native agents. *arXiv preprint arXiv:2501.12326*, 2025.

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

Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H. Lampert. iCaRL: Incremental classifier and representation learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2001–2010, 2017.

Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. Beyond accuracy: Behavioral testing of NLP models with CheckList. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2020.

David Rolnick, Arun Ahuja, Jonathan Schwarz, Timothy Lillicrap, and Gregory Wayne. Experience replay for continual learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 32, 2019.

Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 15 of *Proceedings of Machine Learning Research*, pp. 627–635, 2011.

Svetlana Sagadeeva and Matthias Boehm. SliceLine: Fast, linear-algebra-based slice finding for ML model debugging. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*, pp. 2290–2299, 2021. doi: 10.1145/3448016.3457323.

Hongjin Su, Ruoxi Sun, Jinsung Yoon, Pengcheng Yin, Tao Yu, and Sercan Ö. Arık. Learn-by-interact: A data-centric framework for self-adaptive agents in realistic environments. In *International Conference on Learning Representations (ICLR)*, 2025.

Qiushi Sun, Kanzhi Cheng, Zichen Ding, Chuanyang Jin, Yian Wang, Fangzhi Xu, Zhenyu Wu, Chengyou Jia, Liheng Chen, Zhoumianze Liu, Ben Kao, Guohao Li, Junxian He, Yu Qiao, and Zhiyong Wu. OS-Genesis: Automating GUI agent trajectory construction via reverse task synthesis. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL)*, 2025.

Venus Team, Changlong Gao, Zhangxuan Gu, Yulin Liu, Xinyu Qiu, Shuheng Shen, Yue Wen, Tianyu Xia, Zhenyu Xu, Zhengwen Zeng, Beitong Zhou, Xingran Zhou, Weizhi Chen, Sunhao Dai, Jingya Dou, Yichen Gong, Yuan Guo, Zhenlin Guo, Feng Li, Qian Li, Jinzhen Lin, Yuqi Zhou, Linchao Zhu, Liang Chen, Zhenyu Guo, Changhua Meng, and Weiqiang Wang. UI-Venus-1.5 technical report. *arXiv preprint arXiv:2602.09082*, 2026.

Haoming Wang, Haoyang Zou, Huatong Song, Jiazhan Feng, Junjie Fang, Junting Lu, Longxiang Liu, Qinyu Luo, Shihao Liang, Shijue Huang, Wanjun Zhong, Yining Ye, et al. UI-TARS-2 technical report: Advancing GUI agent with multi-turn reinforcement learning. *arXiv preprint arXiv:2509.02544*, 2025a.

Renxi Wang, Xudong Han, Yixuan Zhang, Timothy Baldwin, and Haonan Li. NAT: Enhancing agent tuning with negative samples. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers) (NAACL)*, pp. 7385–7398, 2025b. doi: 10.18653/v1/2025.naacl-long.378. arXiv version titled "Learning From Failure: Integrating Negative Examples when Fine-tuning Large Language Models as Agents" (arXiv:2402.11651).

Ziyi Wang, Yuxuan Lu, Yimeng Zhang, Qun Liu, Chen Luo, Jiri Gesi, Hanqing Lu, Yisi Sang, Manling Li, Jing Huang, and Dakuo Wang. SENTINEL: Failure-driven reinforcement learning for training tool-using language model agents. *arXiv preprint arXiv:2606.12908*, 2026.

Haiyang Xu, Xi Zhang, Haowei Liu, Junyang Wang, Zhaozai Zhu, Shengjie Zhou, Xuhao Hu, Feiyu Gao, Junjie Cao, Zihua Wang, Zhiyuan Chen, Jitong Liao, Qi Zheng, Jiahui Zeng, Ze Xu, Shuai Bai, Junyang Lin, Jingren Zhou, and Ming Yan. Mobile-Agent-v3.5: Multi-platform fundamental GUI agents. *arXiv preprint arXiv:2602.16855*, 2026.

Yifan Xu, Xiao Liu, Xueqiao Sun, Siyi Cheng, Hao Yu, Hanyu Lai, Shudan Zhang, Dan Zhang, Jie Tang, and Yuxiao Dong. AndroidLab: Training and systematic benchmarking of Android autonomous agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers) (ACL)*, pp. 2144–2166, 2025a.

Yiheng Xu, Dunjie Lu, Zhennan Shen, Junli Wang, Zekun Wang, Yuchen Mao, Caiming Xiong, and Tao Yu. AgentTrek: Agent trajectory synthesis via guiding replay with web tutorials. In *International Conference on Learning Representations (ICLR)*, 2025b.

Jiabo Ye, Xi Zhang, Haiyang Xu, Haowei Liu, Junyang Wang, Zhaoqing Zhu, Ziwei Zheng, Feiyu Gao, Junjie Cao, Zhengxi Lu, Jitong Liao, Qi Zheng, Fei Huang, Jingren Zhou, and Ming Yan. Mobile-Agent-v3: Fundamental agents for GUI automation. *arXiv preprint arXiv:2508.15144*, 2025.

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

Zhiyuan Zeng, Yizhong Wang, Hannaneh Hajishirzi, and Pang Wei Koh. EvalTree: Profiling language model weaknesses via hierarchical capability trees. In *Conference on Language Modeling (COLM)*, 2025.

Yuhui Zhang, Jeff Z. HaoChen, Shih-Cheng Huang, Kuan-Chieh Wang, James Zou, and Serena Yeung. Diagnosing and rectifying vision models using language. In *International Conference on Learning Representations (ICLR)*, 2023.

Hanzhang Zhou, Xu Zhang, Panrong Tong, Jianan Zhang, Liangyu Chen, Quyu Kong, Chenglin Cai, Chen Liu, Yue Wang, Jingren Zhou, and Steven Hoi. MAI-UI technical report: Real-world centric foundation GUI agents. *arXiv preprint arXiv:2512.22047*, 2025.

Hanzhang Zhou, Panrong Tong, Xu Zhang, Quyu Kong, Chenglin Cai, Tianyu Xia, Gongjie Zhang, Jianan Zhang, Long Li, Long Chen, Lei Wang, Gaole Dai, Pengxiang Li, Liangyu Chen, Yue Wang, and Steven Hoi. Qwen-UI-Agent technical report: Toward next-generation real-world centric foundation GUI agents. *arXiv preprint arXiv:2607.28227*, 2026.

A TASK GENERATOR AND FEATURES

Table 4 lists the features the generator controls, grouped into task-side and environment-side features. The discovery pool spans 20 task families, each one operation in one application: deletion in Broccoli, Pro Expense and Simple Calendar, queries in Tasks and OpenTracks, nine text operations in Markor (prepend, append, delete, rename, rewrite, replace, replace a segment, merge two notes and merge three notes), playlist creation in VLC, moving files in Files, replying in Simple SMS Messenger, and selecting, navigating and computing in Chrome. FEATBENCH covers nine of them, namely the three deletion families, the two query families and four Markor operations (prepend, rename and the two merges). For mining, every label the generator writes is a candidate, including each value of the number of targets, while a numeric feature computed from the instruction text contributes the labels "nonzero" and "at least the median of the nonzero values", and a boolean feature the label "true". The same features also describe tasks we did not generate: 94 of the 116 AndroidWorld tasks carry at least one non-default label, and the remaining 22 are single-step toggles and one-shot actions with nothing to vary. Table 5 compares FEATBENCH with existing interactive mobile-agent benchmarks.

Table 4: **Features of the task generator.** Task-side features describe the request, and environment-side features describe the application state. Each row gives a feature and its labels in the discovery pool. Mechanism features (✓) name a property that can cause failures, and every slice contains at least one of them. The other features give magnitudes such as counts and positions and can only narrow a slice. The application and the operation are recorded as context, so they can restrict a slice but never form one alone.

Feature	Labels	Mechanism
Task side		
Target identification	name; name with amount, description or date; file name; attribute; position; content clue	✓
Selection rule	listed targets; all items; one or two filtering conditions; date or numeric range; count or sum of the matches	✓
Information source	instruction; data stored in the app; current screen; partly looked up in the app; copied within the app; another app	✓
Content transfer	none; into a new target; between objects	✓
Target creation	none; a new named target	✓
Text-edit scope	none; prepend; append; replace a segment; rewrite all; compose new	✓
Insertion at an exact point	yes; no	✓
Ordered composition	yes; no	✓
Number of targets	1 to 10	
Required order	sequence length; order required or not	
Verbatim text	none; present; 17 tokens or more	
Catalogue items in the instruction	none; present	
Environment side		
Distractor similarity	unrelated; shared stem; one token apart; one field apart; same name with a secondary cue; identical	✓
Icon-only attribute	yes; no	✓
Entry through a long-press menu	yes; no	✓
Number of distractors	0 to 14	
Scrolling needed	yes; no	
Target position in the list	row index	
Visible candidates	1 to 14	

Table 5: **Comparison with interactive mobile-agent benchmarks.** Difficulty levels indicate whether a benchmark grades its tasks by difficulty, which FEATBENCH does by the length of the reference trajectory. Reference trajectories are one solution trajectory per test task, and evidence is the on-screen text that each step relies on. Judge test cases are released correct and near-miss final states with labels and screenshots, which can be used to test success checkers and model judges.

Benchmark	#Tasks	Difficulty levels	Extensible via	Training data	Reference trajectories	Judge test cases
B-MoCA (Lee et al., 2025)	131	✗	device settings	✓	–	✗
AndroidWorld (Rawles et al., 2025)	116	✓	parameter values	✗	–	✗
AndroidLab (Xu et al., 2025a)	138	✗	–	✓	length only	✗
SPA-Bench (Chen et al., 2025a)	340	✓	–	✗	human	✗
MobileWorld (Kong et al., 2026)	201	✗	–	✗	–	✗
DataScope (Liu et al., 2026a)	96	✓	target/distractor counts	✗	–	✗
FEATBENCH (ours)	974	✓	task features	✓	executor + evidence	✓

Algorithm 1 Harvest-and-peel mining of error slices.

Input: tasks $\mathcal{T}$ with label vectors $\phi(t)$; judged trajectories $\mathcal{D}$; gates t_{fr}, n_f; maximum size K ($K = 3$)
Output: disjoint error slices $\mathcal{S} = \{S_{L_1}, S_{L_2}, \dots\}$

$R \leftarrow \mathcal{T}$; $\mathcal{S} \leftarrow \emptyset$
for $k = 1, \dots, K$ **do** ▷ single labels, then pairs, then triples
  loop ▷ harvest at size k
    $r \leftarrow f(R)$
    $\mathcal{C} \leftarrow \{\, S_L \cap R : |L| = k \,\}$ ▷ distinct features, at least one a mechanism feature (Table 4)
    $\mathcal{C} \leftarrow \{\, S \in \mathcal{C} : f(S) \geq t_{\text{fr}},\ F(S) \geq n_f \,\}$ ▷ gates, Eq. 3
    if $\mathcal{C} = \emptyset$ **then break**
    end if
    $S^\star \leftarrow \arg\max_{S \in \mathcal{C}}\ F(S) - r\,N(S)$ ▷ excess, Eq. 2
    $\mathcal{S} \leftarrow \mathcal{S} \cup \{S^\star\}$; $R \leftarrow R \setminus S^\star$ ▷ peel
  end loop
end for
return $\mathcal{S}$

B FULL SLICE TABLE

Limiting the search to single labels, each optionally restricted to one application, covers 46.9% of the failures in the discovery pool, and adding pairs covers 54.5%. Allowing four or five labels returns exactly the slices of the three-label search, which cover 59.6%.

Table 6 gives the complete slice atlas, including feature combinations, pool statistics and the slices that the check rejects.

C CONFIRMATION PROTOCOL DETAILS

For a candidate slice with n_{on} feature-on and n_{off} control tasks, Δ is the difference between the failure rates of the two batches, pooled over the three agents. X^+ is the 95th percentile of Δ when both batches fail at 50%, the case of largest variance, and X^- the 5th percentile of Δ when the feature-on batch fails at 57% and the control batch at 42%, an effect of 15%. Both are obtained by simulating the two batches at their actual sizes with a task-level beta-binomial model, so that the three agents' outcomes on one task are correlated (intra-task correlation 0.48), with 20,000 draws, and both are fixed before any outcome is read. A slice is confirmed if $\Delta \geq X^+$ and rejected if $\Delta \leq X^-$.

Rejected slices. The two slices the check rejects fail often in the pool for reasons other than their feature. For *two filtering conditions*, tasks with the feature fail 56% of the time and their controls with a single condition 62%: in the query families of Tasks and OpenTracks both sides fail almost always (93–94% against 98–100%), and in the deletion family of Pro Expense both rarely fail (5%

Table 6: **Error slices found in the discovery pool, ordered by the failures they hold.** The twelve slices hold 771 of the 1,294 failed trajectories of the three 8B agents (59.6%), while the failure rate over the whole pool is 42.2%. In the table, k is the number of features, n the number of tasks, and Failures the number of failed trajectories that a slice holds. A slice with $k > 1$ is shown one feature at a time, with the failure rate and the number of tasks after each added feature, and its last feature is the one that makes the tasks fail systematically. The three agent columns give each agent's failure rate inside a slice, against 43.3%, 33.4% and 50.0% over the whole pool. Brackets mark an app restriction. Slices marked with † were rejected by the check on new tasks described in Appendix C, and ✓ in the Repair column marks the slices for which repair data were generated. In the terms of Table 4, near-duplicates are distractors with the same name and a secondary cue, and names one letter apart are distractors one token apart. The slice of a target known only by its body text holds only four tasks in the pool and received no repair data.

Features, added in order	k	Fail. (%)	n	Failures	Fail. by agent (%)			Repair
					GUI-Owl	MAI-UI	UI-Venus	
new named target	1	96.5	66	191	100.0	89.4	100.0	✓
attribute shown only as an icon	1	82.4	36	89	83.3	91.7	72.2	✓
two filtering conditions†	1	83.8	35	88	97.1	54.3	100.0	–
target given by name and amount	2	36.1	179					
+ near-duplicates		**76.9**	36	83	80.6	63.9	86.1	✓
segment inside the text	1	93.6	26	73	96.2	92.3	92.3	✓
option hidden in a long-press menu	1	97.2	24	70	100.0	91.7	100.0	✓
[Recipes]	3	30.1	215					
+ $\geq$3 targets		44.3	91					
+ near-duplicates		**88.9**	21	56	76.2	90.5	100.0	✓
[Tasks]	2	72.2	54					
+ date-range filter†		**100.0**	12	36	100.0	100.0	100.0	–
count or sum of the matches	1	94.4	12	34	100.0	83.3	100.0	✓
[Expense]	3	28.5	200					
+ 3 targets		38.3	54					
+ names one letter apart		**83.3**	10	25	80.0	70.0	100.0	✓
several values from another app	1	83.3	6	15	83.3	83.3	83.3	✓
target known only by its body text	1	91.7	4	11	100.0	75.0	100.0	–

against 12%). For the *date-range filter*, range queries fail 85% of the time and exact-date queries 98% over Tasks and OpenTracks together, and in Tasks alone 90 of 90 against 87 of 90. In both cases the difficulty lies in the task family, such as locating items by date in lists that do not show full dates, and not in the feature the slice names. Neither slice receives repair data.

D Feature Labeling of External Benchmarks

We label the 116 AndroidWorld tasks with the task-side and environment-side features of Table 4, from the task source and instruction and before reading any outcome. A label that rests on a random draw or on facts not visible in the source is marked unsure, and such tasks are left out for that feature. Outcomes come from nine agent and framework combinations with three runs per task. Table 7 compares the tasks with and without each label for the slices that AndroidWorld covers. AndroidWorld has no task with an icon-only attribute or a date-range query, so those slices cannot be compared. The numbers are descriptive: each label covers 5 to 18 tasks. The nine combinations

Table 7: **Failure rates on AndroidWorld by slice label and model scale.** Each row compares the AndroidWorld tasks that carry a slice's label with those that do not, over three runs per task. For the near-duplicate slices, the row uses their shared near-duplicate label. The 7–8B columns pool eight agent and framework combinations and give the number of combinations in which the labelled tasks fail more often, and the last two columns give GUI-Owl-1.5-32B-Thinking alone. Slices marked with † were rejected by the check on new tasks.

		7–8B agents (8 combinations)			GUI-Owl-1.5-32B-Thinking	
Slice	Tasks	With / without (%)	Difference	Combinations	With / without (%)	Difference
Near-duplicates	5	58.3 / 34.4	+23.9	7/8	46.7 / 30.4	+16.3
Several values from another app	7	95.2 / 34.0	+61.3	8/8	95.2 / 29.7	+65.6
Count or sum of the matches	8	52.1 / 36.6	+15.5	8/8	41.7 / 33.0	+8.6
Two filtering conditions†	7	33.9 / 37.9	−4.0	3/8	28.6 / 33.9	−5.4
New named target	14	66.7 / 33.7	+33.0	8/8	69.0 / 28.8	+40.3
Option hidden in a long-press menu	18	51.6 / 34.3	+17.3	8/8	48.1 / 30.7	+17.5

are GUI-Owl-1.5-8B-Instruct and GUI-Owl-1.5-8B-Thinking, each with its official harness and with Mobile-Agent-v3.5; GUI-Owl-1.5-32B-Thinking with its official harness; GUI-Owl-7B with its official harness and with Mobile-Agent-v3; and MAI-UI-8B and UI-Venus-1.5-8B with their official harnesses.

E Training and Evaluation Details

Hyperparameters. Every model is fine-tuned once with LoRA (rank 32, α=64, dropout 0.05) on all linear layers of the language model, with the vision encoder and projector frozen, for one epoch at learning rate 2×10^{-5} (cosine decay, 3% warm-up, weight decay 0.1), an effective batch of 8 and sequences of up to 16,384 tokens. Each agent is run through its native harness and prompt. Evaluation uses greedy decoding, a 60-step budget on FeatBench and the default budgets of AndroidWorld and AndroidLab.

Baselines. Instance-level repair records failed tasks of the discovery pool, which come from the same generator as the test tasks. It leaves out any failed task whose generator settings coincide with those of a test task, or whose identifier or seed appears in the test or training data. The repair data and the untargeted tasks of the ablation are new instances drawn from the generator and checked for disjoint identifiers and seeds under exactly the same rules, so the untargeted control compares targeted and untargeted data under identical conditions. Success-only RFT receives more trajectories than Slice2Fix (779 against 448 for GUI-Owl-1.5-8B). ForgeOwl-8B is trained from GUI-Owl-1.5-8B with the MobileForge pipeline.

Evaluation tasks. The check on new tasks runs the base agents on the tasks of FeatBench, and its outcome decides which slices become targets. No FeatBench task, seed or trajectory enters training, every test task is a new instance with its own seed and content, and all methods are evaluated on the same tasks against the same base runs, so the selection of target slices affects them alike. To keep the tasks that select the targets apart from those that measure the repair, we also split

FEATBENCH in half within each of its task groups, the sets of tasks built from one template with the same main features, rerun the check on one half and measure the gains on the target tasks of the other half, and then swap the halves. On either half the check confirms the same target slices and none of the rejected ones, and it does so in all of 200 further random splits. On the held-out halves, SLICE2FIX gains 22.6% and 17.4% on GUI-Owl-1.5-8B, 21.8% and 25.2% on MAI-UI-8B, and 31.4% and 26.1% on UI-Venus-1.5-8B, in line with the full set, and its margins over instance-level repair, RFT and, on GUI-Owl-1.5-8B, the untargeted tasks keep the same pattern.

Hardware. Fine-tuning runs on NVIDIA A800 80 GB GPUs on a server with two 32-core Intel Xeon Platinum 8336C CPUs and 1 TB of memory. A full fine-tuning run takes about nine hours on four GPUs, and the Android emulators run on the CPUs.

Training data. The rehearsal set of each agent consists of its own successful trajectories on discovery-pool tasks and on a designated rehearsal group of 77 AndroidWorld templates, plus, for GUI-Owl, 89 simple tasks in two testbed applications. It holds 410 trajectories for GUI-Owl-1.5-8B (194 on pool tasks, 127 on AndroidWorld and 89 simple tasks), 386 for MAI-UI-8B (269 and 117) and 311 for UI-Venus-1.5-8B (174 and 137), and the other 39 AndroidWorld templates appear in no training data. Table 8 splits the AndroidWorld results by this exposure. SLICE2FIX adds the same 448 repair trajectories for every agent, spread over the repaired slices marked in Table 6: 216 deletions among near-duplicates, 70 queries on an attribute shown only as an icon, 36 count-or-sum queries, 36 tasks with several values from another app, and 90 text or playlist edits in Markor and VLC; the first group serves the three near-duplicate slices and the last the slices of a new named target, a segment inside the text and an option hidden in a long-press menu. The repair tasks are generated in 24 batches of 18 tasks and one of 16, each fixing the labels of one slice within one application. The slices the check rejected receive no repair data. Instance-level repair adds trajectories recorded with the same executors on the discovery-pool tasks that agent failed (362, 247 and 358), and success-only RFT adds that agent's successful samples, at most two per task (779, 722 and 473). All models are fine-tuned with the same recipe.

Table 8: **AndroidWorld success by training exposure.** We report the success rate (%) of each model on the 39 templates that appear in no training data, on the 77 templates of the rehearsal group and on all 116 templates, averaged over the runs of Table 1.

Model	Unseen (39)	Rehearsal (77)	All (116)
GUI-Owl-1.5-8B			
Base	67.5	63.2	64.7
Instance-level repair	68.4	62.3	64.4
Success-only RFT	66.7	60.2	62.4
SLICE2FIX	73.5	67.1	69.3
MAI-UI-8B			
Base	59.0	54.1	55.7
Instance-level repair	58.1	60.2	59.5
Success-only RFT	62.4	55.0	57.5
SLICE2FIX	64.1	53.2	56.9
UI-Venus-1.5-8B			
Base	67.5	63.6	64.9
Instance-level repair	67.5	65.8	66.4
Success-only RFT	71.8	64.5	67.0
SLICE2FIX	69.9	67.1	68.0

Evaluating the data-ablation models on a subset. The five data-ablation models are evaluated on a fixed 475-task subset of the 974-task evaluation set: all 275 target-slice tasks, plus 200 of the remaining 699 drawn by strata of application and operation. The sampling looks only at task properties—no outcome of any model enters it—and the list was frozen before the models were trained. The target-slice tasks are taken whole. The main-table models are always evaluated on the full 974 tasks, and any comparison between an ablation model and the main model re-scores the main model on the same 475 tasks, task by task.

F REFERENCE EXECUTORS AND TRAJECTORY RECORDING

Why a program suffices here. Each task family is one operation in one application, and its features change the labels a task is built from while the steps that solve it stay the same, so a family has a fixed solution procedure parameterised by the task's labels and a single program can solve all of its instances. The environment utilities of AndroidWorld that the generator builds on provide state seeding and success checks but no interface-level solutions, so we implement this procedure ourselves, once per family and with a coding agent, as a *reference executor*. The executors were first written to validate generated tasks and later reused for recording.

Execution. An executor parses the instruction, reads the accessibility tree of the live screen, and acts in the agent's own action space, such as tapping, typing and scrolling. It calls no model. It reads the exact text of every element and, on a list, scrolls to the end and reads every candidate before selecting the target by its full description. One executor serves a family's feature-on and feature-off tasks alike. A feature that introduces a new page or source of information needs new executor code. Executors are written once per family.

Validation and recording. Every generated task passes a three-way check: the untouched application fails the success check, the executor passes it, and a near-miss on a distractor fails it. When recording, the executor stores the screenshot before each action and the action itself. A recorded trajectory is kept only if every step rests on evidence shown on that step's screen, so that the demonstration can be followed from the screenshot alone, and the task's success check accepts the final state. The text accompanying each action is a templated one-sentence description for GUI-Owl-1.5-8B. For MAI-UI-8B and UI-Venus-1.5-8B, whose output formats include a reasoning field, each base agent writes the reasoning for every recorded step itself, prompted with the executor's action and asked to explain, from the screen and the history, why it is the right next step. The action in the training target remains the executor's.

Engineering cost and scope. SLICE2FIX replaces a stronger teacher with engineering per task family: each family needs a generator that writes tasks from feature labels and a reference executor that solves them. This cost grows with the number of families and not with the number of tasks, since once a family is written, any number of repair tasks can be generated and solved without a stronger model. Each executor is written once with a coding agent, so the effort lies less in writing it than in validating it, which the three-way check of every generated task performs automatically. A feature that introduces a new page or source of information needs new executor code, and applications without programmatic success checks are out of reach. The features are designed in advance, so failures that no feature describes cannot be mined: the 12 slices hold 59.6% of the failures in the discovery pool, and the rest fall outside every slice. The same features nevertheless mark failures on AndroidWorld tasks that we did not write, which suggests that they capture properties that recur beyond our generator.