MOVI-AGENT: LEARNING QUERYABLE SOFTWARE MODELS FROM INSTRUCTIONAL VIDEOS FOR PROFESSIONAL GUI AUTOMATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Vision-language model (VLM) based agents can operate everyday applications through their graphical user interfaces (GUIs), yet they break down on professional software such as Darktable and KiCad, where dense panels, strict operational ordering, sparse on-screen guidance, and heavy domain terminology demand knowledge that pre-training does not supply. We present MOVI-Agent, a training-free agent that learns how a program works before operating it. From instructional videos it compiles a persistent, queryable software model: a function tree of what the software can do, a state graph of how its interface states connect, and an action database of how each demonstrated step is performed. Execution then becomes structured navigation over this model, and a semantic state backtracking mechanism rewinds to the last verified state whenever it drifts off track. We further construct ProbeGUI, a diagnostic benchmark for professional desktop software: 86 tasks in ten applications with step- and subtask-level error attribution, multi-path matching that credits alternative valid routes, and quantified recovery from injected errors. On ProbeGUI, no baseline exceeds 14.0% task success; MOVI-Agent more than triples the task success of its GPT-5.4 backend (5.8% to 19.8%), surpasses the strongest baseline on all four metrics with significant step- and subtask-level margins, and recovers from 62.5% of injected errors. A matched control that receives the same videos as flat text shows where structure matters: it leaves step accuracy within 2.3 points but raises task success from 15.1% to 19.8% and error recovery from 54.2% to 62.5%.

1 INTRODUCTION

Graphical user interface (GUI) automation, in which a program operates applications the way a human user does, underpins a growing share of digital productivity. Where record-and-replay scripts and rule-based robotic process automation break whenever the interface shifts (Leshed et al., 2008; van der Aalst et al., 2018), vision-language model (VLM) based GUI agents read the rendered screen and act through a perceive–predict–act loop (Hong et al., 2024; Qin et al., 2025; Agashe et al., 2025), already handling everyday applications zero-shot.

That competence collapses on professional software such as Darktable (photographic post-processing) and KiCad (electronic design). Four properties separate this software from everyday applications: *high information density*, *strong structural dependency* through deep nesting and strict operational ordering, *low procedural guidance*, and *high semantic load* from domain terminology. Each withholds knowledge the agent cannot recover from the screen. Dense interfaces exceed the effective resolution of vision encoders (Li et al., 2025): not knowing *which element realizes which function*, the agent mislocalizes and collapses distinct interface states (*state aliasing*). Strict ordering demands a map of *how interface states connect*, a topology no single frame reveals. Jargon-laden steps fall outside pre-trained knowledge, and sparse on-screen guidance offers no substitute: lacking a catalog of *what the software can do*, the agent invents functions that do not exist.

Each missing piece is knowledge about the program itself, and none of it fits in a screenshot. We therefore seek a persistent representation of software that captures this knowledge across interactions. We call this representation a *software model*: a catalog of the program's functions, a map of

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.

its interface states, and a record of how experts operate it, one part for each failure above. Humans build such representations by watching demonstrations (Darejeh et al., 2022), and demonstrations exist at scale: instructional videos accompany nearly every professional application and carry the full interaction sequence with every intermediate state. Prior agents retain this knowledge as captions, guideline text, retrieved exemplars (Gao et al., 2024; Xu et al., 2025; Zheng et al., 2024; Sarch et al., 2024), or implicitly in learned weights (Song et al., 2026a; Lu et al., 2025), but these forms provide limited explicit structure for querying which functions exist, how interface states connect, and how demonstrated operations realize them. This raises a central question: *Can an agent acquire a structured, operational representation of software from instructional demonstrations, and use this representation to guide subsequent execution?* We address this question by **learning an explicit, persistent, and queryable software model from instructional videos**, augmenting transient visual reasoning with **structured navigation over a learned functional state space**.

We instantiate this idea as **MOVI-Agent** (**M**odeling **O**perations from **V**ideo **I**nstructions), a training-free GUI agent, which must meet three challenges. **(C1) Structuring video knowledge:** discrete operations must be extracted from pixel streams and organized into a persistent, queryable model. **(C2) Aligning demonstration with runtime:** demonstrated coordinates rarely transfer verbatim across window sizes, themes, and versions, so function-level semantics must be re-anchored on the live screen. **(C3) Executing under knowledge bias:** when video priors are incomplete or outdated, deviations must be detected and corrected on the fly. MOVI-Agent answers them with the queryable software model (C1), function-to-action alignment plus runtime re-grounding (C2), and semantic state backtracking (C3).

Evaluating this question requires more than binary task success. Most existing benchmarks neither attributes failures nor reveals whether an agent can recover from deviations (Nayak et al., 2025; Li et al., 2025). We therefore build **ProbeGUI**, a diagnostic benchmark for professional desktop software with step- and subtask-level error attribution, deterministic multi-path matching, quantified recovery from injected errors, and a demonstration video of a related but different task for every task.

In summary, our contributions are:

1. **Learning queryable software models from instructional videos.** We formulate software learning as constructing a persistent, queryable representation that separates functional organization, semantic interface transitions, and reusable action precedents. MOVI-Agent compiles instructional demonstrations into this representation and queries it to guide execution, without fine-tuning or interactive exploration.
2. **A diagnostic benchmark that separates agents.** ProbeGUI brings step-level error attribution, multi-path matching, and quantified error recovery to ten professional desktop software; its hierarchical metrics separate methods that binary task success cannot: no baseline exceeds 14.0% task success, yet step accuracies span 24.4–70.1% among baselines that satisfy the output contract.
3. **Substantial gains and diagnostic analysis.** On ProbeGUI, MOVI-Agent more than triples the task success of its GPT-5.4 backend ($5.8\% \rightarrow 19.8\%$), leads the strongest agent baseline on every metric, and lifts the backend's error recovery from 45.8% to 62.5%. Ablations and runtime diagnostics characterize how video guidance, structured knowledge, grounding, and verification affect execution at different granularities. Code, benchmark, and harness will be released.

2 Related Work

GUI agents. Recent GUI agents broadly follow three paradigms. *General VLMs* operate interfaces zero-shot (OpenAI, 2026; Anthropic, 2026), while *fine-tuned specialists* improve GUI grounding through domain-specific training (Hong et al., 2024; Qin et al., 2025; Yang et al., 2026c; Zhang et al., 2026). *Agent frameworks* further augment frontier VLMs with planning, memory, and reflection (Gonzalez-Pumariega et al., 2025; Yang et al., 2026a; Song et al., 2026b). These methods primarily reason from runtime observations, providing limited application-specific structure for navigating deeply nested professional software.

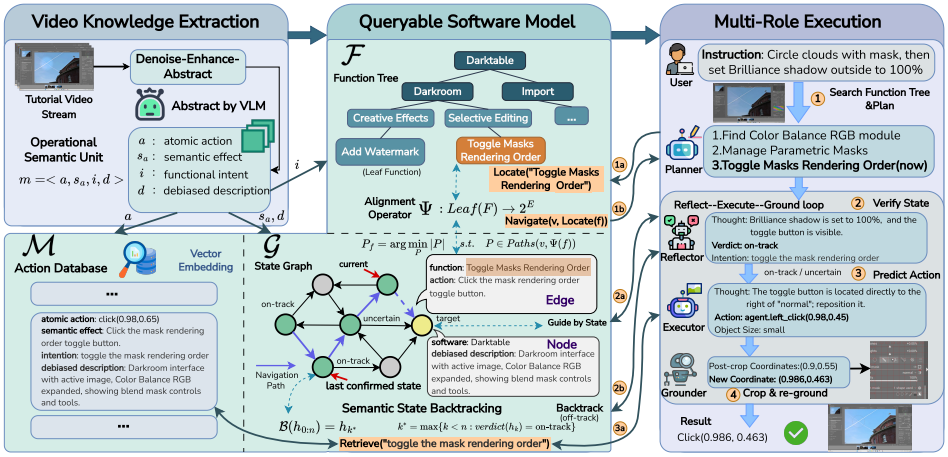


Figure 1: Overview of MOVI-Agent on the running Darktable example. Left: the VideoParser turns instructional videos into operational semantic units $m = \langle a, s_a, i, d \rangle$. Center: the units populate the software model; the alignment operator tags the state-graph edge performing each leaf function. Right: the software model guides multi-role execution through function selection, state-graph navigation, precedent retrieval, and re-grounding; semantic state backtracking uses the last verified state as a reference when off track.

Learning from Experience for GUI Agents. GUI agents increasingly accumulate or retrieve experience beyond the current observation. One line learns by interacting with the environment: AppAgent accumulates application knowledge from exploration and user demonstrations (Zhang et al., 2025), while GUI-Xplore, Agent-SAMA constructs interface transition graphs through exploration (Sun et al., 2025; Guo et al., 2026). Another line leverages demonstrations and instructional media: ICAL distills trajectories into reusable memories (Sarch et al., 2024), AssistGUI extracts guidance from instructional-video captions (Gao et al., 2024), and GUIDE retrieves tutorial videos for planning and grounding (Xie et al., 2026). These methods establish that exploration, demonstrations, and instructional videos can provide useful experience beyond the immediate screen. MOVI-Agent studies a different learning target: instructional videos are compiled into persistent functional, state-transition, and action knowledge that can be systematically queried across subsequent tasks, without interactive exploration.

Benchmarks for GUI agents. Online benchmarks, including professional ones (Xie et al., 2024; Bonatti et al., 2025; Sun et al., 2026; Hu et al., 2026), run tasks live yet collapse a trajectory into a binary verdict that obscures where the agent fails. Offline grounding datasets for professional interfaces (Nayak et al., 2025; Li et al., 2025) score static element localization rather than sequential execution. Process-aware efforts come closest: VideoGUI (Lin et al., 2024) is offline, professional, and hierarchical but grades each run against a single gold trajectory; ProBench (Yang et al., 2026b) examines mobile apps; both rely on LLM judges, which are costly and susceptible to hallucination. ProbeGUI provides deterministic multi-path matching that credits alternative valid routes and injected error states that quantify recovery, across ten professional desktop applications.

3 METHOD

MOVI-Agent learns a persistent software model from instructional videos and queries it to guide subsequent execution. Figure 1 follows this process from left to right: the VideoParser extracts operational semantic units; model construction organizes them into a function tree $\mathcal{F}$, a state graph $\mathcal{G}$, and an action database $\mathcal{M}$; execution queries these structures to plan, navigate, and act while checking progress against the modeled states. Learning here constructs explicit software knowledge without updating the VLM's parameters.

3.1 VIDEO KNOWLEDGE EXTRACTION

Addressing C1, the VideoParser (Figure 1, left) converts an unstructured video into a sequence of *atomic actions* annotated with their semantics. An atomic action is one GUI primitive from a fixed action space (click, text input, hotkey, drag, or scroll). The pipeline has three stages: **Denoise** retains keyframes that capture interface changes; **Enhance** adds aligned narration and spatial cues to recover the demonstrator's intent and action target; **Abstract** has a VLM formalize each enriched keyframe into a structured unit, then filters redundant units and merges mechanically fragmented ones, so a string typed as dozens of keystrokes becomes one text input. Appendix A details the extraction protocol.

Each retained interaction is an *operational semantic unit* $m = \langle a, s_a, i, d \rangle$. In Figure 1, a is the atomic action `Click(0.98, 0.65)`; s_a describes its action-centric semantic effect, "click the mask rendering order toggle button"; i is the functional intent, *Toggle Masks Rendering Order*; and d describes the resulting interface state. To support reuse, d omits personalized inputs and transient visual details while retaining available functions and interactive elements. These fields connect demonstrated interactions to the model's three layers: intents i associate operations with functions, descriptions d and effects s_a define semantic states and transitions, and actions a with their semantics provide retrievable operational precedents.

3.2 QUERYABLE SOFTWARE MODEL

Model construction organizes the extracted units into three persistent, linked structures (Figure 1, center). The function tree $\mathcal{F}$ represents *what the software can do*, the state graph $\mathcal{G}$ represents *how its interface states connect*, and the action database $\mathcal{M}$ records *how demonstrated operations are performed*. Each structure supports a different execution need: function selection, path search, or retrieval of action precedents.

The **function tree** $\mathcal{F}$ organizes directly operable functions as leaves under functional categories. A one-time lookup of official documentation initializes the hierarchy; video-grounded extraction then associates each observed intent i with its leaf. Thus, documentation supplies the organizational scaffold, while demonstrations identify the functions available for planning. In Figure 1, *Selective Editing* contains the leaf *Toggle Masks Rendering Order*. The Planner selects only video-supported leaves, using the closest demonstrated function when no exact match exists. This constrains plans to the acquired vocabulary but does not guarantee that the selected function satisfies the task.

The **state graph** $\mathcal{G}$ abstracts demonstrated screens into semantic interface states. Its nodes are debiased descriptions d, and its directed edges record the semantic effects s_a of observed transitions. Screens with the same debiased structure merge into one node, allowing transitions from separate videos to connect through shared interface states. The resulting graph retains demonstrated navigation relationships without storing each video's personalized content as a distinct state.

The **action database** $\mathcal{M}$ retains atomic actions together with their semantics, embedded into a shared vector space (OpenAI, 2024) and indexed for similarity search. Entries whose semantic descriptions exceed a cosine-similarity threshold merge into one operation with an occurrence count, used as a reliability prior. Unlike the state graph, which supports navigation, this store preserves operational precedents for predicting individual actions. Stored coordinates identify the demonstrator's target; execution re-predicts coordinates on the live screen rather than replaying them.

An **alignment operator** bridges the layers by recording, for each leaf function f, which state transitions accomplish it: $\Psi(f) \subseteq E$, where E is the edge set of $\mathcal{G}$ and $\Psi(f)$ is the set of edges tagged with f. Tags are assigned as the graph is built: an LLM judges from the unit's intent i and observed effect which leaf functions each added transition completes. Executing f then reduces to a constrained shortest-path search on $\mathcal{G}$: $P_f = \arg\min_P |P|$ subject to $P \in \mathrm{Paths}(v, \Psi(f))$, where $\mathrm{Paths}(v, \Psi(f))$ collects the paths from the agent's current node v to the source of any edge in $\Psi(f)$, and $|P|$ counts edges. This alignment supplies the semantic half of C2; runtime re-grounding completes the other half.

Query primitives. Three primitives form the model's runtime interface, and every access to $(\mathcal{F}, \mathcal{G}, \mathcal{M})$ during execution reduces to one of them. $\textsc{Locate}(f) = \Psi(f)$ returns the transitions that accomplish leaf function f; $\textsc{Navigate}(v, E')$ returns a shortest path from the current node v

Algorithm 1 MOVI-Agent execution for one task

Input: instruction q; model $(\mathcal{F}, \mathcal{G}, \mathcal{M})$; screen S; step budget T
$\pi \leftarrow$ PLANNER$(q, \mathcal{F})$ {leaf functions of $\mathcal{F}$; runs once}
anchor S to a node v of $\mathcal{G}$; $h_0 \leftarrow (S, v,$ on-track$)$; $j, t \leftarrow 1$; $P \leftarrow$ NAVIGATE$(v,$ LOCATE$(\pi_j))$
repeat
  $(r, v', u) \leftarrow$ REFLECTOR(S, P, v) {verdict r, best-matching node v' on P, intent query u}
  if r is *on-track* **then** $v \leftarrow v'$ {*uncertain* leaves v unchanged}
  record $h_t{=}(S, v, r)$; **if** r is *off-track* **then** $v \leftarrow \mathrm{node}(\mathcal{B}(h_{0:t}))$ {semantic rewind}
  if v is the completion state of π_j **then** $j \leftarrow j{+}1$; **if** $j \leq |\pi|$ **then** $P \leftarrow$ NAVIGATE$(v,$ LOCATE$(\pi_j))$
  $H \leftarrow$ RETRIEVE(u) {operational precedent from $\mathcal{M}$}
  $(a, z) \leftarrow$ EXECUTOR(S, P, H) {z: target-size estimate}
  $a^* \leftarrow$ GROUNDER(a, z, S); execute a^*; observe the new S; $t \leftarrow t{+}1$
until $j > |\pi|$ **or** $t > T$

to the source of some edge in $E' \subseteq E$, or $\varnothing$ when none is reachable; RETRIEVE(u) returns stored actions whose semantics best match an intent u. The first two compose into structured navigation, $P_f =$ NAVIGATE$(v,$ LOCATE$(f))$. RETRIEVE runs at every step and is also the fallback: when NAVIGATE returns $\varnothing$ because the graph may not be fully connected, execution continues on retrieved precedent, so a gap in the graph slows the agent without stopping it. Anchoring the live screen is retrieval of the same kind: the screen is described with the debiasing prompt that yields d, embedded, and matched to the nearest node of $\mathcal{G}$; a screen matching no node closely enough is treated as unmodeled, and execution proceeds on retrieved precedent until a later screen anchors (Appendix A.4).

3.3 MULTI-ROLE EXECUTION

MOVI-Agent divides execution among four roles according to how they use the software model. The **Planner** selects functions from $\mathcal{F}$, which LOCATE and NAVIGATE translate into reference paths through $\mathcal{G}$. The **Reflector** checks the live screen against the expected graph state and produces an intent query for RETRIEVE. The **Executor** uses the retrieved precedents from $\mathcal{M}$, the reference path, and the live screen to predict an action. The **Grounder** refines its target coordinate using local visual evidence rather than stored coordinates. Together, these roles connect function selection, state verification, precedent-based action prediction, and visual grounding through a shared execution loop (Figure 1, right; Algorithm 1). Runtime re-grounding addresses demonstration–runtime alignment (C2), while semantic state backtracking uses verification to guide recovery (C3).

Model-guided execution. The Planner first decomposes the instruction into leaves of $\mathcal{F}$ (three in Figure 1, ending in *Toggle Masks Rendering Order*). After the opening screen is anchored to $\mathcal{G}$, LOCATE identifies the transitions associated with the first function, and NAVIGATE supplies its reference path. The graph then serves as a verification reference: at each step, the Reflector compares the current screen with the expected semantic state, returns *on-track*, *off-track*, or *uncertain*, and emits the intent query u for RETRIEVE. This frames progress checking as verification against a known state description rather than open-ended action prediction. A function completes when the Reflector confirms a screen matching the target state of one of its tagged transitions; the next reference path is then queried. To act, RETRIEVE supplies precedents from $\mathcal{M}$, and the Executor predicts an atomic action and target-size estimate using these precedents, the reference path, the live screen, and recent context. The Grounder crops around the predicted coordinate with a window scaled to the size estimate, magnifies it, and re-predicts before mapping back to screen coordinates; Figure 1 illustrates the refinement from $(0.98, 0.45)$ to $(0.986, 0.463)$. This *runtime re-grounding* adapts crop-and-zoom grounding (Li et al., 2025): the software model guides *what* operation to perform, while the live screen determines *where* to perform it. The resulting screen becomes the next input to verification.

Model-guided recovery. When verification returns *off-track*, $\mathcal{G}$ serves a second purpose: it supplies a semantic reference for recovery. *Semantic state backtracking* selects the most recent verified record from the execution history. Each record h_k stores the screen, its anchored graph node $\mathrm{node}(h_k)$, and the verdict at that visit. The recovery reference is $\mathcal{B}(h_{0:n}) = h_{k^*}$, where $k^* = \max\{k < n : \mathrm{verdict}(h_k) = \text{on-track}\}$. Execution resumes with $\mathrm{node}(h_{k^*})$ as the reference

and the Reflector's account of the deviation added to context. Backtracking resets the agent's semantic reference, not the application: no snapshot is restored. Instead, the Executor and Grounder predict corrective actions from the live screen toward the selected reference state, and the Reflector checks whether the reference state has been reached. Unlike recovery through backward dependencies in a recorded transition graph (Wen et al., 2024), this mechanism uses a verified semantic state as the recovery target rather than replaying a recorded route. An *uncertain* verdict deliberately does not trigger backtracking, avoiding premature recovery from an ambiguous observation; later screens may resolve the ambiguity or reveal a deviation. Backtracking addresses execution deviations, not errors in the Planner's initial decomposition.

4 PROBEGUI BENCHMARK

ProbeGUI is a *diagnostic* benchmark for GUI agents on professional desktop software: 86 tasks (35 easy, 32 medium, 19 hard) across ten applications, comprising 264 subtasks and 1,574 atomic actions; 48 tasks carry an injected error episode (115 error-recovery states in total), and 29% of annotated states admit more than one valid action. Appendix B illustrates the design (Figure 5) and gives the specification; Appendix C gives statistics and the construction protocol.

Professional software scope. ProbeGUI's ten applications (e.g., Darktable and KiCad) span research, development, engineering, data analysis, and media; tasks are drawn from official documentation and community forums and categorized as easy (< 10 atomic actions), medium (10–19), or hard (≥ 20); all videos are concise instructional demonstrations recorded in-house (Appendix C.3). Each task is paired with a demonstration video of a *similar* task that may differ in parameters, invoked function, workflow structure, or a combination thereof; shared navigation can provide procedural guidance without giving away an identical target task. Across all 86 pairs, 98.8% differ in parameters, 29.1% invoke a different functional module, and 8.1% follow a different abstract workflow; no task is identical to its paired demonstration.

Process-aware design. Two annotations make the benchmark process-aware. For **multi-path workflows**, multiple valid action at each step is annotated together with its resulting application state; in the KiCad task "create a resistor and set its value", double-clicking the resistor and right-clicking it then selecting *Properties* are both credited. For **error recovery**, predefined execution errors are injected and the agent must recognize the mistake and rejoin an annotated recovery path: the injected error in the same task clicks the resistor label instead of the resistor, leaving the value uneditable.

Hierarchical diagnostic metrics. ProbeGUI scores four granularities, since a single binary verdict conflates near-complete executions with total failures. **Step accuracy** scores each atomic operation and attributes every failed step to a wrong action type, position, or parameters, or to an invalid output; **subtask success rate** measures whether each annotated intermediate goal is reached; **task success rate** requires every step along the followed branch to match an annotated valid action. **Error-recovery rate** is the fraction of the 48 error-injected tasks in which the agent rejoins an annotated valid path from the injected error state; recovery is scored in a separate pass that places the agent in the error state with the history that produced it, so injected errors never contaminate the ordinary trajectory, and a task counts as recovered only if every step of its recovery episode matches an annotated recovery action. Outside recovery episodes, scoring follows a *continue-on-miss* rule: a step whose prediction matches no annotated valid action counts as failed, and evaluation advances along the state's first annotated action, so one early mistake degrades the score without zeroing out the rest of the trajectory. ProbeGUI thereby acts as an annotation-driven simulator: a matched action yields its annotated resulting state as the next observation, and every step stays conditioned on a coherent trajectory. Continue-on-miss suppresses the downstream effects of incorrect actions, while valid but unannotated routes may be scored as errors. Absolute scores therefore characterize execution on annotated trajectories rather than unrestricted live success. Every method uses the same continuation protocol, and matching is deterministic (Appendix B.3).

Table 1: Results on ProbeGUI (%; higher is better). ER is reported only overall because recovery cases are assessed independently of difficulty. Best in bold, second-best underlined.

Category	Method	Step Acc. ↑				Subtask SR ↑				Task SR ↑				ER ↑
		Easy	Med.	Hard	Overall	Easy	Med.	Hard	Overall	Easy	Med.	Hard	Overall	Overall
Open-source VLMs	MiniMax M3	22.8	22.6	27.6	24.4	6.0	2.9	1.3	3.4	0.0	0.0	0.0	0.0	27.1
	Qwen3-VL-235B-A22B	51.1	42.7	42.3	44.8	15.5	9.6	3.9	9.8	0.0	0.0	0.0	0.0	12.5
	Kimi K2.6	68.7	66.4	68.6	67.8	40.5	30.8	21.1	31.1	20.0	0.0	0.0	8.1	47.9
Closed-source VLMs	Claude Sonnet 4.6	13.7	10.1	11.3	11.5	3.6	0.0	0.0	1.1	0.0	0.0	0.0	0.0	10.4
	GPT-5.4	72.4	64.6	66.1	67.2	44.0	26.0	21.1	30.3	11.5	3.1	0.0	5.8	45.8
	GPT-5.4 + Video Desc.	73.9	70.7	66.6	70.1	50.0	33.7	31.6	38.3	20.0	3.1	0.0	9.3	52.1
Specialized GUI Agents	UI-TARS-1.5-7B	6.9	5.7	6.5	6.3	1.2	0.0	0.0	0.4	0.0	0.0	0.0	0.0	6.2
	Agent S3	56.2	54.5	47.7	52.6	27.4	24.0	19.7	23.9	5.7	0.0	0.0	2.3	39.6
	UiPath	68.1	62.5	65.9	65.2	47.6	29.8	30.3	35.6	14.3	6.2	**5.3**	9.3	52.1
	OS-Symphony	73.2	65.7	67.4	68.3	48.8	33.7	27.6	36.7	25.7	9.4	0.0	14.0	50.0
Ours	**MOVI-Agent**	**82.1**	**79.0**	**79.6**	**80.0**	**60.7**	**51.9**	**44.7**	**52.7**	**31.4**	**15.6**	**5.3**	**19.8**	**62.5**
	↑ *Gain over best baseline*	*+8.2*	*+8.3*	*+11.0*	***+9.9***	*+10.7*	*+18.2*	*+13.1*	***+14.4***	*+5.7*	*+6.2*	*+0.0*	***+5.8***	***+10.4***

5 EXPERIMENTS

5.1 EXPERIMENTAL SETUP

Evaluation. We report the four metrics above as **Step Acc.**, **Subtask SR**, **Task SR**, and **ER**. Each method runs once per task (pass@1), as is common given API cost, and we report a paired task-level bootstrap with Benjamini–Hochberg correction across all 60 method-by-metric comparisons (Appendix D.5). To check the deterministic scorer against live application behavior, we replayed the logged actions of 60 trajectories in the corresponding applications; manual replay judgments agreed with the offline verdicts for all 60 selected trajectories (Appendix D.2).

Baselines. We compare MOVI-Agent with three groups of baselines: (i) **Open-source VLMs**: MiniMax M3 (MiniMax, 2026), Qwen3-VL-235B-A22B (Bai et al., 2025), and Kimi K2.6 (Moonshot AI, 2026); (ii) **Closed-source VLMs**: Claude Sonnet 4.6 (Anthropic, 2026) and GPT-5.4 (OpenAI, 2026); and (iii) **Specialized GUI agents**: UI-TARS-1.5-7B (Qin et al., 2025), Agent S3 (Gonzalez-Pumariega et al., 2025), the commercial UiPath agent (UiPath, 2026), and OS-Symphony (Yang et al., 2026a). To control for access to instructional-video knowledge, we add **GPT-5.4 + Video Desc.**, which augments plain GPT-5.4 with LLM-generated textual descriptions retrieved from the same video corpus. This matched control separates the benefit of video-derived knowledge from the structured representation and execution mechanism used by MOVI-Agent.

Evaluation setup and implementation. All methods receive the same base task inputs under the OSWorld observation protocol (Xie et al., 2024): the instruction, state-specific constraints, action history, and the current and two preceding screenshots. Framework baselines retain their native prompts, reasoning logic, and grounding modules, with lightweight adapters mapping outputs to the unified action space; standalone VLMs share a common action-space and output-format prompt. MOVI-Agent and other framework baselines use GPT-5.4 as the underlying VLM. For each application, MOVI-Agent builds one software model offline from the instructional-video corpus and reuses it across tasks, while GPT-5.4 + Video Desc. retrieves flat textual descriptions derived from the same corpus. At runtime, MOVI-Agent averages 38.7 model calls and 225.7K tokens per task, compared with 50.7 calls and 355.8K tokens for OS-Symphony; building each application model takes about 74 minutes as a one-time offline cost under concurrency (Appendix D.1).

5.2 MAIN RESULTS

MOVI-Agent leads every metric. Table 1 shows it surpassing the best baseline by 9.9, 14.4, 5.8, and 10.4 points on the four metrics. Under the paired bootstrap, the step- and subtask-level leads are significant against every baseline, including OS-Symphony and the matched GPT-5.4 + Video Desc. control ($q < 0.001$). At the task level, the gain over the GPT-5.4 backend is significant ($q < 0.05$), while the 5.8-point gap over OS-Symphony is not resolvable over 86 binary outcomes ($p = 0.30$); error recovery behaves likewise, significant over the backend ($q < 0.05$) but not over the strongest agents on 48 cases ($p = 0.16$ against OS-Symphony).

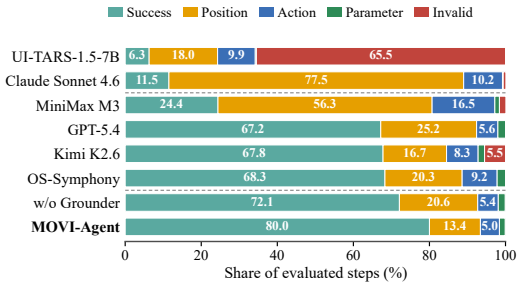


Figure 2: Error-type distribution across evaluated steps (%). Dashed lines separate output-contract outliers (top), position-error-dominant baselines (middle), and the Grounder ablation (bottom); all methods in Appendix D.6.

Table 2: Ablation results on ProbeGUI (%). The three video variants probe C1, *w/o Grounder* probes C2, and *w/o Reflector* probes C3.

Variant	Step Acc.	Subtask SR	Task SR	ER
MOVI-Agent	**80.0**	**52.7**	**19.8**	**62.5**
w/o Video Guidance	74.0	45.5	10.5	47.9
w/ Flat Description	77.7	50.4	15.1	54.2
w/o Paired Video	76.1	44.7	14.0	52.1
w/o Grounder	72.1	40.2	16.3	64.6
w/o Reflector	78.2	50.0	20.9	52.1

Failures remain closer to completion. Nine of ten baselines complete zero hard tasks, and none exceeds 14.0% overall task success. Among MOVI-Agent's 69 failed tasks, 39 (56.5%) contain only one (22) or two (17) failed atomic steps, compared with 20 of 74 (27.0%) for OS-Symphony.

Video priors and structured execution are complementary. GPT-5.4 + Video Desc. improves over plain GPT-5.4 on all four metrics (task success 5.8% $\rightarrow$ 9.3%), and MOVI-Agent more than doubles that variant's task success (9.3% $\rightarrow$ 19.8%); the ablation below separates the execution framework's share of this margin from the structure's.

5.3 Error Analysis

Figure 2 breaks every evaluated step into success and four error causes. Positioning errors dominate baseline failures (56.3% of all steps for MiniMax M3), typically a plausible operation aimed at the wrong control in a dense panel, and are where MOVI-Agent gains most: its 13.4% is roughly half of its GPT-5.4 backend's 25.2%, and its 5.0% action-selection errors are also the lowest. Two outliers are strongly affected by output-contract mismatch: Claude Sonnet 4.6 mixes coordinate conventions within single replies (77.5% positioning errors), and a post hoc reprojection that resolves each reply's convention lifts it to 57.8% step accuracy and 8.1% task success (Appendix D.7); UI-TARS-1.5-7B fails to produce a scorable unified action on 66.0% of generated steps, including 147 outputs in its native action syntax. These raw rows are therefore strongly confounded by interface adaptation; the main comparisons use OS-Symphony and GPT-5.4 + Video Desc., which conform to the unified output contract.

5.4 Ablation Study and Model Diagnostics

Table 2 compares MOVI-Agent with five variants. Three variants probe how video knowledge is represented and reused: *w/o Video Guidance* removes the software model while retaining the four-role execution loop; *w/ Flat Description* replaces the structured model with the same video-derived text used by GPT-5.4 + Video Desc.; and *w/o Paired Video* masks model components attributed to the evaluated task's paired demonstration. The remaining variants probe execution: *w/o Grounder* directly executes the Executor's coarse prediction, while *w/o Reflector* removes state verification and semantic backtracking. Appendix A.5 gives the complete variant definitions and Appendix D.5 reports the statistical analysis.

The software model supplies structured, reusable video knowledge (C1). Removing video guidance eliminates the software model while retaining the four-role execution loop, and causes the largest C1 degradation: Step Acc. falls from 80.0% to 74.0%, Subtask SR from 52.7% to 45.5%, Task SR from 19.8% to 10.5%, and ER from 62.5% to 47.9%. The remaining four-role framework still outperforms plain GPT-5.4, showing that execution structure contributes independently, but most of the task-level gain comes from adding software-specific video knowledge. We then probe two different aspects of that knowledge.

w/ Flat Description keeps the paired video's information but replaces the structured software model with its LLM-generated description. It trails the full model on every metric, by 2.3 points in Step Acc. and Subtask SR, 4.7 points in Task SR ($15.1\% \rightarrow 19.8\%$), and 8.3 points in ER ($54.2\% \rightarrow 62.5\%$). This comparison probes whether task-matched flat video guidance can substitute for the persistent software model, which organizes reusable functional, state, and action knowledge across demonstrations.

w/o Paired Video probes the complementary axis: it preserves the structured model but masks knowledge attributed to the evaluated task's paired demonstration. MOVI-Agent still reaches 76.1% Step Acc., 44.7% Subtask SR, and 14.0% Task SR, showing that the learned model remains useful through knowledge accumulated from other demonstrations, while the paired video provides additional task-specific information. Together, the three ablations show that C1 gains come from the presence of video-derived software knowledge itself, and are further shaped by both how that knowledge is structured and which demonstrations contribute to it.

The Grounder sharpens runtime alignment (C2). Removing it lowers Step Acc. by 7.9 points and Subtask SR by 12.5 points, the largest drops in routine execution among the ablations. The error breakdown shows where the loss comes from: positioning errors rise from 13.4% to 20.6%, while action-selection errors remain nearly unchanged ($5.0\% \rightarrow 5.4\%$; Figure 2). The Executor therefore often identifies the intended operation but only approximately localizes its target; the Grounder's local crop-and-zoom pass converts this coarse prediction into an executable coordinate on the live screen. Its effect on ER is much smaller (-2.1 points, one recovery episode), consistent with re-grounding primarily addressing demonstration–runtime alignment rather than recovery itself.

The Reflector supplies the recovery trigger (C3). Removing it lowers ER by 10.4 points (five cases): without comparing the screen against the expected graph state, the agent loses the trigger for semantic backtracking; Task SR instead ticks up by one task ($+1.2$), as otherwise on-track runs are spared occasional false-positive rewinds. Because this ablation removes verification and backtracking together, we further inspect the full system's verdicts; false alarms are rare, with *off-track* issued on only 0.9% of 973 normal steps. Across the 48 injected episodes, the Reflector outputs *off-track* in 7 cases (14.6%), triggering backtracking, of which 85.7% recover; *uncertain* in 31 (64.6%), where no backtracking is triggered and 64.5% recover through live-screen execution; and *on-track* in 10 (20.8%), where the deviation is not explicitly detected and 50.0% recover. Recovery is highest when the Reflector confidently identifies a deviation; the split is observational and complements the ablation by showing how verification drives recovery in practice (Appendix D.3).

The learned model captures usable operational knowledge. On the intrinsic VideoParser audit, extracted operations achieve a type-agnostic temporal F1 of 63.5%, showing that the model is built from imperfect rather than oracle demonstrations. At the functional level, the demonstration corpus covers 84.8% (224/264) of evaluation subtasks; runtime Function Plans recover 89.7% (201/224) of those supported by the demonstrations, or 76.1% of all evaluation subtasks. For the state graph, NAVIGATE returns a nonempty path for 99.2% of function-switch queries encountered during execution. Appendix A.6 gives the full diagnostics.

6 CONCLUSION AND LIMITATIONS

This paper started from a diagnosis: VLM-based GUI agents struggle on professional software because they lack a persistent, structured model of the program they operate. MOVI-Agent tests this diagnosis constructively by learning an explicit, queryable software model from instructional videos and using it to guide planning, execution, and recovery without fine-tuning. On ProbeGUI, it lifts its GPT-5.4 backend's task success from 5.8% to 19.8%; the ablations and model diagnostics further show how video-derived knowledge is acquired, structured, reused across tasks, and queried during execution. The remedy remains partial (Appendix G): long-horizon tasks remain difficult, VideoParser extraction is imperfect, and models built on fixed application versions may become stale as interfaces evolve. Scaling software-model learning to noisier demonstrations and maintaining learned structure across software updates are natural next steps.

Reproducibility Statement

The supplementary material contains the anonymized code of MOVI-Agent, the ProbeGUI benchmark (tasks, videos, and annotations), and the evaluation harness. Appendix A details construction and execution, Appendix B the action space, annotation schema, and matching rules, Appendix D.1 hyperparameters, model snapshots with dates, decoding parameters, and baseline adapters, Appendix F all prompts, and Appendix D.5 the statistical procedure.

Ethics Statement

All ProbeGUI videos were recorded in-house on locally installed open-source or freely available applications, contain no personally identifiable information or sensitive personal data, and will be released under CC BY 4.0; no human-subject study was conducted. As with any GUI agent, deployment of MOVI-Agent should be confined to accounts and files the user intends it to operate on.

AI Use Statement

We utilized generative AI tools solely to revise and polish portions of the manuscript; every sentence, data point, and figure has been verified against experimental records and reviewed by the authors. The vision-language models (VLMs) discussed in Section 5 serve as the objects of study rather than assistive tools for the research workflow; we did not employ generative AI to conceive research ideas, design experiments, implement core methods, analyze data, or generate results. The authors take full responsibility for the final content of this manuscript.

References

Saaket Agashe, Kyle Wong, Vincent Tu, Jiachen Yang, Ang Li, and Xin Eric Wang. Agent S2: A compositional generalist-specialist framework for computer use agents. In *Second Conference on Language Modeling*, 2025.

Anthropic. Claude sonnet 4.6 system card. System card, Anthropic, February 2026. URL https://www-cdn.anthropic.com/78073f739564e986ff3e28522761a7a0b4484f84.pdf.

Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, Zesen Cheng, Lianghao Deng, Wei Ding, Chang Gao, et al. Qwen3-VL technical report, 2025.

Rogerio Bonatti, Dan Zhao, Francesco Bonacci, Dillon Dupont, Sara Abdali, Yinheng Li, Yadong Lu, Justin Wagle, Kazuhito Koishida, Arthur Bucker, Lawrence Keunho Jang, and Zheng Hui. Windows agent arena: Evaluating multi-modal OS agents at scale. In *Proceedings of the 42nd International Conference on Machine Learning*, pp. 4874–4910. PMLR, 2025.

Ali Darejeh, Sara Mashayekh, and Nadine Marcus. Cognitive-based methods to facilitate learning of software applications via E-learning systems. *Cogent Education*, 9, 2022.

Difei Gao, Lei Ji, Zechen Bai, Mingyu Ouyang, Peiran Li, Dongxing Mao, Qinchen Wu, Weichen Zhang, Peiyi Wang, Xiangwu Guo, Hengxu Wang, Luowei Zhou, and Mike Zheng Shou. AssistGUI: Task-oriented PC graphical user interface automation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 13289–13298, 2024.

Gonzalo Gonzalez-Pumariega, Vincent Tu, Chih-Lun Lee, Jiachen Yang, Ang Li, and Xin Eric Wang. The unreasonable effectiveness of scaling agents for computer use, 2025.

Linqiang Guo, Wei Liu, Yi Wen Heng, Tse-Hsun (Peter) Chen, and Yang Wang. Agent-SAMA: State-aware mobile assistant. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40 (35):29459–29467, 2026. doi: 10.1609/aaai.v40i35.40187.

Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, and Jie Tang. CogAgent: A visual language model for GUI agents. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 14281–14290, 2024.

Haobo Hu, Xiangwu Guo, Zhiheng Chen, Difei Gao, Haotian Liu, Libiao Jin, and Qi Mao. CutVerse: A compositional GUI agents benchmark for media post-production editing, 2026.

Youngmin Im, Byeongung Jo, Jaeyoung Wi, Seungwoo Baek, Tae Hoon Min, Joo Hyung Lee, Sangeun Oh, Insik Shin, and Sunjae Lee. Modular and multi-path-aware offline benchmarking for mobile GUI agents. *arXiv preprint arXiv:2512.12634*, 2025.

Xuanyu Lei, Zonghan Yang, Xinrui Chen, Peng Li, and Yang Liu. Scaffolding coordinates to promote vision-language coordination in large multi-modal models. In *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 2886–2903. Association for Computational Linguistics, 2025.

Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. CoScripter: Automating & sharing how-to knowledge in the enterprise. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 1719–1728. Association for Computing Machinery, 2008.

Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

Kaixin Li, Ziyang Meng, Hongzhan Lin, Ziyang Luo, Yuchen Tian, Jing Ma, Zhiyong Huang, and Tat-Seng Chua. ScreenSpot-Pro: GUI grounding for professional high-resolution computer use. In *Proceedings of the 33rd ACM International Conference on Multimedia*, pp. 8778–8786. ACM, 2025.

Kevin Qinghong Lin, Linjie Li, Difei Gao, Qinchen Wu, Mingyi Yan, Zhengyuan Yang, Lijuan Wang, and Mike Zheng Shou. VideoGUI: A benchmark for GUI automation from instructional videos. In *Advances in Neural Information Processing Systems, Datasets and Benchmarks Track*, 2024.

Dunjie Lu, Yiheng Xu, Junli Wang, Haoyuan Wu, Xinyuan Wang, Zekun Wang, Junlin Yang, Hongjin Su, Jixuan Chen, Junda Chen, Yuchen Mao, Jingren Zhou, Junyang Lin, Binyuan Hui, and Tao Yu. VideoAgentTrek: Computer use pretraining from unlabeled videos. *arXiv preprint arXiv:2510.19488*, 2025.

MiniMax. MiniMax M3: Frontier coding capabilities, 1M context, and native multimodality. `https://www.minimax.io/blog/minimax-m3`, 2026.

Moonshot AI. Kimi K2.6: Next-generation advanced reasoning and long-context language model. `https://www.kimi.com/blog/kimi-k2-6`, 2026.

Jian Mu, Chaoyun Zhang, Chiming Ni, Lu Wang, Bo Qiao, Kartik Mathur, Qianhui Wu, Yuhang Xie, Xiaojun Ma, Mengyu Zhou, Si Qin, Liqun Li, Yu Kang, Minghua Ma, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. GUI-360°: A comprehensive dataset and benchmark for computer-using agents, 2025.

Shravan Nayak, Xiangru Jian, Kevin Qinghong Lin, Juan A. Rodriguez, Montek Kalsi, Nicolas Chapados, M. Tamer Özsu, Aishwarya Agrawal, David Vazquez, Christopher Pal, Perouz Taslakian, Spandana Gella, and Sai Rajeswar. UI-Vision: A desktop-centric GUI benchmark for visual perception and interaction. In *Forty-Second International Conference on Machine Learning*, 2025.

OBS Studio. OBS Studio. `https://obsproject.com/`, 2024.

OpenAI. New embedding models and API updates. `https://openai.com/index/new-embedding-models-and-api-updates/`, 2024.

OpenAI. OpenAI GPT-5 system card, 2026.

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, 2025.

Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision. In *Proceedings of the 40th International Conference on Machine Learning*, pp. 28492–28518. PMLR, 2023.

Gabriel Sarch, Lawrence Jang, Michael J. Tarr, William W. Cohen, Kenneth Marino, and Katerina Fragkiadaki. VLM agents generate their own memories: Distilling experience into embodied programs of thought. In *Advances in Neural Information Processing Systems*, volume 37, pp. 75942–75985, 2024.

Chan Hee Song, Yiwen Song, Palash Goyal, Yu Su, Oriana Riva, Hamid Palangi, and Tomas Pfister. Watch and learn: Learning to use computers from online videos. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2026a.

Linxin Song, Yutong Dai, Viraj Prabhu, Jieyu Zhang, Taiwei Shi, Li Li, Junnan Li, Silvio Savarese, Zeyuan Chen, Jieyu Zhao, Ran Xu, and Caiming Xiong. CoAct-1: Computer-using multi-agent system with coding actions. In *The Fourteenth International Conference on Learning Representations*, 2026b.

Qiushi Sun, Zhoumianze Liu, Chang Ma, Zichen Ding, Fangzhi Xu, Zhangyue Yin, Haiteng Zhao, Zhenyu Wu, Kanzhi Cheng, Zhaoyang Liu, Jianing Wang, Qintong Li, Xiangru Tang, Tianbao Xie, Xiachong Feng, Xiang Li, Ben Kao, Wenhai Wang, Biqing Qi, Lingpeng Kong, and Zhiyong Wu. ScienceBoard: Evaluating multimodal autonomous agents in realistic scientific workflows. In *The Fourteenth International Conference on Learning Representations*, 2026.

Yuchen Sun, Shanhui Zhao, Tao Yu, Hao Wen, Samith Va, Mengwei Xu, Yuanchun Li, and Chongyang Zhang. GUI-Xplore: Empowering generalizable GUI agents with one exploration. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 19477–19486, 2025.

UiPath. UiPath Screen Agent. `https://www.uipath.com/ai/research/screen-agent`, 2026.

Wil M. P. van der Aalst, Martin Bichler, and Armin Heinzl. Robotic process automation. *Business & Information Systems Engineering*, 60(4):269–272, 2018.

Zhou Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: From error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

Hao Wen, Shizuo Tian, Borislav Pavlov, Wenjie Du, Y. J. Li, Chang Ge, Shanhui Zhao, Jiacheng Liu, Yunxin Liu, Ya-Qin Zhang, and Yuanchun Li. AutoDroid-V2: Boosting SLM-based GUI agents via code generation. *arXiv preprint arXiv:2412.18116*, 2024.

Rui Xie, Zhi Gao, Chenrui Shi, Zirui Shang, Lu Chen, and Qing Li. GUIDE:Resolving Domain Bias in GUI Agents through Real-Time Web Video Retrieval and Plug-and-Play Annotation. In Paolo Favaro, Zuzana Kukelova, Atsuto Maki, Anna Rohrbach, Konrad Schindler, and Federico Tombari (eds.), *Computer Vision – ECCV 2026*, pp. 581–598, Cham, 2026. Springer Nature Switzerland. ISBN 978-3-032-37026-6. doi: 10.1007/978-3-032-37026-6_32.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J. 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. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

Ran Xu, Kaixin Ma, Wenhao Yu, Hongming Zhang, Joyce C. Ho, Carl Yang, and Dong Yu. Retrieval-augmented GUI agents with generative guidelines. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 17866–17875. Association for Computational Linguistics, 2025.

Bowen Yang, Kaiming Jin, Zhenyu Wu, Zhaoyang Liu, Qiushi Sun, Zehao Li, JingJing Xie, Zhoumianze Liu, Fangzhi Xu, Kanzhi Cheng, Qingyun Li, Yian Wang, Yu Qiao, Zun Wang, and Zichen Ding. OS-Symphony: A holistic framework for robust and generalist computer-using agent, 2026a.

Leyang Yang, Ziwei Wang, Xiaoxuan Tang, Sheng Zhou, Dajun Chen, Wei Jiang, and Yong Li. ProBench: Benchmarking GUI agents with accurate process information. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40(32):27547–27555, 2026b. doi: 10.1609/aaai.v40i32.39974.

Yan Yang, Dongxu Li, Yutong Dai, Yuhao Yang, Ziyang Luo, Zirui Zhao, Zhiyuan Hu, Junzhe Huang, Amrita Saha, Zeyuan Chen, Ran Xu, Liyuan Pan, Caiming Xiong, and Junnan Li. GTA1: GUI test-time scaling agent. In *The Fourteenth International Conference on Learning Representations*, 2026c.

Bofei Zhang, Zirui Shang, Zhi Gao, Wang Zhang, Rui Xie, Xiaojian Ma, Tao Yuan, Xinxiao Wu, Song-Chun Zhu, and Qing Li. TongUI: Internet-scale trajectories from multimodal web tutorials for generalized GUI agents. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40 (15):12367–12375, 2026. doi: 10.1609/aaai.v40i15.38229.

Chi Zhang, Zhao Yang, Jiaxuan Liu, Yanda Li, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. AppAgent: Multimodal agents as smartphone users. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, 2025.

Henry Hengyuan Zhao, Kaiming Yang, Wendi Yu, Difei Gao, and Mike Zheng Shou. WorldGUI: An interactive benchmark for desktop GUI automation from any starting point. *arXiv preprint arXiv:2502.08047*, 2025.

Longtao Zheng, Rundong Wang, Xinrun Wang, and Bo An. Synapse: Trajectory-as-exemplar prompting with memory for computer control. In *International Conference on Learning Representations*, 2024.

A Building and Using the Software Model

Section 3 formalizes the software model $(\mathcal{F}, \mathcal{G}, \mathcal{M})$; this appendix details its construction pipeline, shows a fragment of a learned model, specifies the runtime loop, and defines the ablation variants.

A.1 From Video to Operational Semantic Units

Keyframe filtering. Most raw frames of an instructional video show passive cursor movement or transient loading animations. We sample the stream and score adjacent frame pairs with a composite similarity metric: a blockwise $1 - \mathrm{SSIM}$ term (Wang et al., 2004) detects macro-level visual change (window transitions, dialog pop-ups), and a Levenshtein distance (Levenshtein, 1966) over OCR-extracted text detects textual updates that leave the layout intact (editing a configuration value). A frame pair is retained only if either term or the composite score exceeds its threshold and the frame contains enough recognized characters, which suppresses spurious triggers on blank or loading screens.

Perception enhancement. A static screenshot rarely conveys the expert rationale behind an interaction, so each keyframe is enriched twice before abstraction. Narration alignment transcribes the audio track with Whisper (Radford et al., 2023), fuses it with OCR of burned-in subtitles, and aligns the text to keyframe timestamps, so the demonstrator's stated intent is available downstream. Spatial scaffolding overlays a coordinate grid on the image before VLM processing (Lei et al., 2025), which markedly improves coordinate grounding.

Abstraction. Two VLM queries run concurrently on every enriched keyframe pair, each carrying a short history of recent actions for temporal coherence. An *abstraction proxy* compares the screenshot immediately before the action with the one immediately after it and formalizes the atomic action a, its action-centric semantic effect s_a, and the functional intent i it serves. A *description proxy* operates on the resulting screenshot alone, independent of any action semantics, and produces the debiased, state-centric description d that later grounds state merging in $\mathcal{G}$.

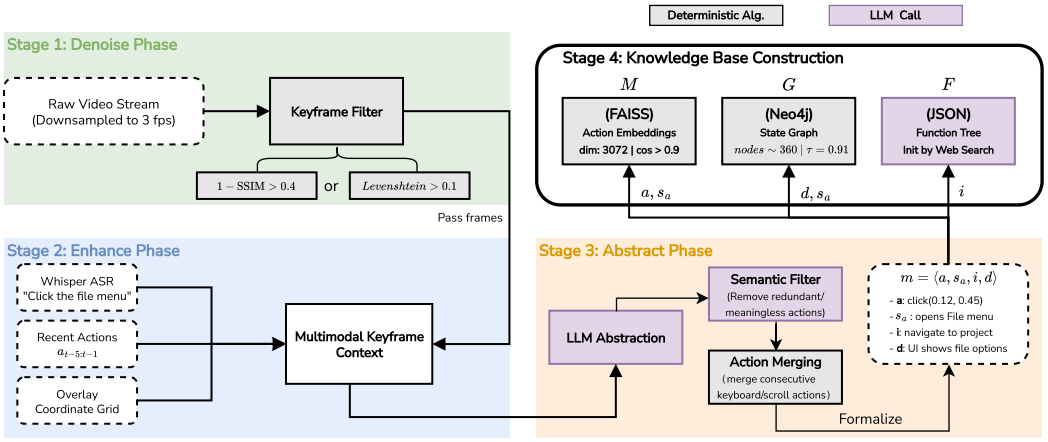


Figure 3: Detailed VideoParser construction pipeline. Denoising, enhancement, and abstraction convert raw instructional videos into operational semantic units $m = \langle a, s_a, i, d \rangle$, which populate $\mathcal{M}$ (FAISS vector store), $\mathcal{G}$ (Neo4j graph database), and $\mathcal{F}$ (JSON tree). Purple nodes are LLM calls, gray nodes deterministic operations; hyperparameter values are shown inline.

Table 3: Serialized format of an operational semantic unit. The upper block is the four-tuple $m = \langle a, s_a, i, d \rangle$; the lower block holds trajectory-level context and multimodal grounding signals.

Field	Description
`action`	Atomic physical operation a
`coordinate`	Coordinates of the above action
`semantic_action`	Action-centric semantic effect s_a describing the action at the semantic level
`intention`	Functional intent i served by the action
`state_evidence`	Debiased state-centric screen description d, the basis for building the state graph
`current_id`	Keyframe index within the source video
`previous_scene`	Description of the screen immediately before the action
`current_scene`	Description of the screen immediately after the action
`transcript`	Timestamp-aligned narration transcript, when available
`subtitle`	Subtitles detected in the video, if any

Action merging. Raw per-frame action streams are heavily redundant. An LLM pass filters semantically empty events (loading delays, noise-induced false positives, duplicate interactions from prolonged mouse clicks) in bounded temporal batches. A deterministic pass then consolidates mechanical fragmentation: consecutive same-direction scroll events at adjacent coordinates become one scroll action, and sequential keystrokes into one input box become one text string. Table 3 lists the serialized fields of the resulting unit $m = \langle a, s_a, i, d \rangle$, and Figure 3 shows the whole pipeline with its hyperparameters.

A.2 Assembling the Three Model Layers

Each layer lives in a store optimized for its query pattern: $\mathcal{M}$ in a vector index for nearest-neighbor retrieval, $\mathcal{G}$ in a graph database for shortest-path navigation, and $\mathcal{F}$ as a hierarchical tree for constrained intent enumeration. Algorithm 2 states the construction pipeline, and Table 4 records where each part of the model comes from.

Two entries in $\mathcal{M}$ are treated as the same operation when the cosine similarity of their semantic descriptions exceeds 0.9; instead of a redundant entry, the system increments an occurrence counter, so operations corroborated across several demonstrations carry a higher reliability prior. In $\mathcal{G}$, pixel comparison is deliberately avoided because functional interfaces fluctuate with user data, transient dialogs, and version drift; two debiased descriptions d collapse into one node when their cosine

Algorithm 2 Software model construction pipeline

Input: instructional video set V for a target application
$\mathcal{F} \leftarrow$ InitFunctionTree() {category skeleton from the official documentation}
for all $v \in V$ **do**
  $K \leftarrow$ KeyFrames(v); $K \leftarrow$ Enrich(K)
  $A \leftarrow$ Abstract(K); $A \leftarrow$ MergeActions(A)
  for all $m = \langle a, s_a, i, d \rangle \in A$ **do**
    UpsertAction$(\mathcal{M}, m, v)$ {cosine similarity $> 0.9 \Rightarrow$ increment occurrence count}
    $u, v' \leftarrow$ MergeState$(\mathcal{G}, d, v)$ {cosine similarity $> 0.91 \Rightarrow$ collapse to one node}
    $e \leftarrow$ AddEdge$(\mathcal{G},\ u \xrightarrow{s_a} v', v)$ {edge carries its source video v}
    $\mathcal{F} \leftarrow$ ExtendFunctionTree$(\mathcal{F}, i)$
    Tag$(e,$ AchievedFunctions$(i, s_a))$ {materializes operator Ψ}
  end for
end for
return $(\mathcal{F}, \mathcal{G}, \mathcal{M})$

Table 4: Provenance of the information in the software model and at run time. Each row names an artifact, the raw source it is derived from, and the component that produces it.

Artifact	Source	Produced by
Category skeleton of $\mathcal{F}$	Official documentation	LLM (function search prompt, Figure 19)
Leaf functions of $\mathcal{F}$	Intents i of video units	LLM (function extraction and detection prompts)
Nodes of $\mathcal{G}$	Video keyframes	VLM description proxy yields d; embedding merge at 0.91
Edges of $\mathcal{G}$	Consecutive video units	Deterministic: effect s_a between merged nodes
Tags $\Psi(f)$	Intent i and effect s_a	LLM judgment at build time
Entries of $\mathcal{M}$	Video units $\langle a, s_a, i, d \rangle$	VLM abstraction proxy; embedding merge at 0.9
Anchor node at run time	Runtime screenshot	Same debiasing prompt as d, nearest node of $\mathcal{G}$
Executed coordinates	Runtime screenshot	Executor prediction refined by the Grounder

similarity exceeds 0.91. Every edge is tagged with the function or functions whose transition it accomplishes, which materializes the operator Ψ at build time rather than computing it at query time. $\mathcal{F}$ is built in two stages: an initialization phase queries an LLM with the official application documentation to establish the category skeleton, and a video-integration phase decides for each extracted trajectory segment whether it completes a known function or reveals a new one, extending the tree accordingly. For the *w/o Paired Video* ablation, we reconstructed the model once from existing operational semantic units, recording the contributing video identifiers for each node, edge, and action entry, including merged entries with multiple sources. The initial model build did not retain this per-video provenance; the reconstruction enables task-specific masking without rerunning video extraction (Appendix A.5).

A.3 Empirical Analysis of a Learned Software Model

Figure 4 shows a model fragment learned from Zettlr demonstrations. Read alongside Figure 1, it clarifies one point: a function maps not to a *node* but to a *set of transition edges*. The leaf function "Open Assets Manager" is tagged onto the transition that invokes the dialog from the main workspace; $\Psi(f)$ is that edge, and reaching f means navigating to the edge's source node, the state in which the central Markdown editor is active and the top menu bar accessible, and then taking the edge. Architecturally, all candidate outputs of the Planner are constrained by $\mathcal{F}$, all reference paths of the Reflector are queried from $\mathcal{G}$, and all action precedents of the Executor are retrieved from $\mathcal{M}$; only the Grounder operates independently of the layers, relying on the live screen, which keeps demonstration-resolution mismatches from corrupting physical execution.

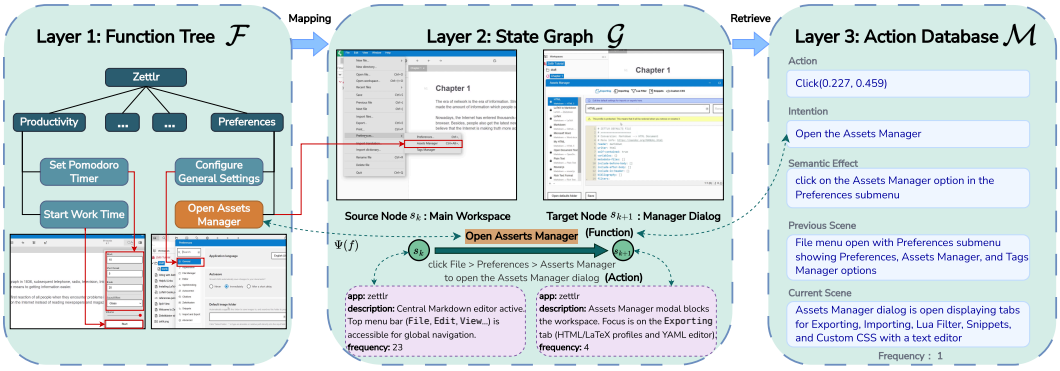


Figure 4: Model fragment learned for Zettlr. Left: function tree $\mathcal{F}$, whose leaves are the only outputs the Planner may emit. Center: state graph $\mathcal{G}$, whose nodes are debiased screen descriptions d; the highlighted edge carries the tag of the function it completes, and tagged edges collectively define $\Psi(f)$. Right: the $\mathcal{M}$ entry retrieved for the target edge, with action semantics, intent, and state context.

A.4 Execution Loop Details

The execution loop is Algorithm 1. Its query primitives have the following signatures, where V and E denote the node and edge sets of $\mathcal{G}$, $\mathcal{P}$ the set of paths over $\mathcal{G}$, and $\mathcal{I}$ the space of textual intents:

$$\begin{aligned} \textsc{Locate} &: \mathrm{Leaf}(\mathcal{F}) \to 2^E, \\ \textsc{Navigate} &: V \times 2^E \to \mathcal{P} \cup \{\varnothing\}, \\ \textsc{Retrieve} &: \mathcal{I} \to 2^{\mathcal{M}}. \end{aligned} \tag{1}$$

$\textsc{Navigate}(v, E')$ returns a shortest path from v to the source node of an edge in E'; that tagged edge represents the transition that completes the target function.

Screen-to-node anchoring. Anchoring maps the live screen to a node of $\mathcal{G}$. The screen is described with the same debiasing prompt that produces d during construction, the description is embedded, and the nearest node by cosine similarity becomes the anchor. If $\textsc{Navigate}$ from that anchor returns no path, the target may be unreachable from that node in the directed graph rather than the anchor being wrong, so the agent retries with the next-nearest node, trading some similarity for reachability. When no candidate anchor yields a path, the screen is treated as unmodeled for that step: no reference path is built, execution proceeds on retrieved precedents alone, and anchoring is re-attempted at every subsequent step. This failure mode degrades accuracy for the affected steps without producing a hard halt.

Reflector verdicts and interruption policy. The three verdicts act on the reference position and the checkpoint history as follows. *On-track* advances the reference position to the matched node v' and records that node as a checkpoint for future rewinds. *Uncertain*, the default, leaves the reference position unchanged and records no checkpoint; execution proceeds, with the Executor predicting from the live screen. *Off-track* triggers the rewind of Algorithm 1, line 8, and is the only verdict accompanied by a written account of the deviation. The Reflector is prompted to issue *off-track* only under two conditions: the screenshot shows an unexpected dialog, wrong menu, or error state inconsistent with the instruction, or a directly visible action violates an explicit hard constraint of the instruction. Checkpoint records are scoped to the active reference path and reset whenever a new path is constructed.

Robustness via conservative error detection. Backtracking relies on timely detection, yet the Reflector is itself a fallible model, so the loop is designed so that neither false positives nor false negatives are catastrophic. False alarms are the costlier error, because they rewind an agent making valid progress; the Reflector is therefore configured conservatively and reports deviations only with explicit evidence, which is why *uncertain* dominates its verdicts on injected errors (Table 15). A

missed deviation is not lost: the Executor re-examines the current screen at every step rather than committing to the reference path, so an unflagged error can still surface as a mismatch later and be repaired.

Iterative spatial grounding and contextual memory. The Grounder regulates action *localization*, not action *selection*. It crops a window centered on the Executor's predicted coordinate, scaled by the Executor's coarse target-size estimate, magnifies the crop, performs a second coordinate prediction, and maps the result back to full-screen space. Drag interactions ground their start and end coordinates independently; hotkey operations bypass grounding. Any failure falls back to the Executor's ungrounded coordinate, so spatial refinement can degrade in precision but never block execution. A shared memory keeps a sliding window of recent screens, plan status, and actions, together with a running summary updated at every step, which preserves progress beyond the active window.

A.5 Ablation Variant Specification

Each variant of Table 2 changes exactly one thing relative to the full system; run-time parameters (Table 11) are unchanged.

w/o Video Guidance removes $(\mathcal{F}, \mathcal{G}, \mathcal{M})$ entirely, including the documentation-derived skeleton of $\mathcal{F}$, and keeps the four roles. The Planner decomposes the instruction into free-form subgoals because no leaves exist to select from; the Reflector judges progress against the active subgoal's text because no reference path exists, and it still emits verdicts and the intent query; RETRIEVE returns nothing, so the Executor predicts from the screen, the plan, and the history alone; the Grounder is unchanged.

w/ Flat Description restores the video knowledge as the LLM-generated description of the instructional video retrieved for the task, the same text the GPT-5.4 + Video Desc. baseline receives. The description replaces the function tree in the Planner's prompt, the reference path in the Reflector's prompt, and the retrieved precedents in the Executor's prompt; no graph exists, so no anchoring or rewind can occur, and the variant isolates the representation of the same video content.

w/o Paired Video uses a provenance-aware model rebuilt once from the existing operational semantic units, with each node, edge, and action entry recording its contributing source videos. This one-time reconstruction does not rerun VideoParser and is shared across all evaluation tasks; it is not a separate rebuild for each paired demonstration. For each task, the harness masks model components attributed to its paired video while retaining knowledge contributed by the application's other demonstrations; model traversal, thresholds, and runtime settings are otherwise unchanged. The one-time reconstruction took a few minutes of wall-clock time under concurrency and about 3.32 USD in API cost.

w/o Grounder executes the Executor's coarse coordinate directly, without cropping and re-prediction.

w/o Reflector removes the Reflector, so no verdict is issued, no checkpoint is recorded, and no rewind can fire; the reference path is still supplied to the Executor, which advances along it by its own judgment, and the intent query for RETRIEVE is emitted by the Executor.

A.6 Software Model Diagnostics

The software model is an induced semantic abstraction rather than a uniquely labeled reconstruction of an application. We therefore characterize it through three directly observable stages: the operational knowledge extracted from videos, the functional coverage retained in the learned model, and the accessibility of that knowledge during execution.

VideoParser extraction. We manually annotate semantic actions in the instructional videos with their action type and approximate temporal interval, and match parser outputs to these references temporally.

Among 931 predicted non-wait actions, the type-agnostic evaluation yields 604 matched actions and 437 missed references, corresponding to 58.0% recall and 64.9% raw precision. After deterministic removal of 68 duplicate clicks and two focus clicks absorbed by subsequent text inputs, precision

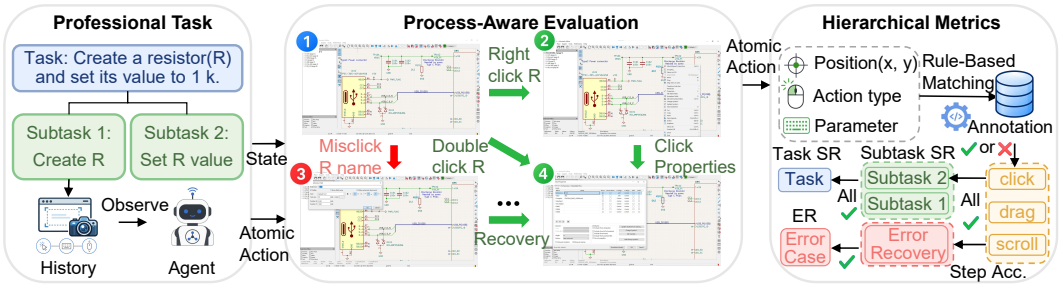


Figure 5: Overview of ProbeGUI on the KiCad task "create a resistor and set its resistance value". Left: tasks decompose into annotated subtasks. Center: every valid action per step is annotated (right-click and double-click are both credited) and an injected error tests recovery. Right: rule-based matching aggregates per-action verdicts into step, subtask, task, and error-recovery metrics.

rises to 70.2%, yielding an F1 of 63.5%. When a match additionally requires the predicted action type to agree with the reference, precision, recall, and F1 are 61.4%, 50.8%, and 55.6%, respectively. These results show that the downstream software model is constructed from useful but imperfect operational extractions rather than oracle trajectories.

Functional knowledge coverage. We next trace how extracted knowledge propagates from demonstrations to evaluation tasks. The learned function library covers 215 of 254 annotated function instances in the demonstration videos (84.6%); 188 instances (74.0%) match directly, with an additional 27 matched through semantic correspondence. Across the evaluation set, functions represented by the demonstration corpus cover 224 of 264 annotated subtask instances (84.8%), establishing the portion of evaluation functionality supported by the available demonstrations. At runtime, the Function Plans cover 201 of the 264 evaluation subtasks under strict semantic matching (76.1% overall), including 201 of the 224 demonstration-supported instances (89.7%). Thus, the overall planning coverage can be decomposed into the availability of relevant knowledge in the demonstration corpus and the Planner's use of that knowledge once available. Because demonstration and evaluation annotations differ in naming and granularity, these quantities measure semantic coverage rather than accuracy against a unique function hierarchy.

State-graph structure and runtime access. Across the ten applications, the learned state graphs contain 360 nodes and 682 directed edges. During construction, 81.6% of incoming states are merged into an existing node, reflecting substantial reuse of recurrent or semantically similar interface states. At runtime, NAVIGATE returns a nonempty path for 99.2% of observed function-switch queries. Because $\mathcal{G}$ is an induced semantic abstraction whose valid state granularity is not unique, we do not define graph or path accuracy against a single gold graph; instead, these statistics characterize its structure and the availability of its navigation interface at runtime.

Taken together, these measurements trace the software model from noisy video extraction to functional knowledge coverage and runtime query availability, complementing the component ablations in Section 5.4.

B PROBEGUI: SPECIFICATION

Figure 5 illustrates the three components of ProbeGUI on one KiCad task: annotated subtasks, per-step multi-path annotation with an injected error, and rule-based aggregation into the four metrics.

B.1 ACTION SPACE

ProbeGUI implements a unified action space of six mouse- and keyboard-based operations covering the principal forms of human–computer interaction (Table 5). Manual verification confirmed that this action space covers all operations required by the professional tasks in ProbeGUI. Every method, MOVI-Agent and every baseline, emits actions through this interface, and all are scored by the rules of Appendix B.3; no LLM judge is involved anywhere in ProbeGUI.

Table 5: Atomic operations in the unified action space of ProbeGUI.

Action	Function	Description
Left click	`left_click(x, y, numclicks=1/2)`	Performs a single or double left click at the specified position.
Right click	`right_click(x, y)`	Performs a right click at the specified position.
Type	`type(x, y, text, enter, overwrite)`	Enters text into a target field, with optional submission and overwriting.
Hotkey	`hotkey(keys)`	Executes a keyboard shortcut using the specified key combination.
Drag	`drag(x1, y1, x2, y2)`	Drags from the specified start position to the end position.
Scroll	`scroll(x, y, direction)`	Scrolls up or down within the specified interface region.

B.2 Annotation Format

Each task is organized into three metadata levels (Table 6). Task-level metadata records the fixed software setting, the paired evaluation and learning videos, the instruction, the ordered functional goals, and task tags, supporting standardized task construction and hierarchical scoring. State-level metadata anchors each observation to the reference execution, associates it with a subtask, and marks injected error states, enabling coherent trajectory simulation and recovery evaluation. The `tips` field specifies state-level execution constraints used to canonicalize evaluation when multiple valid interaction forms or action orders are possible. These constraints may rule out impractical alternatives (e.g., approximating a precise value by dragging a slider) or fix a deterministic order among otherwise equivalent choices (e.g., connecting several components from top to bottom). They define the benchmark-accepted execution protocol rather than the task answer itself, and are provided identically to every evaluated method. Action-level metadata enumerates all valid operations from each state together with their matching constraints and successor states, forming a directed graph for deterministic multi-path matching and fine-grained error diagnosis. Table 7 lists the type-specific parameters annotated for each atomic action.

B.3 Rule-based Matching

For each evaluated step, ProbeGUI compares the predicted action against every `valid_action` of the current annotated state. A prediction that does not conform to the unified action space is labeled an *invalid output*. The step is scored as successful if the prediction fully matches any valid action; otherwise a deterministic diagnosis attributes the failure to action type, position, or non-spatial parameters.

Action-type matching. If no valid action has the predicted action type, the prediction is labeled an *action error*; otherwise every same-type candidate is examined with the position and parameter rules below. This separation distinguishes selecting an incorrect operation from executing the correct operation incorrectly.

Position matching. For left click, right click, type, and scroll, the predicted coordinate must lie within the annotated `bbox`; for drag, both endpoints must fall within `start_bbox` and `end_bbox`. A same-type prediction outside the target region is a *positioning error*. The prompt requires coordinates normalized to $[0, 1]$ by the screen resolution. Before matching, the raw output is validated and its coordinates mapped to $[0, 1]$ by screen-resolution normalization or by 1000-normalization; for fairness, all reported results use screen-resolution normalization.

Parameter matching. After type and position match, the remaining action-specific parameters are verified. For `left_click`, the click count must equal the annotation. For `type`, numeric strings are compared by exact decimal value, and ordinary text is normalized and checked for containment of the annotated text, which allows additional context while requiring the complete target content; the `enter` and `overwrite` flags are enforced only when set to `true` in the annotation. Hotkeys

Table 6: Annotation metadata schema of ProbeGUI.

Metadata field	Description
Task-level metadata	
task_id	Unique identifier of the task.
evaluating_video_file	Reference execution video used to construct the evaluation annotations.
learning_video_file	Paired instructional video provided as the agent's learning resource.
software, version	Application and fixed version in which the task is performed.
instruction	Natural-language task objective and execution constraints.
gt_subtasks	Ordered, independent functional goals used for subtask-level evaluation.
tags	Task attributes, including application domain and difficulty.
annotations	Collection of the state-level annotations defined below.
State-level metadata	
annotation_id	Unique identifier of the annotated interface state.
frame_index, timestamp	Temporal position of the state in the reference video.
screenshot_file	Screenshot representing the observable interface state.
subtask	Functional subtask associated with the current state.
is_trap	Whether the state is an injected error state.
trap_description	Description of the injected error.
tips	State-level execution constraints that canonicalize valid interaction forms or action orders; served identically to every method.
valid_actions	Collection of all valid action transitions under benchmark constraints.
Action-level metadata	
action_id, description	Unique action identifier and its human-readable description.
type	Atomic operation type: left click, right click, type, hotkey, drag, or scroll.
parameters	Type-specific constraints, such as target regions, text, keys, or direction.
next_annotation_id	Successor state after execution; `-1` denotes task completion.

Table 7: Type-specific annotation parameters of the unified action space. Each `bbox` is given as `[x_min, y_min, x_max, y_max]` and specifies the valid region for the operation.

Action	Parameters	Description
Left click	`bbox, num_clicks`	Target region and required number of clicks (one or two).
Right click	`bbox`	Target region for the right-click operation.
Type	`bbox, text, enter, overwrite`	Target field, accepted text, optional submission and overwriting.
Hotkey	`keys`	Required keyboard key combination.
Drag	`start_bbox, end_bbox`	Target regions for the start and end positions.
Scroll	`bbox, direction`	Target scrolling region and required direction (`up` or `down`).

are compared as order-insensitive normalized key sets, with aliases such as `control` mapped to `ctrl`. For `scroll`, the direction must match exactly. Right-click and drag actions have no regular non-spatial parameters and are determined by position matching. A prediction with the correct type and position but a wrong click count, text, key combination, or direction is a *parameter error*; when several same-type candidates fail, a parameter error from a spatially matched candidate takes precedence over positioning errors from the others.

B.4 Continue-on-Miss Scoring

ProbeGUI keeps a coherent execution trajectory after both correct and incorrect predictions, so that a single early mistake does not invalidate all subsequent steps, and applies the same continuation rule to every method.

Table 8: GUI benchmark comparison. "Inst. video" is task-aligned instructional guidance; "Process-aware" scores execution paths and recovery; "Env.-indep." requires no live application.

Benchmark	# Prof. apps	Inst. video	Exec. protocol		Eval. granularity			
			Process-aware	Env.-indep.	Atomic	Subtask	Task	Recovery
Online benchmarks								
OSWorld (Xie et al., 2024)	2	×	✓	×	×	×	✓	×
WorldGUI (Zhao et al., 2025)	1	✓	✓	×	×	×	✓	×
ScienceBoard (Sun et al., 2026)	6	×	✓	×	×	×	✓	×
CutVerse (Hu et al., 2026)	7	×	✓	×	×	✓	✓	×
Offline benchmarks								
GUI-360° (Mu et al., 2025)	0	×	×	✓	✓	×	×	×
VideoGUI (Lin et al., 2024)	7	✓	×	✓	✓	✓	✓	×
ProbeGUI	**10**	✓	✓	✓	✓	✓	✓	✓

Trajectory continuation. When a prediction matches a valid action, it is canonicalized to the matched annotation before entering the action history; for spatial actions the predicted coordinate is replaced by the center of the annotated bounding box. The evaluator then transitions to the corresponding successor state. When no valid action matches, the step is scored as incorrect and the first annotated valid action serves as the default continuation, following prior offline benchmarking practice (Im et al., 2025); its successor state becomes the next observation, so later steps remain conditioned on a valid trajectory.

Injected-error recovery. For tasks with injected errors, the first state marked `is_trap=true` is the normal pre-error state; the agent's prediction there is skipped and the annotated `valid_action`, which is the injected erroneous operation, is executed directly. Recovery evaluation begins from the second state marked `is_trap=true` and measures whether the agent leaves the error state and rejoins the normal annotated trajectory, that is, reaches a state with `is_trap=false`. Continue-on-miss does not apply inside a recovery episode: a task counts as recovered only if every step of the episode matches an annotated recovery action, and it counts once, whatever the number of error-recovery states the episode spans, so ER is a fraction over the 48 error-injected tasks. This design reuses the same state-transition annotations for normal execution and recovery while keeping injected errors out of normal-step scoring.

C ProbeGUI: Statistics and Construction

C.1 Comparison with Related Benchmarks

Table 8 positions ProbeGUI among related GUI benchmarks along the axes discussed in Section 2: professional-application coverage, task-aligned instructional videos, execution protocol, and evaluation granularity. Only ProbeGUI combines process-aware, environment-independent evaluation with all four granularities, including a dedicated error-recovery metric.

C.2 Benchmark Statistics

Task-level statistics. ProbeGUI comprises 86 tasks across ten professional desktop applications in five domains: research, development, engineering, data analysis, and media. Figure 6 (left) shows the distribution of tasks across domains and applications, and Table 9 gives per-application statistics. The benchmark contains 35 easy tasks (40.7%), 32 medium tasks (37.2%), and 19 hard tasks (22.1%); 48 tasks (55.8%) include an error-recovery case.

State-level statistics. ProbeGUI contains 1,033 non-error states and 115 error-recovery states. A full task contains 12.0 non-error states on average and each subtask 3.9; each error-recovery case spans 2.4 error states on average. Figure 6 (middle) shows the distribution of state counts across the three difficulty levels.

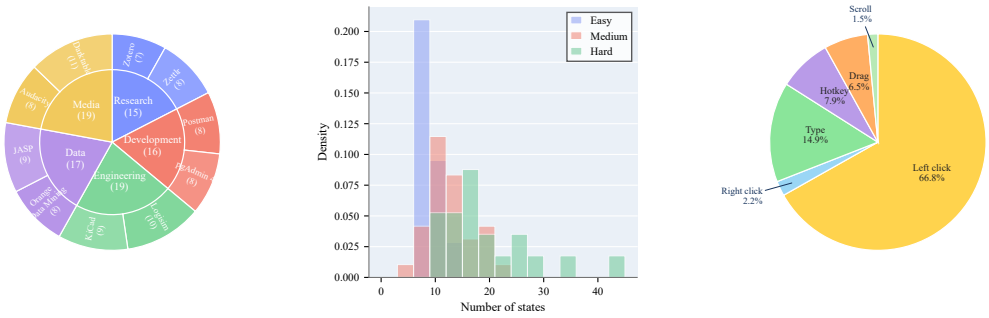


Figure 6: Statistics of ProbeGUI. Left: distribution of tasks across domains and applications, with task counts in parentheses. Middle: distribution of state counts for easy, medium, and hard tasks. Right: distribution of atomic action types.

Table 9: Per-application ProbeGUI statistics. Difficulty columns give within-application task shares; ER is the share of tasks with recovery cases. State counts exclude error states except in the last column, which averages over recovery cases.

Domain	Software (version)	Easy%	Medium%	Hard%	Error-recovery%	# States per full task	# States per subtask	# States per error case
Research	Zotero (8.0.4)	42.9	42.9	14.3	100.0	10.3	4.0	2.0
	Zettlr (4.3.1)	50.0	37.5	12.5	75.0	9.4	3.9	2.0
Development	Postman (12.4.6)	62.5	37.5	0.0	25.0	9.0	2.9	2.5
	pgAdmin 4 (9.14)	37.5	50.0	12.5	37.5	11.8	3.8	2.7
Engineering	Logisim (3.8.0)	40.0	30.0	30.0	90.0	13.7	3.7	2.8
	KiCad (10.0.0)	44.4	22.2	33.3	33.3	18.3	5.7	2.3
Data	Orange Data Mining (3.40.0)	37.5	37.5	25.0	50.0	12.1	3.7	2.8
	JASP (0.96)	44.4	22.2	33.3	33.3	15.3	5.3	3.7
Media	Audacity (3.7.7)	50.0	37.5	12.5	25.0	10.6	3.5	2.0
	Darktable (5.4.1)	9.1	54.5	36.4	81.8	8.9	2.8	2.0
Total		**40.7**	**37.2**	**22.1**	**55.8**	**12.0**	**3.9**	**2.4**

Action-level statistics. ProbeGUI includes 1,574 annotated atomic actions. As Figure 6 (right) shows, `LeftClick` is the most frequent type (66.8%), followed by `Type` (14.9%), `Hotkey` (7.9%), and `Drag` (6.5%); `RightClick` and `Scroll` account for 2.2% and 1.5%.

Video statistics. The corpus contains 172 videos forming 86 demonstration–evaluation pairs, 1h 57m 20s in total. Each pair consists of a concise, action-dense demonstration video, 48m 08s in total across the corpus, from which operational semantics are extracted to build the software model, and an evaluation video, 1h 09m 12s in total, from which the ground-truth annotation is derived. The aggregate demonstration duration per application is compact, from 3m 06s for Zotero to 8m 38s for KiCad.

C.3 Construction Protocol

Data collection settings. Tasks were curated from the official documentation and community forums of each application. All videos were recorded on Windows with OBS Studio (OBS Studio, 2024). Within each of the 86 pairs, the target-task video and its paired instructional video share one resolution: 42 pairs (48.8%) at 1280×720, 7 (8.1%) at 1920×1080, and 37 (43.0%) at 2560×1440; the resolution is fixed per application. Two annotators jointly constructed each task and produced its videos and annotations, all of which underwent a second-round review. All recordings were created in-house on locally installed copies of the pinned application versions, contain no personal data, and include no third-party video content. The benchmark, videos, and annotations will be released under CC BY 4.0.

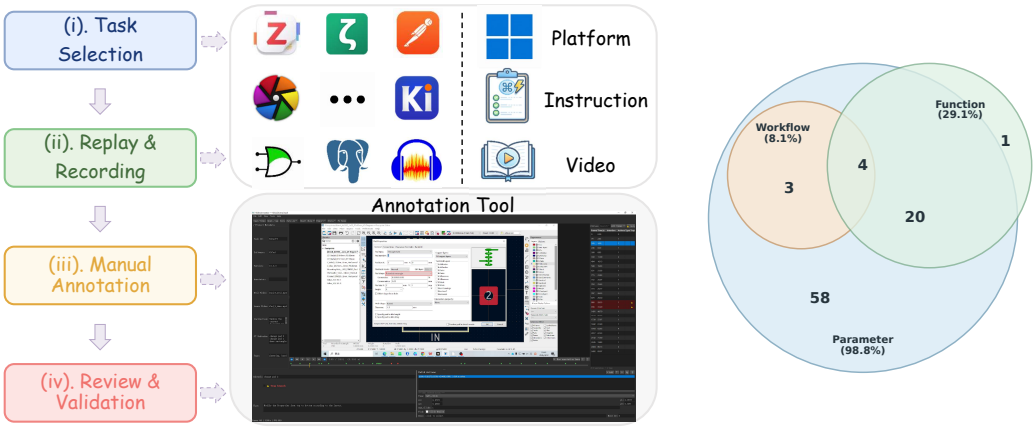


Figure 7: ProbeGUI construction and demonstration–task overlap. Left: task selection, replay and recording, manual annotation, and validation. Right: joint distribution of parameter, function, and workflow differences in the 86 demonstration–evaluation pairs; counts are non-exclusive.

Construction pipeline. Figure 7 (left) shows the four stages. (i) *Task selection* screens candidate tasks from the sources above to retain tasks that reflect realistic professional workflows and cover representative application functions; for a subset of tasks involving operations prone to execution errors, realistic error-injection cases are designed. (ii) *Replay and recording* executes each selected task in the application and records its complete trajectory, including the injected erroneous action and the recovery where applicable. Each target task is also paired with an instructional video of a similar task that may differ in parameters, invoked function, workflow structure, or a combination thereof, so that it provides procedural guidance without showing an identical target solution; in Zotero, for example, the instructional video changes highlight colors in the advanced preferences, whereas the target task changes the trash retention policy in the same panel. (iii) *Manual annotation* uses a dedicated tool to decompose each evaluation trajectory into interface states, atomic actions, and functional subtasks, and to annotate valid transitions, alternative paths, and error-recovery cases. (iv) *Review and validation* re-examines task specifications, videos, and annotations, then manually replays the annotated workflows to verify correctness, consistency, and reproducibility.

C.4 Demonstration–Task Overlap

Demonstrations are meant to convey procedural priors, not executable solutions that can be replayed verbatim, so we characterize the difference between each demonstration and its paired evaluation task along three non-exclusive dimensions. A *parameter difference* alters task-specific values, options, names, targets, inputs, or manipulated objects while preserving the invoked function and overall procedure. A *function difference* changes the functional module invoked during execution, including different functionalities accessed within the same or a similar panel. A *workflow difference* changes the abstract execution structure by adding, removing, or reordering semantically distinct stages. Table 10 illustrates each dimension with a representative pair.

Figure 7 (right) gives the joint distribution. Every evaluation task differs from its paired demonstration in at least one dimension: parameter differences occur in 85 pairs (98.8%), function differences in 25 (29.1%), and workflow differences in 7 (8.1%); 27 pairs (31.4%) differ in at least two dimensions and 4 (4.7%) in all three. A difference in any dimension changes the execution path, the interface states encountered, the action parameters required, or several of these. In the parameter-level example of Table 10, creating three columns rather than two adds column-creation steps to the path, the different schema produces distinct intermediate states, and the table and column names, types, and constraints require different action parameters. Because path and states determine the agent's input context while parameters determine the required output, these differences affect both what the agent sees and what it must produce, which precludes direct replay of the paired demonstration.

Table 10: Representative differences between paired demonstrations and evaluation tasks. Each row illustrates one focal dimension; the dimensions are non-exclusive and can co-occur within a pair.

Dimension	Demonstration	Evaluation task	Difference
Parameter	Create a `student` table under `demo_database.public` with three columns: `id` as the integer primary key, `sex` as `bit`, and `name` as `text`.	Create a `customers` table in a different database with only two differently named and defined columns.	The two tasks share the table-creation procedure but differ in their database and table targets, column count, names, types, and constraints.
Function	Change the endpoint to `/delete` and configure API Key authentication.	Change the endpoint to `/get` and configure Basic Auth.	Both tasks follow the same request-authentication workflow but invoke different functions, leading to distinct interface states and operation sequences; their endpoint parameters also differ.
Workflow	Place VCC, a resistor, an LED, and GND in a 2×2 layout in the order VCC $\rightarrow$ R $\rightarrow$ LED $\rightarrow$ GND, rotate the LED by 90°, and connect the components.	Place the same component types vertically in the order VCC $\rightarrow$ LED $\rightarrow$ R $\rightarrow$ GND, and connect them with a different wiring sequence.	Component placement order, spatial layout, and wiring sequence differ, resulting in distinct execution workflows.

D Further Experimental Details

D.1 Experimental Setup

Table 11 lists every hyperparameter that influences model behavior; operational flags that do not affect model output (such as the presence of an audio track) are omitted. The two similarity thresholds are the only parameters that reshape the model structure, and a single fixed pair of values is used across all applications and experiments without per-application tuning. Across the ten applications, the learned state graphs contain 360 nodes and 682 edges in total, with 81.6% of incoming states merging into an existing node. This merge rate reflects three sources of state equivalence: repeated visits to the same interface state within a trajectory, equivalent states reached across different tasks or paths, and minor visual variation between functionally identical states (for example, during scrolling). Appendix A.6 reports these structural statistics; the 99.2% nonempty-path rate of NAVIGATE characterizes runtime path availability rather than the semantic correctness of the merged graph. Table 12 lists the exact model snapshots used in all reported results.

Execution environment. All experiments ran on Windows with Python 3.14. Model construction used a single NVIDIA RTX 4060 GPU for local acceleration only; the pipeline also runs on CPU.

Baseline adaptation and input parity. Every evaluated agent, MOVI-Agent included, operates under the same observation format (Section 5.1) and the action interface of Table 5. The harness serves each method the instruction, the state's `tips` constraints, the action history, and the screenshots; GPT-5.4 + Video Desc. additionally receives the retrieved video description, and MOVI-Agent additionally consults its software model. The Planner prompt (Figure 11) retains legacy wording referring to a "possibly correct solution from the Internet"; no web search is performed and no Internet solution is supplied in the reported MOVI-Agent runs. Standalone VLM baselines use one uniform system prompt (Figure 15), so the model is the sole variable. Framework baselines (Agent S3, OS-Symphony, and UiPath) keep their native framework logic, internal prompts, and output conventions; a lightweight adapter maps each framework's structured action objects, not free-form model text, onto the unified action space, and GPT-5.4 serves as the backend of every framework for a controlled comparison. Standalone models, UI-TARS included, are therefore held to the prompt's output format like every other VLM, and the exploration-based knowledge systems of Section 2 could not be included because they target Android and provide no desktop implementation. UiPath runs in its native Screen Agent configuration with all internal logic unmodified and relies on the

Table 11: Hyperparameters used by MOVI-Agent; common decoding parameters are matched across methods where supported.

Setting	Value	Function / description
Model construction		
Frame resolution	1920×1080	Standard resolution to which all input videos are normalized
`sample_fps`	3	Frame sampling rate prior to keyframe filtering
`context_limit`	5	Number of recent actions retained for intent inference
`merge_sequence_limit`	30	Maximum action sequence length per LLM cleanup batch
`state_merge_threshold`	**0.91**	Cosine similarity threshold for merging interface screens
`action_layer_extend_threshold`	**0.9**	Similarity threshold for deduplicating redundant operations
Embedding model	`text-embedding-3-large`	
Agent execution		
`memory_window`	2	Historical screenshots retained alongside the active screen
`max_tokens`	8192	Output token budget per API call
`temperature`	1	Sampling temperature
`top_p`	0.9	Nucleus sampling parameter
Step budget T	50	Maximum agent steps per task (Algorithm 1)
Retrieval top-k	3	Action precedents returned per RETRIEVE query

Table 12: Exact model snapshots used in all reported evaluation results.

Model identifier	Snapshot date
`openai/gpt-5.4`	Mar 6, 2026
`anthropic/claude-sonnet-4.6`	Feb 17, 2026
`qwen/qwen3-vl-235b-a22b-instruct`	Mar 2025
`moonshotai/kimi-k2.6`	Apr 20, 2026
`minimax/minimax-m3`	Jun 1, 2026
`bytedance/ui-tars-1.5-7b`	Jul 23, 2025

adapter only for action-space alignment. Where a framework's design includes a coder-style component that emits directly executable scripts (for example, `pyautogui`) rather than discrete UI actions, that component is disabled because its code output cannot be mapped onto the action space.

D.2 Offline Scoring Validation

To verify that ProbeGUI's offline scoring reflects real system behavior, we replayed the logged actions of 60 agent trajectories in the live applications and compared the physical outcomes with the benchmark's automated verdicts; the verdicts agreed in all 60 cases. The replay executes the actions an agent actually emitted; it does not let the agent observe and react to the live application, so it validates the scoring rule rather than measuring closed-loop performance.

Validation sample. The sample was built in two stages under predefined coverage criteria. At the task level, we selected three distinct tasks from each of the ten applications, 30 tasks in total, covering the difficulty levels available within each application and including at least one task per application with an injected error-recovery case. At the trajectory level, we selected two executions per task, one successful and one failed whenever both outcomes were available. The 60 trajectories cover MOVI-Agent, every baseline, and every ablation variant across applications, difficulties, outcomes, and recovery conditions (Table 13).

Table 13: Coverage and statistics of the offline scoring validation sample. Failed trajectories outnumber successful ones because many tasks, especially hard tasks, were failed by all methods; under the sampling rule, 25 successful trajectories is the maximum attainable count.

Category	Count	Coverage / ratio
Coverage criteria		
Applications	10 / 10	100.0%
Difficulty levels	3 / 3	100.0%
Evaluated methods	16 / 16	100.0%
Sample statistics		
Selected tasks	30	–
Selected trajectories	60	–
Recorded action steps	611	–
Unique subtasks	89	–
Tasks with error recovery	19 / 30	63.3%
Successful trajectories	25 / 60	41.7%
Failed trajectories	35 / 60	58.3%
MOVI-Agent trajectories	4 / 60	6.7%
Baseline trajectories	37 / 60	61.7%
Ablation-variant trajectories	19 / 60	31.7%

Live replay protocol. For each trajectory, we replayed all atomic actions in the live application with PyAutoGUI and compared the observed behavior with the offline verdicts. For step accuracy, after each action we checked whether the resulting interface state matched the annotated reference state, and separately whether the action made valid progress toward the objective, so that a feasible path omitted from the annotation would not be penalized unnoticed. For subtask success, we verified whether the execution completed the functional goal of the subtask. For task success, after all actions we compared the final application state and output with the instruction. Error recovery was validated like subtask success, by checking whether the execution returned from the injected error state to a valid trajectory. Two authors conducted all replays and judgments independently and cross-checked them.

D.3 Runtime Query Statistics

NAVIGATE statistics. Table 14 reports per-application NAVIGATE statistics and logged path-build events relative to planned functions. Logisim and KiCad have the longest navigation paths, 12.70 and 10.78 graph hops per task, consistent with tasks that cross more functional modules; Audacity and Postman have the shortest, 4.38 and 5.75. Across archived function-switch queries, 99.2% yield a nonempty path. The archived logs additionally record 204 path-build events for 225 planned functions (90.7%); this is an execution-log ratio, not a semantic correctness judgment against annotated subtasks.

Reflector statistics. Table 15 shows that the Reflector stays selective during normal execution and that its verdict distribution shifts sharply on injected errors. Among 973 non-error steps, 81.5% receive `on_track` and only 0.9% an erroneous `off_track`, so unnecessary rewinds are rare. On the 48 error-injected tasks, it issues a verdict other than `on_track` in 79.2% of cases, 64.6% `uncertain` and 14.6% `off_track`; the jump from 18.5% non-`on_track` verdicts on normal steps to 79.2% on injected errors shows that the verdicts discriminate anomalous states without disrupting valid execution, although only the 14.6% `off_track` verdicts trigger a rewind. The verdicts are also predictive of outcomes. On non-error steps, step accuracy falls from 82.3% (653/793) under `on_track` to 70.8% (121/171) under `uncertain` and 44.4% (4/9) under `off_track`. In recovery, only `off_track` triggers the rewind: recovery reaches 85.7% (6/7) when the rewind fires, 64.5% (20/31) under `uncertain`, where the Executor repairs the deviation from the live screen without a rewind, and 50.0% (5/10) when the deviation is missed as `on_track`. Tasks with a non-`on_track` verdict recover in 68.4% (26/38) of cases. This split is observational, since the verdict is not assigned at random, but it separates detection from rewind within the full system in a way the *w/o Reflector* ablation, which removes both, cannot.

Table 14: Per-application NAVIGATE statistics. Navigation length counts graph hops traversed per task (mean, minimum, and maximum). Path-build / plan reports logged path-build events divided by planned functions; it is not semantic matching accuracy against annotated subtasks.

Software	Mean nav. len.	Min	Max	Path-build / plan
Zotero	9.14	4	13	92.3% (12/13)
Zettlr	7.25	4	14	92.9% (13/14)
Postman	5.75	2	12	76.2% (16/21)
pgAdmin 4	8.38	3	13	100.0% (23/23)
Logisim	12.70	5	26	92.5% (37/40)
KiCad	10.78	3	23	100.0% (29/29)
Orange Data Mining	14.25	8	19	84.0% (21/25)
JASP	9.00	3	23	75.0% (15/20)
Audacity	4.38	2	8	88.2% (15/17)
Darktable	8.09	2	17	100.0% (23/23)
Overall	**9.05**	**2**	**26**	**90.7% (204/225)**

Table 15: Reflector outcomes on 973 non-error steps and 48 error-injected tasks. Share is the proportion of each verdict within the evaluation setting. Outcome is step accuracy for non-error steps and error-recovery rate for error-injected tasks; "Not on-track" combines `off_track` and `uncertain`. Only `off_track` triggers a rewind.

Evaluation	Verdict	n	Share	Outcome
Non-error steps	`on_track`	793	81.5%	82.3%
	`uncertain`	171	17.6%	70.8%
	`off_track`	9	0.9%	44.4%
Error-injected tasks	`on_track`	10	20.8%	50.0%
	`uncertain`	31	64.6%	64.5%
	`off_track`	7	14.6%	85.7%
	Not on-track	38	79.2%	68.4%

D.4 PER-TASK PERFORMANCE GAINS

Figure 8 compares the subtask success of MOVI-Agent and OS-Symphony, the strongest baseline, on each of the 86 tasks. MOVI-Agent outperforms OS-Symphony on 43 tasks, ties on 27, and underperforms on 16. The mean improvement is $+0.25$ in Engineering (13 of 19 tasks improve), $+0.20$ in Development (9/16), $+0.20$ in Media (9/19), and $+0.14$ in Research (7/15); Data is the only domain with a negative mean (-0.02), and MOVI-Agent still improves on 5 of its 17 tasks. The magnitude tracks interface complexity: Engineering and Media applications such as KiCad, Logisim, and Darktable place target functions behind deeply nested panels and dense, domain-specific menus, where a persistent model of functions and states pays off most, whereas Orange Data Mining and JASP expose more functionality through directly labeled, shallow controls, leaving less structural ambiguity to resolve. Because a task carries only three to six subtasks, gains take discrete values such as ± 0.20, ± 0.25, ± 0.33, or ± 0.50; the horizontal spread within a column only separates coincident points.

D.5 STATISTICAL SIGNIFICANCE AND RELIABILITY

Statistical procedure. We assess reliability with a paired task-level bootstrap over the 86 tasks (10,000 resamples, fixed seed). Each replicate redraws task indices once and re-aggregates MOVI-Agent and the comparison method over the same sampled tasks, so both are evaluated on an identical task set; every metric is recomputed as a ratio of sums, as in the reported results. Confidence intervals are bias-corrected and accelerated (BCa) at the 95% level. Two-sided p-values are obtained by inverting the bootstrap distribution of the paired difference and corrected for multiplicity across all 60 method-by-metric comparisons with the Benjamini–Hochberg procedure at $\alpha = 0.05$. Table 16 reports every comparison. The procedure quantifies task-sampling variability; run-to-run

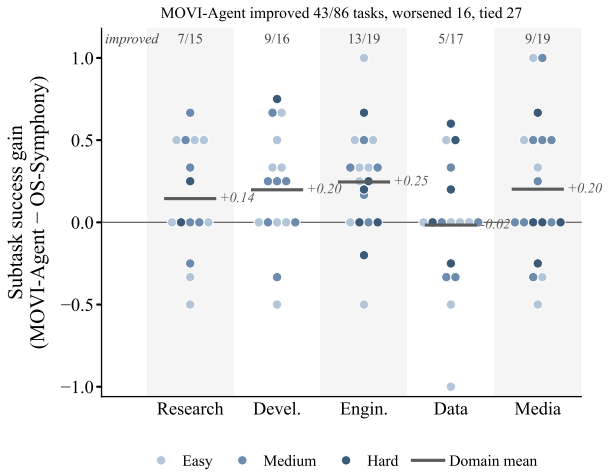


Figure 8: Per-task subtask-success gain of MOVI-Agent over OS-Symphony, grouped by domain and colored by difficulty. Each dot is one task; the rule marks the domain mean.

Table 16: Paired task-level bootstrap differences from MOVI-Agent (percentage points, with 95% BCa intervals); negative values favor the comparison method. MOVI-Agent scores 80.0 [76.7, 82.9], 52.7 [45.7, 59.4], 19.8 [12.8, 29.1], and 62.5 [47.9, 75.6] on Step Acc., Subtask SR, Task SR, and ER, respectively. Benjamini–Hochberg correction covers all 60 comparisons: $^{***}q < 0.001$, $^{**}q < 0.01$, $^{*}q < 0.05$; ns is non-significant.

Method	Step Acc.	Subtask SR	Task SR	ER
External baselines				
UI-TARS-1.5-7B	+73.7 [70.4, 76.8]***	+52.3 [45.5, 58.8]***	+19.8 [12.8, 29.1]***	+56.2 [42.2, 70.6]***
Claude Sonnet 4.6	+68.6 [64.2, 72.6]***	+51.5 [44.4, 58.4]***	+19.8 [12.8, 29.1]***	+52.1 [37.5, 66.0]***
MiniMax M3	+55.6 [51.8, 59.4]***	+49.2 [42.5, 55.7]***	+19.8 [12.8, 29.1]***	+35.4 [18.9, 51.2]***
Qwen3-VL-235B-A22B	+35.2 [31.7, 38.6]***	+42.8 [36.6, 48.5]***	+19.8 [12.8, 29.1]***	+50.0 [34.0, 64.7]***
Agent S3	+27.5 [22.2, 32.4]***	+28.8 [22.3, 35.3]***	+17.4 [8.1, 26.7]***	+22.9 [7.4, 38.9]**
UiPath	+14.9 [10.3, 19.0]***	+17.0 [8.5, 25.0]***	+10.5 [0.0, 19.8] ns	+10.4 [−4.1, 25.0] ns
GPT-5.4	+12.8 [9.3, 16.2]***	+22.3 [14.9, 28.9]***	+14.0 [3.5, 23.3]*	+16.7 [3.9, 30.4]*
Kimi K2.6	+12.2 [8.5, 16.0]***	+21.6 [14.8, 28.2]***	+11.6 [2.3, 20.9]*	+14.6 [0.0, 28.9] ns
OS-Symphony	+11.7 [7.8, 15.6]***	+16.0 [8.3, 23.5]***	+5.8 [−4.7, 15.1] ns	+12.5 [−3.9, 28.6] ns
GPT-5.4 + Video Desc.	+9.9 [6.1, 13.4]***	+14.4 [7.0, 22.3]***	+10.5 [1.2, 19.8]*	+10.4 [0.0, 22.9] ns
Ablations of MOVI-Agent				
w/o Grounder	+7.9 [5.4, 10.6]***	+12.5 [7.1, 17.6]***	+3.5 [−4.7, 10.5] ns	−2.1 [−11.6, 6.7] ns
w/o Video Guidance	+6.0 [3.3, 8.9]***	+7.2 [0.8, 13.3]*	+9.3 [−1.2, 18.6] ns	+14.6 [1.8, 28.9] ns
w/o Paired Video	+3.9 [1.5, 6.5]**	+8.0 [2.6, 13.6]**	+5.8 [−1.2, 12.8] ns	+10.4 [0.0, 23.1] ns
w/ Flat Description	+2.3 [−0.3, 5.0] ns	+2.3 [−4.1, 8.7] ns	+4.7 [−5.8, 15.1] ns	+8.3 [−5.6, 22.7] ns
w/o Reflector	+1.8 [−0.6, 4.3] ns	+2.7 [−3.2, 8.1] ns	−1.2 [−11.6, 7.0] ns	+10.4 [−4.8, 26.1] ns

stochasticity at decoding temperature 1 under pass@1 remains unmeasured, and placing baselines and ablations in one correction family is conservative for the ablations.

Interpreting endpoint-level significance. Task SR and ER remain informative when their differences do not reach significance. Both are stringent long-horizon metrics: completion requires every decision along a trajectory to succeed, and recovery requires a multi-step return to a valid path. Yet each trajectory contributes one binary outcome to the test, 86 observations for Task SR and 48 for ER under pass@1, so the many decisions summarized by an endpoint do not add independent samples. A non-significant endpoint difference therefore reflects limited power, not equivalence of the compared methods, and the hierarchical metrics preserve the step- and subtask-level outcomes that the endpoints aggregate.

Hierarchical evidence for the gains. Against every external baseline, MOVI-Agent's Step Acc. and Subtask SR advantages remain significant after correction; against OS-Symphony they are $+11.7$ points (CI $[7.8, 15.6]$) and $+16.0$ points (CI $[8.3, 23.5]$). Appendix D.4 shows improvements on 43 of 86 tasks spread across domains, so the advantage is not carried by a few favorable cases. Relative to its GPT-5.4 backend, MOVI-Agent significantly improves all four metrics, including Task SR by 14.0 points and ER by 16.7 points. Among the ablations, removing the Grounder, the video guidance, or the paired video costs significant Step Acc. and Subtask SR (Grounder: -7.9 / -12.5; video guidance: -6.0 / -7.2; paired video: -3.9 / -8.0).

Mechanistic support for the remaining ablations. Two ablation differences do not survive correction, and for both the runtime diagnostics supply mechanism-level evidence in the direction of the aggregate gap. Relative to *w/ Flat Description*, the structured model improves all four metrics, including Task SR by 4.7 and ER by 8.3 points; Appendix A.6 shows that Function Plans cover 76.1% (201/264) of all evaluation subtasks and 89.7% (201/224) of demonstration-supported subtasks; NAVIGATE returns a path for 99.2% of function switches. Table 17 shows the full model lowering `FarMiss` from 12.6% to 10.6% and parameter errors from 2.0% to 1.5% while action errors stay at 5.0%, a selective improvement consistent with more accurate target identification rather than a generic shift. For the Reflector, the 10.4-point ER gain is accompanied by Table 15: 0.9% false `off_track` on normal steps, a non-`on_track` verdict on 79.2% of injected cases, and recovery rising from 50.0% to 64.5% to 85.7% with verdict strength. By contrast, removing the Grounder raises ER by only 2.1 points; the Grounder refines localization and has no role in detecting deviations or triggering recovery, so this one-case reversal has no mechanism behind it and is read as sampling variation.

D.6 FULL ERROR-TYPE BREAKDOWN

Table 17 reports the complete error distribution for every method and ablation variant, with `Position` errors split into `NearMiss` and `FarMiss`. A `NearMiss` places the predicted coordinate outside the ground-truth bounding box but within a 20% expanded margin, indicating correct semantic grounding with insufficient spatial precision; a `FarMiss` falls beyond this margin, typically because the agent grounds the action to a semantically incorrect control. Across most methods `FarMiss` substantially exceeds `NearMiss`, so spatial failures arise mainly from acting on the wrong control rather than from small deviations around the right one. MOVI-Agent reduces `FarMiss` to 10.6%, against 18.0% for OS-Symphony, while removing the Grounder or the video guidance raises it to 16.4% and 15.3%: video-derived knowledge identifies the intended function and control, and the Grounder aligns that target with its location on the current interface.

D.7 OUTLIER ANALYSIS OF OUTPUT-CONTRACT FAILURES

Two outliers, Claude Sonnet 4.6 and UI-TARS-1.5-7B, fail primarily to satisfy the prescribed output contract: Claude applies coordinate conventions inconsistently, and UI-TARS frequently produces missing or non-parseable actions. We analyze both to clarify what their low step accuracies represent.

Mixed coordinate conventions in Claude Sonnet 4.6. Claude's positioning profile differs sharply from the other agents': its `FarMiss`-to-`NearMiss` ratio is far higher (Table 17), which points to systematic coordinate displacement rather than spatial jitter around the correct controls. Reinterpreting all predictions as normalized by 1000 raises Step Acc. from 11.5% to 36.0%, Subtask SR from 1.1% to 15.1%, Task SR from 0.0% to 4.7%, and ER from 10.4% to 21.3% (Table 18), so many coordinates are indeed 1000-normalized, yet positioning errors still account for 53.1% of steps. Treating the candidate conventions as diagnostic hypotheses, we find Claude's outputs most consistent with a mixture of resolution-normalized, 1000-normalized, and raw pixel coordinates across replies, and in some actions the two axes follow different conventions. A targeted reprojection using the effective image dimensions induced by Claude's preprocessing (inputs captured at 1280×720 keep that size; those captured at 1920×1080 or 2560×1440 are processed at 1456×819), corrected per prediction and per axis according to the observed convention, raises Step Acc. to 57.8%, Subtask SR to 24.6%, Task SR to 8.1%, and ER to 37.5%. These are the best values recoverable from Claude's logged outputs and estimate its underlying capability; because the correction is post hoc and prediction-

Table 17: Outcome distribution over evaluated steps (%) for every method and ablation variant: correct steps (`Success`), inapplicable action types (`Action`), positioning errors split into `NearMiss` and `FarMiss`, wrong parameters (`Parameter`), and unparseable replies (`Invalid`). Percentages may not sum exactly to 100 due to rounding.

Method	Success	Action	NearMiss	FarMiss	Parameter	Invalid
UI-TARS-1.5-7B	6.3	9.9	0.9	17.1	0.3	65.5
Claude Sonnet 4.6	11.5	10.2	2.8	74.7	0.2	0.7
MiniMax M3	24.4	16.5	6.9	49.3	1.2	1.7
Qwen3-VL-235B-A22B	44.8	12.6	4.6	34.2	2.2	1.6
Kimi K2.6	67.8	8.3	2.0	14.7	1.7	5.5
GPT-5.4	67.2	5.7	4.1	21.0	2.0	0.0
Agent S3	52.6	8.4	6.5	30.6	1.2	0.7
UiPath	65.2	7.3	2.7	22.5	1.8	0.5
OS-Symphony	68.3	9.2	2.3	18.0	2.2	0.0
GPT-5.4 + Video Desc.	70.1	4.6	4.7	19.4	1.1	0.1
MOVI-Agent	80.0	5.0	2.9	10.6	1.5	0.0
Ablations of MOVI-Agent						
w/o Grounder	72.1	5.4	4.2	16.4	1.8	0.0
w/o Video Guidance	74.0	6.4	2.7	15.3	1.6	0.0
w/o Paired Video	76.1	6.0	3.0	12.7	2.2	0.0
w/ Flat Description	77.7	5.0	2.7	12.6	2.0	0.0
w/o Reflector	78.2	4.7	3.3	12.5	1.3	0.0

Table 18: Claude Sonnet 4.6 under the prescribed $[0, 1]$ coordinate convention, under uniform reinterpretation of all coordinates as normalized by 1000, and under input-aware diagnostic reprojection. The latter two are diagnostic only.

Setting	Step Acc.	Subtask SR	Task SR	ER
Prescribed $[0, 1]$	11.5	1.1	0.0	10.4
Normalize by 1000	36.0	15.1	4.7	21.3
Input-aware reprojection	57.8	24.6	8.1	37.5

specific, the uniformly scored result stays in the main comparison. Even the reprojected estimate remains below OS-Symphony on every metric, so no ranking in Section 5 depends on how Claude's coordinates are read.

Invalid action outputs in UI-TARS-1.5-7B. UI-TARS fails earlier in the pipeline. Of its 1,075 generated steps, 710 (66.0%) never reach scoring: 350 are empty responses, 147 use UI-TARS's native `click(start_box=...)` syntax rather than the prescribed interface, and the rest are truncated generations, unsupported operations such as `terminate()`, or degenerate loops. This generated-step rate has a different denominator from the 65.5% `Invalid` share computed over evaluated steps in Table 17. On the 365 parseable outputs, its step accuracy is 18.4%. The low aggregate score therefore reflects non-response and format noncompliance rather than grounding alone.

Reporting under a uniform output contract. We retain the original scores because every agent is evaluated under the same explicitly specified interface. Normalized $[0, 1]$ coordinates give a resolution-independent representation regardless of how a provider resizes screenshots; introducing model-specific parsers would weaken comparability and still could not resolve outputs that switch conventions between replies or between the two axes of one action. The two outliers expose distinct failures, inconsistent coordinate representation and failure to produce a valid action, which a binary task score would conflate; separating them is what ProbeGUI's diagnostic evaluation is for.

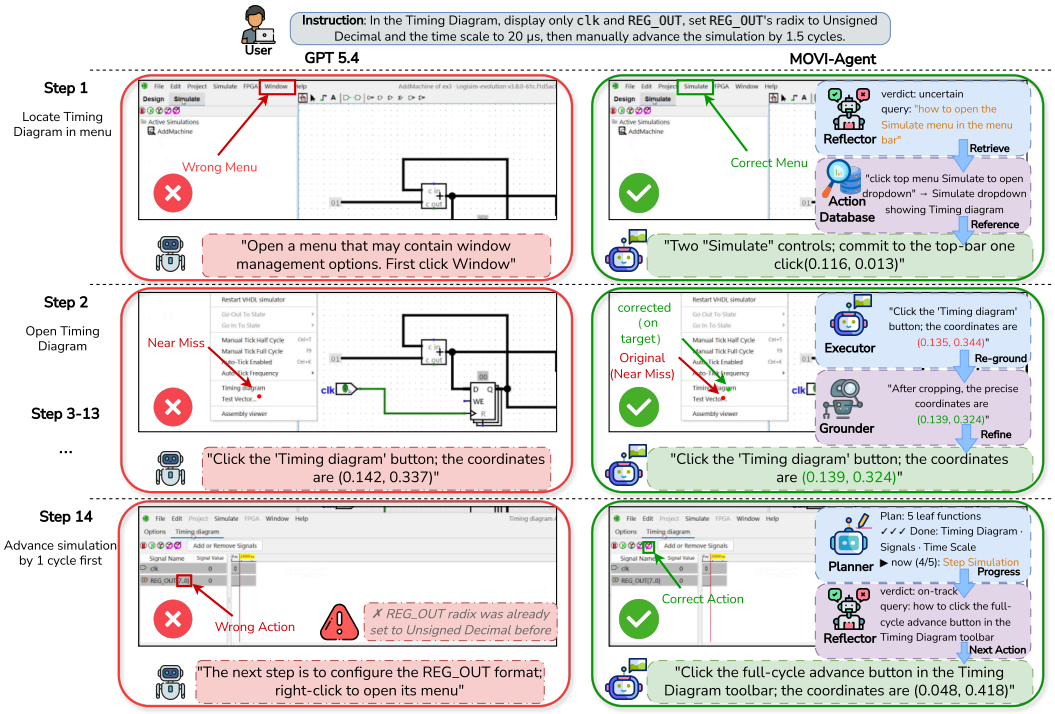


Figure 9: Logisim-6: GPT-5.4 (left) against MOVI-Agent (right) at three decisive steps. At step 1 the baseline mistakes the injected two-*Simulate* ambiguity for a window-management problem, while retrieval hands MOVI-Agent the demonstrated top-bar menu operation. At step 2 both produce a coarse estimate outside the target row, and only MOVI-Agent's Grounder crops and corrects it onto the entry. At step 14 the baseline re-issues a configuration step it already performed, while the Planner's tracked position in the function plan keeps MOVI-Agent advancing on the fourth stage.

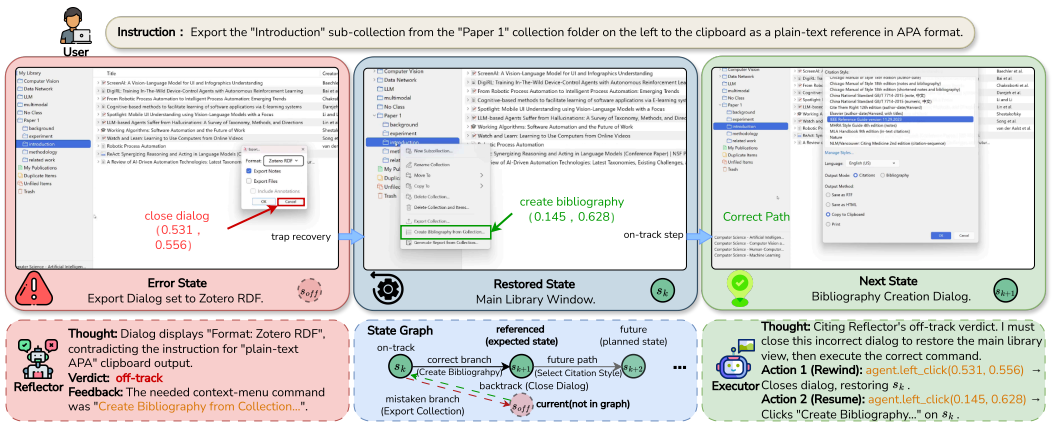


Figure 10: Semantic state backtracking on an injected error state (Zotero-3). Left: the agent is placed in an Export dialog set to `Zotero RDF`, a state s_{off} with no counterpart in the graph; the Reflector returns *off-track* and names the command that should have been taken. Center: backtracking rewinds the reference state to s_k, the last node confirmed on-track, and the correct branch out of it leads toward the planned path. Right: the Executor closes the dialog to restore s_k on screen, then takes that branch, and the run continues on path.

E CASE STUDIES

E.1 MULTI-ROLE EXECUTION ON A COMPLEX TASK (LOGISIM-6)

Figure 9 traces the hard engineering task Logisim-6 through three decisive steps, with the pure-vision GPT-5.4 baseline on the left and MOVI-Agent on the right. The instruction chains five stages: open the Timing Diagram window, keep only `clk` and `REG_OUT` in the waveform, switch `REG_OUT` from Binary to Unsigned Decimal, set the time scale to 20 μs, and manually advance the simulation by one and a half cycles. An error state is injected at the entry, where the screen carries two controls labeled *Simulate*: a mode tab under the toolbar and the actual menu in the top bar; only the top-bar menu reaches the Timing Diagram. The baseline ends the run at 68.75% step accuracy with one of the four annotated subtasks complete, MOVI-Agent at 87.5% with two.

- **Step 1: retrieval resolves the injected ambiguity.** The baseline reads the goal as a window-management problem, reasoning that it should "open a menu that may contain window management options," and clicks *Window*. The deviated screen and the expected one render almost identically, separated only by a few peripheral details on the left, the state aliasing that dense professional interfaces produce. MOVI-Agent's Reflector cannot confirm the screen against the reference path and returns *uncertain*, which by design leaves execution in place, yet it emits the intent query "how to open the Simulate menu in the menu bar". Retrieval answers with the demonstrated operation, "click top menu Simulate to open dropdown," together with the dropdown state that operation produces; reading that reference, the Executor names both *Simulate* controls and commits to the top-bar one at $(0.116, 0.013)$. An uncertain verdict suffices here because the retrieved operation supplies the disambiguation the verdict withholds.
- **Step 2: local re-grounding recovers a near miss.** Both agents now want the same target, the *Timing diagram* entry in a densely packed dropdown. The baseline commits $(0.142, 0.337)$, slightly overshooting the lower boundary of the entry, and the step is scored a positioning error. MOVI-Agent's Executor produces an equally coarse $(0.135, 0.344)$; the Grounder then crops the region around that estimate, re-reads it at higher effective resolution, and returns $(0.139, 0.324)$, inside the row. The same correction recurs when the agent selects the 20 μs time scale, where $(0.180, 0.556)$ is refined to $(0.183, 0.534)$.
- **Step 14: the function plan keeps the long horizon straight.** Thirteen steps in, the baseline's rolling context no longer carries the radix change it made earlier, so it re-derives its goal from the screen alone, concludes that "the next step is to configure the REG_OUT format," and right-clicks the signal a second time, an action error spent on finished work. MOVI-Agent instead holds the Planner's decomposition as an explicit position among five leaf functions of $\mathcal{F}$, with Timing Diagram, Signals, and Time Scale marked done and *Step Simulation* active as the fourth. The Reflector confirms the screen as *on-track* at that position, queries the toolbar's full-cycle control, and the Executor clicks it at $(0.048, 0.418)$.

MOVI-Agent does not complete Logisim-6: it commits an action error of its own during signal configuration and misses a confirmation button late in the run, so the benchmark scores the task a failure. The trajectory shows the distance covered: retrieval over the software model, local re-grounding, and an explicit position in the function plan carry the agent through the injected error state and most of the workflow, on a task where the baseline loses the entry menu at its first move. This is the per-task shape of the aggregate result in Section 5, where MOVI-Agent's failed trajectories end far closer to completion than the strongest baseline's.

E.2 SEMANTIC STATE BACKTRACKING FROM AN INJECTED ERROR STATE (ZOTERO-3)

The previous case turns on knowledge the agent had before acting; this one traces a single recovery episode from an injected trap. The task asks for the introduction subcollection of "Paper 1" to be exported as a plain-text APA bibliography copied to the clipboard, and the Planner resolves it into the leaf function *Create Bibliography from Collection*. At the trap checkpoint the agent is placed in a state it did not choose: an Export dialog is already open, set to `Format: Zotero RDF`. Nothing on screen announces an error; the dialog is ordinary, and it simply produces a file in a serialization format instead of formatted text on the clipboard.

The Reflector returns *off-track* and writes the account that accompanies such a verdict: the open window is the Export dialog set to Zotero RDF, this does not match a plain-text APA bibliography copied to the clipboard, and "the needed context-menu command was 'Create Bibliography from Collection...' ". The first half detects the deviation by comparing the live screen against the expected state on the reference path; the second half names the command that should have been chosen, which the agent reads off the software model rather than inferring from the dialog in front of it. Backtracking then rewinds to s_k, the last state confirmed on-track, which the graph holds as the plain library window with no dialog open. As Figure 10 shows, the deviated state s_{off} has no counterpart in the graph, so the rewind is a return to a modeled state rather than an undo of a recorded transition. Citing the Reflector's account in its own reasoning, the Executor closes the Export dialog at $(0.531, 0.556)$, which restores exactly that screen, then takes the correct branch out of s_k at $(0.145, 0.628)$, and the run stays on path through the remaining controls: *APA Style 7th edition* at $(0.391, 0.236)$, the *Bibliography* output mode at $(0.353, 0.548)$, and confirmation at $(0.480, 0.781)$. The deviation costs one step and leaves no residue in the states that follow, whereas an agent without a reference position faces a harder problem here: the screen after dismissing the dialog looks identical to the screen before the trap was injected.

F PROMPTS

This section reproduces every system prompt used in this work, grouped by the stage that issues it. Figures 11 to 14 show the prompts of the four execution roles: the Planner selects leaf functions from the function tree; the Reflector audits the current screen against the reference path and emits the verdict, matched index, and intent query; the Executor emits the next atomic action with a target-size estimate; and the Grounder refines the coordinate inside a magnified crop. Figure 15 gives the prompt for baselines that are bare VLMs rather than agent frameworks; it fixes the same action space, output format, and coordinate convention. Figures 16 to 21 cover the offline pipeline that builds the software model: three prompts turn video into operational semantic units, and three assemble those units into the function tree and its alignment with the state graph.

G EXTENDED DISCUSSION OF LIMITATIONS

This section expands the limitations of Section 6 and pairs each with the evidence that bounds it.

Offline, annotation-driven scoring. ProbeGUI scores trajectories against annotated actions under continue-on-miss, so a valid route the annotators did not foresee counts as an error, and an incorrect action does not produce its real consequence in the simulated trajectory. Annotating alternative valid actions (29% of states admit more than one) reduces, but cannot eliminate, the risk of penalizing unannotated routes. Continue-on-miss suppresses the downstream consequences of an incorrect action, so these two effects can pull absolute scores in opposite directions; the scores characterize execution on annotated trajectories rather than unrestricted live success. The same continuation and matching rules apply to every method. The 60 live replays found no disagreement between offline verdicts and observed outcomes on the selected trajectories, validating the deterministic scoring rule there rather than establishing closed-loop success or an invariant method ranking. No existing live harness covers the ten ProbeGUI applications with verifiable end states; building one is a direct extension.

Fixed versions and one resolution per application. Each application is pinned to one version, and within an application all videos and screenshots share one recording resolution (Appendix C.3). Transfer across software versions and across window geometries is therefore untested. The design partially anticipates these shifts: stored coordinates are never executed verbatim, and anchoring and verification use debiased descriptions rather than pixels. The provenance-aware reconstruction for the paired-video ablation records contributing videos, but transfer to new application versions and replacement of stale knowledge have not been evaluated. What the design does not yet provide is a mechanism that detects transitions rendered stale by an interface change and retires them; semantic backtracking corrects the agent's runtime belief but does not update the graph.

System Prompt: Planner Agent PROMPT

```
You are a function selector for a GUI benchmark agent. Your task is to identify which
EXISTING functions from the software's knowledge base are needed to satisfy the user's
request.

You are given:
1. The existing function tree of the software (leaf nodes are the only valid function
names): %s
2. A natural-language user request.
3. A possibly correct solution from the Internet.

### CRITICAL RULES
1. Select, do NOT create — You MUST only use function names that appear as leaf nodes in
the given tree. Do NOT invent new function names or modify spelling. If no exact match
exists, choose the CLOSEST leaf node semantically.
2. Strict Structure Names — The function names you output MUST be exactly identical to
what was provided. A name not in the tree causes a lookup failure and the agent gets no
KB hint.
3. The "Milestone" Rule — Each function must represent a significant, verifiable progress
in the user's requirement.
4. Minimal set — Output the minimal sequence of existing functions needed. ONLY include
functions explicitly mentioned or mechanically indispensable for the requested change.
Do not treat it as a complete workflow: If the user doesn't say "Load Data", assume data
is already loaded.
5. Similar References — If you cannot find a suitable function, Select a similar function
that operates in a comparable manner.

Output format:
[
    {
      "function_name": "root-parent-leaf function name exactly as in tree",
      "evidence": "brief explanation of why this function is needed"
    },
    ...
]
```

Figure 11: Planner prompt. The function tree is injected into the prompt, and the Planner selects from its existing leaves instead of naming functions freely. The phrase "a possibly correct solution from the Internet" is retained from the prompt template; no web search is performed and no Internet solution or flat video description is supplied to the Planner in reported MOVI-Agent runs.

Video extraction and demonstration scope. The intrinsic action-level audit in Appendix A.6 yields a type-agnostic temporal F1 of 63.5% and a type-consistent F1 of 55.6%, showing that extraction remains imperfect; it does not separately measure the accuracy of every pipeline stage. The current pipeline is evaluated on concise, action-dense demonstrations recorded in-house (Appendix C.3); robustness to noisier in-the-wild tutorials with weaker narration and looser editing remains untested.

Single pass. All results come from one run per task, following common practice. The paired bootstrap with Benjamini–Hochberg correction (Appendix D.5) quantifies task-sampling uncertainty; it weighs most on error recovery, where one of 48 cases moves the score by 2.1 points, and comparisons should be read at that granularity. Run-to-run stochasticity at decoding temperature 1 remains unmeasured.

System Prompt: Reflector Agent PROMPT

```
You are a Reflector Agent for a GUI operation. You will receive the screenshots (red cross
marker on screenshot indicates operating positions) and context about the task.

Your job :
(High priority) indicate the current state and function indexes
(Middle priority) conservative oversight: stay silent unless you have concrete evidence of a
problem. False alarms mislead the worker more than missed detections — when in doubt, output
 "uncertain" .
(Low priority) generate a "how to" query

### Process (follow in order)
Step 1 — Observe: Describe what did the agent do in last action. No interpretation, no
inference.
Step 2 — Judge: If a reference path to the KB exists, the best-matching index is retrieved from
it; subsequently, it is determined whether the agent has completed its current function.
Step 3 — Verdict: Determined strictly by your judgement with the trap whitelist below.
uncertain is the default. Being uncertain is correct behavior, not failure.

### Verdict rules (strict)
This is a trap candidate if the you have observed one of following issues:
  (a) The current screenshot shows an unexpected dialog, wrong menu, or error state that is
inconsistent with the instruction
  (b) A directly visible action violates an explicit hard constraint mentioned in the
instruction (Except order problems, you don't need to care. Focus on current Task)

Then, Only output trap for INTOLERABLE errors: errors that make the task impossible to
continue and must close/undo or directly contradict the instruction. Past errors are irrelevant
unless their consequences are still visible and blocking progress now. Prefer silence over
interruption.

### Field rules
- feedback: REQUIRED if verdict=trap, MUST have sufficent justification. Empty string if
verdict=on_track or uncertain.
- kb_index: best-match index from the KB reference path. If unsure, reuse the previous index.
If these states have already been surpassed, output the maximum index. The KB path may be
imprecise — treat it as a hint, not ground truth. If you believe the current function is
complete—or that you need to reference the next function—output the last kb_index.
- function_index: Currently best-match index from the Planned Functions. If unsure, set -1.
- query: a concise "how to [specific UI action]" question for the next step. Must focus on low-
level action, not the whole instruction.

### Output (XML only, no other text)
<observation>description</observation>
<verdict>on_track | trap | uncertain</verdict>
<feedback>cite specific UI element if trap, else empty</feedback>
<kb_index>N</kb_index>
<function_index>M</function_index>
<query>how to [specific action]</query>
```

Figure 12: Reflector prompt. The explicit license to answer *uncertain* instead of guessing makes the Reflector conservative; *trap* in the prompt corresponds to the *off-track* verdict of Section 3.3.

Benchmark and method developed together. ProbeGUI and MOVI-Agent were developed within the same research effort. Tasks were curated from official documentation and community forums rather than selected around behaviors MOVI-Agent was known to handle, and every method is evaluated under the same observation protocol, action space, harness, and scoring rules, which removes method-specific evaluation advantages.

System Prompt: Executor Agent PROMPT

```
You are a GUI agent operating %s. You receive a demonstration history (screenshots + action captions from a
correct demo, red cross marker on screenshot indicates operating positions) followed by the CURRENT
screenshot. Your single job is to emit ONE action call that best advances the user's task.

You may call exactly ONE of the following actions. Coordinates are screen ratios in [0.0, 1.0]; positional
grading is bbox-based, not pixel-perfect, so aim for the visual centre of the intended element. %s

Hard constraints:
- Output ONE action only. NEVER chain multiple agent.* calls.
- Do not invent action names. If none of the listed actions fit, pick the closest one.
- Use floats in [0,1] for coordinates. NO fractions, NO integers, NO arithmetic. At least three decimal
places. Provide the coordinates of the object's center based on the grid.
- For type, wrap text in triple quotes """...""". Pay attention to overwrite and enter options
- For hotkey, separate keys as positional args (e.g. agent.hotkey("ctrl","s")). Keep them lowercase.

Context included above your turn:
- Instruction — the user's high-level goal.
- Planned Functions — soft scaffold of function-level sub-goals; → marks the current one. Trust the screenshot
when they disagree.
- Previous Milestones — semantic summary of what has been accomplished so far.
- Tip — human hint for this frame. You MUST follow the tip in your prediction.
- Reflector Feedback — Analysis provided by the Reflector agent. If it flags a potential pitfall, you may
choose to perform a remedial action (delete, undo, or close)—or, provided there are no side effects, proceed
directly with the correct action. Always treat this as a HINT, not a command.
- KB Reference Path — scene descriptions from past recordings near the estimated current position. REFERENCE
ONLY — your actual path may differ legitimately. Never let this override what you see on screen.
- Summary — You will receive Progress summary of the previous history. You must continuously maintain this
within the <summary> field.

HINT:
- The demo history screenshots (images before the last one) show ONE valid trajectory. Your action does NOT
need to match it literally — different UI paths to the same outcome are fine. A wrong step is scored 0 with no
recovery, so be conservative.
- Prefer to use the search bar(agent.type) in menu unless option is visible; this helps avoid errors. But this
requires you highly convinced the search content.Otherwise, find with click and scroll.
- When encounter trap, Prioritize direct Redirection if the correct target is visible(like in same window).
Only use "X/Cancel" if the current state occludes the target or makes it unreachable. Shortcuts like
esc/ctrl+z are last choices.
- The <size> field estimates how much of the screen the longest side of TARGET ELEMENT occupies, Estimate
quickly on it:
- - big: element covers more than 0.25 of the screen (large dialog, full panel, ribbon bar)
- - medium: element covers 0.1-0.25 the screen (toolbar button group, menu column, sidebar, searchbar)
- - small: element covers less than 0.1 of the screen (single button, checkbox, icon, menu item)

# Response Format
You MUST output exactly this XML, nothing else:
<response>
    <summary>
      Update the Progress summary to provide an overview of everything we have accomplished so far exclude
current prediction, ensuring that nothing is omitted.
    </summary>
    <thought>
      Analyse the current screenshot (last image). Note the sub-goal, tip, and reflector feedback. Justify the
single action you will emit. If the reflector flagged a trap, explain your recovery plan. Also, describe the
object you want to interact with.
    </thought>
    <action> agent.left_click(0.512, 0.334, numclicks=1) </action>
    <size>small | medium | big</size>
</response>
```

Figure 13: Executor prompt. Alongside the next atomic action, it returns an estimate of the target's on-screen size, which sets the crop window the Grounder works in. The "Tip" it refers to is the `tips` annotation field that the harness serves to every method.

Scope of backtracking. Semantic state backtracking repairs execution-level deviations; a flawed decomposition by the Planner lies beyond its reach by design. The hierarchical metrics keep this boundary visible: failures in function selection can surface at the subtask level. Separately, the

System Prompt: Grounder Agent PROMPT

```
You are a GUI Coordinate Grounder. You receive a zoomed-in, square crop of a GUI
screenshot with a coordinate grid overlay. A red cross marker marks where the
worker agent predicted the interaction target to be.

Your ONLY job: output the coordinates of the EXACT CENTER of the target element
described in the user message.

## Rules
1. Coordinates are ratios in [0.0, 1.0] relative to THIS CROPPED IMAGE — not the
full screen.
2. Output the geometric center of the element — unless the target is a line/curve.
3. If the target element is NOT clearly visible in the crop, output the provided
raw coordinate to preserve the worker's approximate prediction.
4. If the CENTER of red cross is already on the target obviously, do not fine-tune;
simply output the raw coordinates directly.

Output ONLY strict JSON, nothing else:
{"x": 0.XXX, "y": 0.XXX}
```

Figure 14: Grounder prompt. It sees only the magnified crop and re-predicts the coordinate inside it, with no access to the software model.

semantic coverage analysis finds that runtime Function Plans cover 76.1% (201/264) of all evaluation subtasks and 89.7% (201/224) of those supported by demonstrations (Appendix A.6); these are coverage measures, not failure-rate bounds.

System Prompt: Baseline Agent Prompt PROMPT

```
You are a GUI automation expert. Your task is to analyze the task instruction, any
available action history, and the current screenshot, then choose exactly one immediate
next atomic action.

You have access to the following action methods:
{methods}

The available actions are only the API surface. You must obey task-specific constraints
first; do not use an available action if the task forbids it.

Use any previous screenshots and completed actions as context for understanding the
workflow and how the interface reached its current state. They are not the source of
coordinates or parameters for the next action. The current screenshot is authoritative
for the next action, including the action type, coordinates, and parameters.

Coordinate rules:
- Any action with coordinates must use normalized coordinates in [0, 1] relative to the
current screenshot size.
- The origin is the top-left corner of the screenshot.
- x increases to the right; y increases downward.
- For coordinate-based actions, choose coordinates according to the intended effect on
the current screenshot. If the action targets a visible UI element, use a stable point
near its center. If the action is region-based rather than element-targeted, use the
appropriate location for that operation.

Action rules:
- Complete only the immediate next atomic action.
- Do not combine multiple steps into one action.
- Do not invent actions. The action must be one of the available agent methods exactly.

You MUST output your response in XML format:
<response>
    <thought>Your reasoning about what action to take and why</thought>
    <action>agent.method_name(param1, param2, ...)</action>
</response>

Output rules:
- Output ONLY the XML response, nothing else.
- The <action> content must be a single agent.method(...) call.
- Do not put markdown fences in <action>.
- Do not put natural language in <action>.
- Do not output an empty <action>.
```

Figure 15: Baseline prompt for VLMs evaluated without an agent framework. It states the same action space, output format, and normalized coordinate convention required of every method.

System Prompt: Video Abstract Prompt PROMPT

```
You are a GUI agent. You need to predict the action between user's two screenshots with grids
to help you find coordinate.

Your response must be a strictly formatted JSON object with trailing commas on each line and
no comments.
This object needs to contain the following fields to describe the action you are predicting:
'action': the action type in action space given below.
'coordinate':(need to be ratio of the whole screen) [x1,y1](if action is scroll, y1
represents the scrolling ratio of the screen) or [x1,y1,x2,y2](if action is drag, x1,y1 is
start point)
'keyboard_input': like ctrl+c when action is hotkey or the real content when action is type
'semantic_action': describe the semantics of the action
'previous_scene': summarize the functionality of the page before the action happens
'current_scene': summarize the functionality of the page after the action happens
'intention': predict user's intention in this action

## Action Space
click: need a pair of coordinate
left_double: need a pair of coordinate
right_single: need a pair of coordinate
drag: need two pairs of coordinate
hotkey: need a keyboard_input that use '+' to join lowcase keys. Do not use more than 3 keys
in one hotkey action.
type: need a pair of coordinate and the real content as keyboard_input
scroll_up: need a pair of coordinate that shows how origin point moves, like(0,0.5)
scroll_down: need a pair of coordinate that shows how origin point moves, like(0,0.5)
wait: no action is truely taken.

Output STRICT JSON only — no prose, no commentary, no markdown fences:
here is a example:
{
  "action": "click",
  "coordinate": [0.35, 0.42],
  "keyboard_input": null,
  "semantic_action": "click on the filter button to open the filter menu",
  "previous_scene": "image editing workspace with toolbar",
  "current_scene": "filter selection panel displayed",
  "intention": "to apply image filter"
}
```

Figure 16: Video abstraction prompt. It formalizes the enhanced keyframes as (a, s_a, i) within the operational semantic unit m; d is generated separately.

System Prompt: Debiased Description Prompt PROMPT

```
You are a meticulous and neutral software analyst.
Given a software screenshot, you MUST output TWO fields:

1. "signature": a SINGLE-LINE canonical state identifier (<= 30 words). It is used to decide
whether two screens are the same UI state, so it must be COMPACT yet DISCRIMINATIVE. It MUST
follow the order:
   - app + page/view in focus
   - any open menu / dialog / modal / overlay (use markers like menu:File, dialog:SaveAs,
panel:Layers, popup:none)
   - active selection or input focus (e.g. focus:caret in cell B3, focus:none, focus:button
OK)
   Avoid mentioning personalized content (file names, user text, photo content). Two screens
that differ ONLY by such content must produce the SAME signature.

2. "evidence": a longer human-readable description of what the user can do on this screen
(visible buttons, ribbons, panels, input boxes, with their apparent role). Same rules as the
signature about ignoring personalized content. Several short bullet-style sentences are fine.

Rules common to both fields:
- Describe only what is visibly enabled; do not speculate about hidden or future actions.
- Stay objective, functional, English.

Output STRICT JSON only — no prose, no commentary, no markdown fences:
{
  "signature": "Microsoft Word document edit view; menu:none; dialog:none; focus:caret in body
paragraph.",
  "evidence": "This interface is a Microsoft Word document editing screen. It allows users to:
navigate and search via the left Navigation pane (search input, tabs for
Headings/Pages/Results); format text via the Home ribbon (font styling, alignment, paragraph
styles); access additional capabilities via
Insert/Design/Layout/References/Mailings/Review/View tabs; edit and compose text directly in
the central document area."
}

Now describe the provided screenshot.
```

Figure 17: Prompt that generates the debiased description d. It strips personalized inputs and transient content from a screenshot, leaving the topological structure that forms a state-graph node.

System Prompt: Agent Merger Prompt PROMPT

```
You are given a partial sequential list of actions extracted from a tutorial video.
Your job is to clean the sequence by removing actions that have no meaningful effect, such as
accidental clicks, unintended cursor movements, redundant window switches, or state changes
that are immediately undone.

Your rules:
An action is invalid if it does not contribute to any meaningful progress or does not change
the functional state of the application.

Examples of invalid actions:
- Opening a panel and immediately closing it with no change.
- Navigating to a page/window and immediately navigating back.
- Actions that are clearly repeated several times and whose effects do not accumulate should
only be retained once (keep last one).
- Clicking empty space or UI that causes no effect.
- Accidental toggles that end up in the same state.
- A meaningless wait action is not waiting for something to respond.
- Scenes unrelated to video, such as advertisements and animations

Preserve only actions that:
- change the application's state,
- contribute toward completing a task,
- or reveal a user intention that leads to progress.

Your output
Return:
A cleaned list indicates the meaningless actions index splited with comma.(index is provided
by user)
No explanation, no commentary.
Maintain original ordering.

Input example:
{ {action:0, action 0's description},...{action: m-1, action m-1's description}}

Output format:
3,7,8,...m-3
```

Figure 18: Action merger prompt. It cleans the action trajectory by removing semantically empty, accidental, or immediately undone user actions.

System Prompt: Function Searcher Prompt PROMPT

```
You are an expert in software operation modeling.
Your task is to search the web and build an initial high-quality function library for a
software.

Your goals:
Use online information (official documentation, tutorials, feature lists, user guides, FAQ,
and credible tech blogs) to infer the software's major functional categories.
For each functional category, extract its typical user operations and the underlying
semantics (what the function accomplishes).
Output only general software-level functions in English, not UI layout or step-by-step
instructions. However, functions can't be too generalized.

Rules:
- Focus on capabilities, not UI positions or interface appearance.
- If online information is scarce or conflicting, infer the most likely functions based on
common patterns in similar software.
- Do not hallucinate detailed UI sequences; keep everything high-level and feature-oriented.
- One function serves only one specific purpose; Do not include 'and'.
- Avoid describing tasks unrelated to software functionality (such as installation steps).

Output Format Example
{
  "category": "WeChat",
  "children": [
    {"category": "Chats", "children": [{"category": "View Chat History"}, {"category": "Video
Call"}, {"category": "Send Files"}, ...]},
    {"category": "Contacts", "children": [{"category": "Search New Friends"}, {"category":
"View Contact List"}, ...]},
    {"category": "Discover", "children": [{"category": "Moments", "children": [{"category":
"View Friend Updates"}, ...]}, ...]},
    {"category": "Search"},
    {"category": "Me", "children": [{"category": "Account Info", "children": [...]},
{"category": "Pay", "children": [...]}, ...]},
    {"category": "Add", "children": [{"category": "Start Group Chat"}, ...]}br  ]
}

Your task:
Search the web and produce such a structured initial function library for Software:
```

Figure 19: Function search prompt. It consults the official documentation and web resources offline to bootstrap the category hierarchy of the function tree.

System Prompt: Function Extractor Prompt PROMPT

```
You are an expert software interaction analyst.
Your task is to determine whether the current frame/action in a video completes any known
software function.

You will receive:
A list of known functions.
A series of action/frames description extracted from video.

Your job:
Find the action at which a certain existing function was completed.
Do not ouput unrelated functions if the series demonstrates a function not mentioned.
If no function is completed, output empty JSON object {}.
The name of function must be completely same with provided ones.
You MUST output using the exact JSON schema provided below. No free text explanation is
allowed.
MUST use single quotes ' inside a string.

{
  "8": [
    {
      "function_name": "function 1",
      "evidence": "one-sentence natural-language justification"
    },
    {
      "function_name": "function 2 (maybe complete concurrently)",
      "evidence": "one-sentence natural-language justification"
    }&br  ],
  "15": ...(if frame 15 also complete some functions)
}

Known Functions:
%s
```

Figure 20: Function extraction prompt. It maps the intents of operational semantic units to completed leaf nodes of the existing function tree.

System Prompt: Function Detector Prompt PROMPT

```
You are a Function Discovery Expert.
Your job: Analyze user interface operations from action logs and distill NEW high-level
software functions.

CRITICAL DEFINITION: What is a "Function"?
A "Function" must represent a COMPLETE USER INTENT (e.g., "Formatting a Paragraph").
- DO NOT extract "Atomic Actions" or "UI Navigation" (e.g., "Clicking OK", "Opening a Menu",
"Hovering", "Closing a Dialog").
- A function must result in a meaningful change in the software's state, data, or
presentation.
- The granularity of your new function MUST match the leaf nodes in the provided reference
tree.

CONSTRAINTS:
1. Compare the current action against the Known Functions tree.
2. If the action is a sub-step of an existing function, IGNORE IT.
3. If the action completes a legitimate new capability not in the tree, distill it.
4. Output the complete path from root to the new leaf node, separated by '-'. If you create
new parent categories, ensure they are logically consistent.
5. If user's given completed functions can represents user's actions and no new function is
needed, output empty JSON object {}. MUST use single quotes ' inside a string.

Output format Sample:
{
  "8": [
    {
      "function_name": "root-ancient1-...-ancient n-new function 1(some ancient nodes may be
created by you too)",
      "evidence": "brief natural-language justification, why we need it?"
    },
    {
      "function_name": "root-...-new function 2 (maybe complete concurrently)",
      "evidence": "brief natural-language justification, why we need it?"
    }
  ],
  "15": ...(if frame 15 also complete some new functions)
}
```

Figure 21: Function detection prompt. It analyzes operation logs to discover unrecorded software capabilities and creates new nodes to extend the function tree.