LocalLSTC: A Long Short-Term Control Architecture for Locally Deployed GUI Agents

Anonymous authors
Paper under double-blind review

Abstract

Modern GUI-agent frameworks achieve strong desktop task performance with frontier API models, yet persistent control information often remains implicit in growing interaction trajectories. At each step, the planner reconstructs the active task stage, accumulated evidence, and runtime feedback before deciding the next action. This dependence becomes more pronounced under weaker local reasoning backbones. Across four representative state-of-the-art frameworks, replacing GPT-5 with Qwen3.5-9B reduces average OSWorld SR-100 from 60.9% to 35.2%. Trajectory annotation further identifies at least one control failure in 94.7% of failed trajectories. To address this problem, we introduce LocalLSTC, a training-free architecture that organizes control by temporal scope, maintaining persistent cross-step state to guide short-term execution commitments. The framework comprises two complementary mechanisms. Long-to-Short Planning forms each commitment from persistent state, while Short-to-Long Control integrates execution outcomes back into that state for progress assessment, recovery, and termination. With Qwen3.8-27B, LocalLSTC reaches 73.7% SR-100 on OSWorld and 68.5% on WindowsAgentArena, achieving the strongest local OSWorld result and a new state of the art on WindowsAgentArena. Ablations further support contributions from both mechanisms. Together, these results show that temporal organization of control information can reduce cross-step control failures and complement backbone scaling in locally deployed GUI agents.

1 Introduction

GUI agents extend multimodal models from interface understanding to sequential action in computer-use tasks. Long-horizon tasks require agents to preserve the active task stage across interactions, interpret execution outcomes, recover from failures, and terminate only after task constraints are satisfied. Recent systems improve desktop task performance through hierarchical planning, specialized components, code execution, and test-time search (Agashe et al., 2025a;b; Gonzalez-Pumariega et al., 2026; Yang et al., 2026b; Song et al., 2026). Yet persistent control information often remains implicit in the growing interaction trajectory. At each decision, the planner needs to reconstruct the active task stage, accumulated evidence, and relevant runtime feedback before determining how to proceed. We call this repeated reconstruction the *implicit trajectory-level control burden*. As illustrated in Figure 1(a), reconstruction errors can manifest as loss of stage commitment, incorrect progress assessment, repeated no-progress behavior, mismatched recovery, or premature termination. We refer to these observable manifestations as *control failures*.

Local deployment makes this control burden particularly consequential. Keeping inference local allows screenshots, files, credentials, browser state, and application data to remain within the local deployment boundary, while reducing dependence on external-service latency, availability, and recurring invocation costs (Liu et al., 2024; Xue et al., 2024). However, public benchmarks and open-model studies continue to show a substantial capability gap between locally deployed open models and strong API models on desktop tasks (Xie et al., 2024; Bonatti et al., 2025; Wang et al., 2025c). Recent reasoning models narrow this gap (Qwen Team, 2026a;b;c; Google DeepMind, 2026), but deployable model scale remains constrained by local hardware capacity, compute, and inference throughput (Liu et al., 2024; Xue et al., 2024). Existing approaches primarily strengthen model parameters or system composition through GUI-specific training, specialized components, or additional inference-time computation (Agashe et al., 2025b; Song et al., 2026; Qin et al., 2025;

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

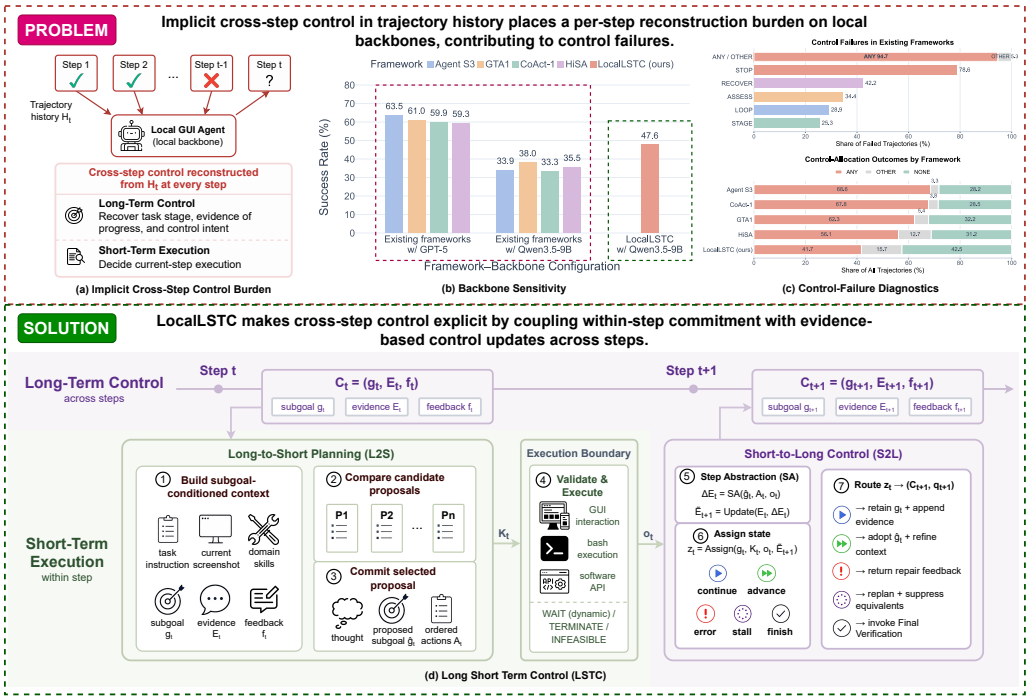


Figure 1: Overview of the control problem, empirical evidence, and LocalLSTC. Panel (a) illustrates the implicit trajectory-level control burden. Panel (b) shows the SR-100 drop after replacing GPT-5 with Qwen3.5-9B across four GUI-agent frameworks. Panel (c) reports cross-step control-failure incidence under Qwen3.5-9B. Panel (d) presents the seven-step L2S–S2L control loop.

Xu et al., 2025). Runtime interfaces for adapting off-the-shelf general backbones to long-horizon desktop control remain less systematically studied (Xu et al., 2026b).

Across four representative state-of-the-art GUI-agent frameworks, our Qwen3.5-9B evaluations average 35.2% OSWorld SR-100, 25.7 percentage points below the reported GPT-5 average of 60.9%, as shown in Figure 1(b). Averaged equally across the four frameworks, 94.7% of failed local-backbone trajectories contain at least one control failure. Figure 1(c) shows that these failures span stage maintenance, progress assessment, repeated no-progress behavior, recovery, and termination, with the dominant category varying across frameworks.

To address this problem, we introduce LocalLSTC[1] , a Long Short-Term Control Architecture for Locally Deployed GUI Agents. LocalLSTC organizes control by temporal scope, maintaining persistent cross-step state to guide bounded short-term execution commitments. The framework comprises two complementary mechanisms. Long-to-Short Planning (L2S) forms each commitment from the persistent state and current observation, while Short-to-Long Control (S2L) integrates execution outcomes back into that state for subsequent progress assessment, recovery, and termination. The resulting L2S–S2L loop, illustrated in Figure 1(d), carries the active subgoal, accumulated evidence, and runtime feedback across interactions without requiring each planner call to reconstruct them from the full trajectory.

Empirically, LocalLSTC improves both task performance and cross-step control behavior under constrained local backbones. As shown in Figure 1(b), under the same Qwen3.5-9B backbone, LocalLSTC reaches 47.6% SR-100, outperforming the four-framework baseline average of 35.2% by 12.4 percentage points. Figure 1(c) further shows that LocalLSTC reduces ANY control-failure incidence from the 63.7% baseline average to 41.7%. With Qwen3.8-27B, LocalLSTC reaches 73.7% SR-100 on OSWorld and 68.5% on WindowsAgentArena, achieving the strongest local OSWorld result and a new state of the art on WindowsAgentArena. Fixed-backbone ablations further show

[1]Code available at https://anonymous.4open.science/r/LocalLSTC-361C

contributions from both L2S and S2L. Together, these results indicate that explicitly organizing control information across temporal scopes can reduce cross-step control failures and complement backbone scaling in locally deployed GUI agents. Our contributions are as follows.

- We quantify reasoning-backbone sensitivity across four state-of-the-art GUI-agent frameworks and characterize failed local trajectories with five observable cross-step control-failure labels.
- We introduce LocalLSTC, a training-free architecture that organizes persistent cross-step state and bounded short-term execution through complementary L2S and S2L mechanisms.
- LocalLSTC achieves the strongest local results on OSWorld and establishes a new state of the art on WindowsAgentArena. Under a fixed backbone, LocalLSTC shows fewer cross-step control failures than the baselines, while ablations support contributions beyond backbone scaling alone.

2 Related Work

2.1 GUI-Specific Models and Parameter Adaptation

One route to improving locally deployable GUI agents is to strengthen the model itself through GUI-specific data and parameter adaptation. UI-TARS and Aguvis jointly train grounding, reasoning, and GUI action generation (Qin et al., 2025; Xu et al., 2025). OS-ATLAS scales GUI grounding data to improve localization and out-of-distribution generalization, while AgentCPM-GUI combines grounding-aware pre-training, trajectory supervised fine-tuning, and reinforcement fine-tuning for mobile interaction (Wu et al., 2025; Zhang et al., 2025). These approaches improve GUI competence by encoding task-relevant perception, grounding, and interaction patterns into model parameters, but their gains depend on additional training data and adaptation procedures. LocalLSTC operates with an off-the-shelf backbone and leaves model parameters unchanged, providing a training-free control alternative for improving long-horizon GUI interaction.

2.2 GUI Agent Frameworks and Inference-Time Scaling

Beyond parameter adaptation, GUI-agent frameworks improve performance by strengthening the system around the backbone through composition, execution tools, and additional inference-time computation. Agent S and Agent S2 combine hierarchical planning, grounding, and execution through an Agent-Computer Interface and generalist–specialist coordination (Agashe et al., 2025a;b). Agent S3 and GTA1 increase test-time computation through candidate generation, rollout selection, and judge-based ranking (Gonzalez-Pumariega et al., 2026; Yang et al., 2026b). CoAct-1 extends GUI interaction with programmatic execution, while Avenir-Web and ClawGUI develop grounding and system infrastructure for broader computer-use settings (Song et al., 2026; Li et al., 2026a; Tang et al., 2026). These systems demonstrate that stronger orchestration, richer execution interfaces, and additional inference-time computation can substantially improve computer-use performance. LocalLSTC is compatible with these system-level enhancements and contributes a complementary control structure for maintaining task state and execution evidence across interactions.

2.3 Sequential Planning and Runtime Control

Beyond these system-level approaches, sequential agents organize long-horizon decisions through search, subgoals, explicit execution states, and revisable plans. Language Agent Tree Search explores candidate trajectories with inference-time search, while the Subgoal-driven Framework tracks task progress through online subgoal planning (Zhou et al., 2024; Wang et al., 2026). Agent-SAMA uses a finite-state machine for execution verification and recovery, AgentProg represents interaction history as programs with variables and control flow, and Plover exposes plans as inspectable and revisable objects (Guo et al., 2026; Tian et al., 2026; Venkatesan et al., 2026). LIFE-HARNESS adapts a runtime harness around a fixed model (Xu et al., 2026b). HiSA introduces hierarchical state abstraction for GUI agents through Step Abstraction and periodic Context Refinement (Li et al., 2026b). LocalLSTC adopts these evidence-compression mechanisms while coupling them to an L2S–S2L loop that organizes persistent state and bounded execution by temporal scope.

3 METHOD

3.1 PROBLEM FORMULATION AND STATE REPRESENTATION

A common GUI-agent control interface conditions each decision on the current observation and accumulated interaction trajectory. At step t,

$$a_t = \pi(x_t, H_t), \qquad o_t = \mathrm{Exec}(a_t), \qquad H_{t+1} = H_t \oplus (a_t, o_t),$$

where x_t contains the task instruction and current observation, H_t is the interaction trajectory accumulated through step t, and a_t is the action selected by the policy π. The execution operator Exec applies a_t to the environment and returns the resulting outcome o_t. The operator $\oplus$ appends the current action–outcome pair (a_t, o_t) to the trajectory, producing H_{t+1}. This trajectory-conditioned interface leaves the active task stage, accumulated evidence, and runtime feedback implicit in H_t, requiring them to be reconstructed before each new decision.

LocalLSTC externalizes information that persists across interactions as a *Long-Term Control State*

$$C_t = (g_t, E_t, f_t), \qquad C_0 = (\bot, \varnothing, \bot).$$

where g_t represents the active subgoal and unmet constraints, E_t records subgoal-aligned execution evidence, and f_t carries runtime feedback from errors, stalls, or rejected termination. Together, these variables preserve task stage, progress evidence, and control feedback across planner calls.

The corresponding step-local object is a *Short-Term Execution Commitment*

$$K_t = (\hat{g}_t, A_t),$$

where $\hat{g}_t$ is the execution subgoal selected for the current step and A_t is the ordered action list generated under that subgoal.

L2S forms each Short-Term Execution Commitment from persistent state and the current observation, while S2L incorporates its outcome into the next Long-Term Control State. The interaction between the two temporal scopes is

$$K_t = \mathrm{L2S}(x_t, C_t), \qquad o_t = \mathrm{Exec}(A_t), \qquad (C_{t+1}, q_{t+1}) = \mathrm{S2L}(C_t, K_t, o_t),$$

where L2S maps observation x_t and persistent state C_t to commitment K_t. Executing its action list A_t produces o_t. If execution halts early, o_t records only the executed prefix and its outcomes. The unexecuted suffix provides no evidence. S2L incorporates the commitment and outcome into C_{t+1} and determines task status q_{t+1}. Here, $q_t \in \{\texttt{running}, \texttt{success}, \texttt{failure}\}$, with $q_0 = \texttt{running}$. LocalLSTC is instantiated with a planner backbone and an optional visual grounder, which supplies GUI coordinates when the planner has insufficient visual grounding capability.

3.2 L2S–S2L CONTROL LOOP

Panel (d) of Figure 1 instantiates the interaction between the two temporal scopes as a seven-step control cycle. Steps 1–3 form the Short-Term Execution Commitment K_t from the current observation and persistent state. Step 4 validates and executes the commitment to produce outcome o_t. Steps 5–7 abstract the execution outcome, determine the corresponding control transition, and update the Long-Term Control State C_{t+1} and task status q_{t+1}. The updated state then conditions the next interaction. The following subsections describe each part of the L2S–S2L control loop in detail.

3.2.1 LONG-TO-SHORT PLANNING

L2S converts persistent control state into the execution commitment for the current step. In step 1, g_t provides the active stage reference, E_t provides evidence accumulated from prior execution, and f_t provides feedback from the preceding transition. The initial call derives the first subgoal from the task instruction and current observation. Subsequent calls condition planning on the persisted state. Domain skills are selected from the task-related applications and focused by the foreground

application when multiple applications are involved, keeping operational guidance aligned with the active execution context. Appendix A.1 details planner context construction.

In step 2, the planner may generate one to three candidate proposals as sequential autoregressive text within a single completion, with all candidates conditioned on the same persistent control state. The same completion then emits a `Best` candidate index and a brief selection rationale before producing the final executable JSON. No additional candidate sampling, planner call, or separate judge is used. This differs from inference-time approaches that sample candidates across multiple planner calls and rank them with a separate judge, as in GTA1 (Yang et al., 2026b). Appendix B.3 provides the response format.

In step 3, the selected proposal is serialized as a JSON object containing `thought`, `subgoal`, and `actions`. The proposal specifies the execution subgoal $\hat{g}_t$, where the reserved value `continue` retains g_t and any other value proposes a new subgoal, together with an action list A_t. Actions can combine GUI, Bash, and software API channels as needed within the same commitment. LocalL-STC adopts programmatic Bash execution from CoAct-1 (Song et al., 2026), while its software API reuses the schema-constrained interface and execution handlers from ComputerRL (Lai et al., 2026). Appendix A.1 specifies the planner interface and action space.

3.2.2 Execution

Step 4 validates and executes $K_t = (\hat{g}_t, A_t)$. Valid actions are dispatched in list order through the corresponding executor. Reserved actions `WAIT`, `TERMINATE`, and `INFEASIBLE` are handled directly by the runtime. The resulting outcome o_t, together with K_t, are passed to S2L. Execution is supported by deterministic runtime operations for output parsing, schema validation, action fingerprinting, and timeout handling, with the corresponding runtime details provided in Appendix A.

3.2.3 Short-to-Long Control

S2L converts the outcome of the current commitment into the next persistent control state. In step 5, Step Abstraction (SA) evaluates the observed execution outcome against the intended effect of the current commitment. Given the execution subgoal $\hat{g}_t$, ordered actions A_t, and observed outcome o_t, Step Abstraction determines whether execution produces task-relevant progress, no change, or an exception, and records this assessment as a subgoal-aligned evidence increment

$$\Delta E_t = \mathrm{SA}(\hat{g}_t, A_t, o_t), \qquad \widetilde{E}_{t+1} = \mathrm{Update}(E_t, \Delta E_t).$$

where ΔE_t denotes the subgoal-aligned evidence extracted from the current execution outcome, and $\widetilde{E}_{t+1}$ is the evidence state after incorporating this increment into E_t. This process converts heterogeneous execution traces into evidence interpreted with respect to the action intent and current subgoal. The mechanism builds on HiSA's Step Abstraction for GUI execution and extends the evidence interface to Bash and software API channels (Li et al., 2026b). Appendix A.3 details the channel-specific observation and abstraction mechanisms.

In step 6, the runtime assigns a transient routing state

$$z_t = \mathrm{Assign}(g_t, K_t, o_t, \widetilde{E}_{t+1}),$$

where $z_t \in \{\texttt{continue}, \texttt{advance}, \texttt{error}, \texttt{stall}, \texttt{finish}\}$. Assignment follows a fixed precedence. A `TERMINATE` request maps to `finish`. Other execution failures map to `error`. Repeated no-progress maps to `stall` only when it remains under the current subgoal. A proposed execution subgoal different from the persistent subgoal maps to `advance`. All remaining cases map to `continue`. Appendix A.4 specifies the detection rules and thresholds.

In step 7, the routing state determines the next control state

$$(C_{t+1}, q_{t+1}) = \mathrm{Route}(z_t, C_t, K_t, \widetilde{E}_{t+1}).$$

where Route applies the transition specified by routing state z_t to the previous control state C_t, the current commitment K_t, and the updated evidence $\widetilde{E}_{t+1}$, producing the next control state C_{t+1} and task status q_{t+1}. The `continue` route retains the current subgoal state and appends new evidence, while the `advance` route adopts the proposed subgoal state. Successful `continue` and

`advance` transitions clear stale runtime feedback. An `advance` can trigger Context Refinement, which compresses accumulated evidence into compact history while retaining recent evidence for subsequent planning and verification (Appendix A.5). LocalLSTC inherits this refinement mechanism from HiSA but replaces its fixed-period schedule with execution-state triggering (Li et al., 2026b). The `error` route writes execution-specific feedback into the persistent state, while `stall` records no-progress feedback for the next planner call. The `finish` route invokes Final Verification, which evaluates task completion against accumulated visual, textual, and software API evidence (Appendix A.6). A `PASS` sets the task status to `success`. A `FAIL` records the rejected termination as a `stall`, writes recovery feedback into the persistent state, and resumes planning with status `running`. The `INFEASIBLE` action and predefined runtime failure boundaries set $q_{t+1} = \texttt{failure}$. Step Abstraction, Context Refinement, and Final Verification use the planner backbone under dedicated prompts (Appendix B), while state assignment and routing are deterministic runtime operations.

4 EXPERIMENTS

Experiments evaluate three aspects of LocalLSTC. We first examine planner-backbone sensitivity and cross-step control failures in existing GUI-agent frameworks. We then evaluate LocalLSTC across two desktop benchmarks and multiple local backbones. Finally, fixed-backbone ablations assess the effects of removing L2S, S2L, and components of the L2S–S2L control loop, with runtime-event analysis examining recovery after execution problems.

4.1 EXPERIMENTAL SETUP

Benchmarks. We evaluate LocalLSTC on OSWorld-Verified with the Google Drive tasks included (Xie et al., 2024) as the primary benchmark and WindowsAgentArena (Bonatti et al., 2025) as a secondary benchmark for cross-environment evaluation. OSWorld contains 369 Linux tasks, while WindowsAgentArena contains 154 Windows tasks. Most analyses are conducted on OSWorld, with WindowsAgentArena used to assess transfer to a distinct operating environment.

Implementation. OSWorld and WindowsAgentArena run in benchmark-defined Ubuntu and Windows virtual machines, respectively, hosted in headless Docker containers. All experiments use screenshots at a resolution of 1280×720. Local inference runs on a workstation with an Intel Core Ultra 7 265 processor and a 96 GB NVIDIA RTX Pro 6000 Blackwell Workstation Edition GPU. Unless stated otherwise, LocalLSTC uses Qwen3.5-9B (Qwen Team, 2026a) as the planner backbone and GTA1-7B (Yang et al., 2026b) for visual grounding. Additional experiments evaluate Qwen3.5-4B, Qwen3.6-27B, Qwen3.8-27B, Gemma-4-E4B-it, and Gemma-4-26B-A4B-it (Qwen Team, 2026a;b;c; Google DeepMind, 2026). Main benchmark results are averaged over two independent runs with the same parameter settings. Unless otherwise stated, other experiments use a single run. Appendix A.7 lists runtime parameters, and Appendix B provides the prompt templates.

Control-failure annotation. To characterize the cross-step control failures associated with local-backbone execution, we use GPT-5.6-sol to perform multi-label annotation of OSWorld trajectories from Agent S3, CoAct-1, GTA1, HiSA, and LocalLSTC under the same Qwen3.5-9B planner backbone. The labels STAGE, ASSESS, LOOP, RECOVER, and STOP capture unresolved failures in stage maintenance, progress assessment, repeated no-progress behavior, recovery, and termination. ANY indicates that a trajectory contains at least one of these five control failures, OTHER records a failure outside these categories, and NONE indicates that no final unresolved failure is present. The complete annotation protocol, including detailed label definitions and boundary rules, is provided in Appendix C. To validate the reliability of the GPT annotations, we stratify OSWorld tasks by domain and randomly sample 37 tasks, yielding 185 trajectories across the five methods. Two PhD-level AI researchers independently annotate the trajectories under blinded method identities using the same protocol.

Metrics. We report task success rate (SR) under different step budgets. SR-100, SR-50, and SR-15 impose maximum budgets of 100, 50, and 15 steps. Following OSWorld (Xie et al., 2024), one step corresponds to one planner decision. Control-failure comparisons use paired binary outcomes on the same tasks.

4.2 RESULTS

4.2.1 BACKBONE SENSITIVITY AND CROSS-STEP CONTROL FAILURES

Existing GUI-agent frameworks are consistently sensitive to backbone strength. We use GPT-5 results from the OSWorld leaderboard[2] as frontier references and evaluate Agent S3, GTA1, CoAct-1, and HiSA with Qwen3.5-9B while preserving each framework's native prompts, grounder, action space, tools, search procedure, and 100-step environment budget. As shown in Figure 1(b), SR-100 is 23.0–29.6 points lower in the Qwen3.5-9B evaluations, reducing the four-framework average from 60.9% to 35.2%. Under the same Qwen3.5-9B backbone, LocalLSTC reaches 47.6% SR-100, 12.4 points above the four-framework average.

Trajectory annotation further localizes this sensitivity to cross-step control behavior. As shown in Figure 1(c), the four baselines have a mean failed-trajectory control-failure incidence of 94.7% when averaged equally by method. Under the same method-level averaging, STOP, RECOVER, and ASSESS are the three most frequent categories, occurring in 78.6%, 42.2%, and 34.4% of failed trajectories, respectively.

Human verification supports the reliability of the aggregate annotation. The two annotators agree on 87.6% of ANY labels, with Cohen's $\kappa = 0.751$. Against the two annotators, GPT-5.6-sol achieves ANY F1 scores of 84.1% and 84.9%, with $\kappa = 0.632$ and 0.637, respectively. Appendix C.4 reports the complete validation results.

Across all OSWorld tasks, LocalLSTC reduces the incidence of observable cross-step control-failure manifestations, with ANY decreasing to 41.7% from a 63.7% average across the four baselines, a reduction of 22.0 percentage points. LocalLSTC also achieves the highest NONE incidence at 42.5%. Appendix C.5 reports the complete label distribution and uncertainty analysis.

4.2.2 BENCHMARK RESULTS AND RUNTIME BEHAVIOR

Table 1 summarizes the benchmark results across step budgets. On OSWorld, LocalLSTC with Qwen3.8-27B reaches $73.7 \pm 1.8\%$ SR-100, $72.3 \pm 2.0\%$ SR-50, and $61.1 \pm 4.5\%$ SR-15, achieving the strongest local results at all three step budgets. These results improve the previous strongest local scores by 19.8, 15.6, and 14.8 points, respectively. The advantage remains consistent across different step budgets, suggesting that LocalLSTC maintains stable control across varying interaction horizons and is robust to long-horizon error accumulation.

On WindowsAgentArena, the same configuration reaches $68.5 \pm 1.6\%$ SR-100, $67.9 \pm 0.6\%$ SR-50, and $57.5 \pm 4.9\%$ SR-15, establishing a new state of the art at all three step budgets. Its mean SR-50 improves the previous strongest local result by 22.6 points and exceeds the strongest API result by 4.4 points. The strong results across Ubuntu and Windows indicate that the control architecture generalizes across different operating environments without benchmark-specific training.

LocalLSTC also benefits consistently from stronger local reasoning backbones. Across the evaluated Qwen backbones and within the Gemma-4 family, stronger models improve SR at every evaluated step budget on both benchmarks. This scaling pattern indicates that stronger temporal control and stronger reasoning backbones provide complementary performance gains.

Table 2 summarizes LocalLSTC's runtime behavior across OSWorld domains with Qwen3.5-9B. Averaged over two runs, each task uses 26.1 planner steps and 9.9 subgoal segments, with 3.5 actions per subgoal and 1.3 actions per planner step. GUI actions account for 73.1% of execution, Bash accounts for 14.3%, and software APIs account for 5.2%. The execution mix varies substantially across domains. Bash reaches 54.0% in OS tasks, while GUI interaction dominates Chrome at 86.8% and GIMP at 86.3%. LocalLSTC performs 2.7 context refinements per task. Pass, Fail, and Bypass account for 56.0%, 16.7%, and 27.4%, respectively. WindowsAgentArena shows similar runtime patterns, with full statistics reported in Appendix D.

4.2.3 ABLATIONS AND MECHANISM ANALYSIS

We examine which mechanisms account for LocalLSTC's gains under a fixed Qwen3.5-9B backbone. Table 3 reports grouped ablations of L2S and S2L together with individual component ab-

[2] https://os-world.github.io/

Table 1: Success rates on OSWorld and WindowsAgentArena under step budgets of 100, 50, and 15. LocalLSTC results report mean±SD over two runs. Other methods use their reported results. Bold and underlining mark the highest and second-highest values, and "-" indicates an unreported result.

Access	Method / Backbone	OSWorld			WindowsAgentArena		
		SR-100↑	SR-50↑	SR-15↑	SR-100↑	SR-50↑	SR-15↑
API	o3 (OpenAI, 2025b)	23.0	17.2	9.1	-	-	-
	Agent S / GPT-4o (Agashe et al., 2025a)	-	-	20.6	-	-	18.2
	Navi / GPT-4V-1106 (Bonatti et al., 2025)	-	-	-	-	-	19.5
	computer-use-preview (OpenAI, 2025a)	30.5	31.3	26.0	-	-	-
	Claude 3.7 Sonnet (Anthropic, 2025a)	35.6	35.8	27.1	-	-	-
	Claude 4 Sonnet (Anthropic, 2025b)	41.4	43.9	31.2	-	-	-
	Agent S2 / Gemini 2.5 Pro (Agashe et al., 2025b)	-	45.8	34.6	-	41.4	-
	GTA1-32B / o3 (Yang et al., 2026b)	53.1	48.6	-	51.2	-	-
	UI-TARS-2-2509 (Wang et al., 2025a)	53.1	-	-	-	50.6	-
	CoAct-1 / CUA 4o + o3 + o4-mini (Song et al., 2026)	59.9	56.4	39.8	52.5	43.5	21.4
	Agentic-Lybic-Maestro (Guo et al., 2025)	61.9	56.9	-	-	-	-
	Claude Sonnet 4.5 (Anthropic, 2025c)	62.9	58.1	42.9	-	-	-
	Agent S3 / GPT-5 (Gonzalez-Pumariega et al., 2026)	63.5	-	-	50.2	49.0	-
	UiPath Screen Agent / Opus 4.5 (Cristescu et al., 2025)	67.1	64.4	-	-	-	-
	Agent S3 / GPT-5 bBoN (N=10) (Gonzalez-Pumariega et al., 2026)	69.9	-	-	56.6	54.1	-
	VLAA-GUI / Gemini 3 Flash (Han et al., 2026)	-	-	-	61.0	60.4	-
	OS-Symphony / GPT-5 (Yang et al., 2026a)	-	65.8	-	-	63.5	-
	Claude Sonnet 4.6 (Anthropic, 2026)	72.1	-	-	-	-	-
	Agent S3 / Opus 4.5 + GPT-5 bBoN (N=10) (Gonzalez-Pumariega et al., 2026)	72.6	-	-	-	-	-
	VLAA-GUI / Opus 4.5 (Han et al., 2026)	**76.3**	-	-	-	-	-
LOCAL	Qwen2.5-VL-7B (Bai et al., 2025)	-	-	-	-	3.4	3.4
	Qwen2.5-VL-72B (Bai et al., 2025)	5.0	-	4.4	-	9.7	-
	Kimi-VL-A3B (Kimi Team et al., 2025)	10.3	-	9.7	-	-	10.4
	InternVL3.5-8B (Wang et al., 2025b)	-	-	-	-	10.5	-
	InternVL3.5-38B (Wang et al., 2025b)	-	-	-	-	14.5	-
	OpenCUA-Qwen2-7B (Wang et al., 2025c)	23.1	20.6	19.9	-	-	-
	OpenCUA-7B (Wang et al., 2025c)	26.6	28.2	24.3	-	-	-
	UI-TARS-72B-DPO (Qin et al., 2025)	27.1	25.8	24.0	-	17.9	-
	UI-TARS-1.5-7B (Qin et al., 2025)	27.4	27.3	24.5	-	15.9	-
	OpenCUA-32B (Wang et al., 2025c)	34.8	34.1	29.7	-	-	-
	DeepMiner-Mano-7B (Fu et al., 2025)	40.2	-	-	-	-	-
	OpenCUA-72B-Preview (Wang et al., 2025c)	45.0	44.9	39.0	-	-	-
	OS-Symphony / Qwen3-VL-32B (Yang et al., 2026a)	-	-	-	-	45.3	-
	EvoCUA-8B-20260105 (Xue et al., 2026)	-	46.1	-	-	-	-
	AutoGLM-OS-9B (Lai et al., 2026)	-	47.3	46.3	-	-	-
	DeepMiner-Mano-72B (Fu et al., 2025)	53.9	-	-	-	-	-
	GUI-Owl-1.5-32B (Xu et al., 2026a)	-	55.4	-	-	-	-
	EvoCUA-20260105 (Xue et al., 2026)	-	56.7	-	-	-	-
Ours	LocalLSTC / Gemma-4-E4B-it	28.1±2.1	26.9±2.7	20.3±0.6	35.9±4.6	33.7±1.6	20.7±4.6
	LocalLSTC / Qwen3.5-4B	37.3±4.8	35.5±3.3	24.7±1.2	35.9±1.6	31.7±1.3	22.5±2.0
	LocalLSTC / Qwen3.5-9B	47.6±2.7	44.5±3.4	34.8±2.1	43.1±0.5	39.2±1.3	28.8±4.1
	LocalLSTC / Gemma-4-26B-A4B-it	60.3±0.2	54.7±2.4	42.2±1.4	53.3±1.1	44.6±4.6	35.9±1.6
	LocalLSTC / Qwen3.6-27B	64.7±0.6	62.7±0.4	51.4±1.9	59.8±1.6	59.8±1.6	45.7±0.0
	LocalLSTC / Qwen3.8-27B	73.7±1.8	**72.3**±2.0	**61.1**±4.5	**68.5**±1.6	**67.9**±0.6	**57.5**±4.9

Table 2: Domain-level runtime behavior of LocalLSTC with Qwen3.5-9B on OSWorld, averaged over two runs. Execution-channel columns report action shares, with the remainder reserved for control actions. Pass, Fail, and Bypass are mutually exclusive trajectory categories based on Final Verification history. Bypass takes precedence for consecutive-stall or maximum-step cutoffs without an explicit `INFEASIBLE` action. In all other cases, a trajectory is classified as Pass if it has no rejected Final Verification and as Fail if it has at least one, even when a later verification succeeds. An explicit `INFEASIBLE` action counts as an accepted Final Verification outcome, with earlier rejections still counted.

Summary			Commitment				Execution Channels			Evidence Control			
Domain	Tasks	SR↑ (%)	steps	Subgoals / Task	Actions / Subgoal	Actions / step	GUI (%)	Bash (%)	API (%)	Context Refines	Pass (%)	Fail (%)	Bypass (%)
Chrome	46	59.7	20.4	7.8	2.9	1.1	86.8	1.3	2.7	2.0	64.1	13.0	22.8
GIMP	26	78.8	14.3	5.1	3.9	1.4	86.3	7.4	0.0	1.1	75.0	5.8	19.2
LibreOffice Calc	47	28.7	30.1	11.3	4.2	1.6	75.5	5.0	13.3	3.2	42.6	20.2	37.2
LibreOffice Impress	47	33.4	24.7	8.1	4.7	1.6	81.0	1.6	9.5	2.2	41.5	19.1	39.4
LibreOffice Writer	23	39.1	21.7	8.0	4.2	1.6	83.4	3.8	4.7	2.1	45.7	19.6	34.8
Multi-apps	101	32.2	36.1	14.1	3.2	1.3	65.6	25.1	2.4	3.9	55.4	16.8	27.7
OS	24	83.3	17.8	6.1	3.2	1.1	38.3	54.0	0.0	1.6	77.1	14.6	8.3
Thunderbird	15	73.3	24.9	9.5	3.3	1.3	85.7	9.6	0.0	2.7	73.3	13.3	13.3
VLC	17	58.7	24.3	10.1	2.7	1.1	68.5	15.7	4.2	2.6	61.8	17.6	20.6
VS Code	23	69.6	16.4	7.5	2.8	1.3	68.2	20.6	0.7	1.6	50.0	23.9	26.1
All	369	47.6	26.1	9.9	3.5	1.3	73.1	14.3	5.2	2.7	56.0	16.7	27.4

lations of the L2S–S2L control loop. Appendix E specifies the exact settings for all grouped and component ablations.

Table 3: Grouped-mechanism and component ablations on OSWorld with Qwen3.5-9B. Resource metrics report per-task averages. Bold marks the highest SR or lowest resource value in each column.

Category	Method	SR-100↑	Tokens (k)↓	Prompt (k)↓	Completion (k)↓	Steps↓	Time (s)↓
Full	**LocalLSTC**	**47.6**	252.2	237.6	14.6	26.1	390.4
BOUNDARY	w/o L2S	36.4	**192.9**	**179.1**	13.8	31.4	382.2
	w/o S2L	41.5	246.5	229.1	17.4	27.1	472.7
COMPONENTS	w/o persistent-subgoal interface	38.0	276.6	257.3	19.3	29.6	554.4
	w/o candidate proposals	40.7	273.0	259.1	13.9	31.2	412.5
	w/o multi-action list	43.5	260.7	243.0	17.8	28.8	445.5
	w/o Step Abstraction	43.0	276.2	261.1	15.1	31.9	428.7
	w/o state-conditioned routing	43.9	193.5	180.1	13.4	**22.8**	**365.9**
	w/o stall/loop handling	45.1	285.0	266.9	18.1	29.8	517.2
	w/o Final Verification	44.7	227.8	214.7	**13.1**	24.1	429.0

Table 4: Runtime-event diagnostics for Full, `w/o L2S`, and `w/o S2L` on OSWorld. Rate reports the fraction of tasks that enter each event at least once, and Score reports mean evaluator score conditional on event entry. Bold marks the column maximum, and "–" indicates an event unavailable after removing the corresponding mechanism.

	Stall / repetition		Recovery entry		Rejected termination	
Method	**Rate (%)**	**Score (%)↑**	**Rate (%)**	**Score (%)↑**	**Rate (%)**	**Score (%)↑**
Full	4.5	**34.3**	**60.0**	**34.0**	**32.1**	**38.9**
`w/o L2S`	4.3	18.8	56.5	20.9	19.3	23.1
`w/o S2L`	**4.6**	11.8	–	–	–	–

Removing L2S lowers SR-100 from 47.6% to 36.4%, while removing S2L lowers it to 41.5%, showing that performance degrades when either grouped mechanism is removed under the same Qwen3.5-9B backbone. Among individual components, removing the persistent-subgoal interface produces the largest reduction of 9.6 points. Removing candidate proposals, Step Abstraction, the multi-action list, state-conditioned routing, Final Verification, and stall/loop handling reduces SR by 6.9, 4.6, 4.1, 3.7, 2.9, and 2.5 points, respectively. These reductions span both commitment formation and cross-step evidence control, supporting contributions from both temporal scopes.

Several components improve both success and interaction efficiency. Removing Step Abstraction increases token use from 252.2k to 276.2k, planner steps from 26.1 to 31.9, and runtime from 390.4 to 428.7 seconds, while removing the persistent-subgoal interface raises token use to 276.6k and runtime to 554.4 seconds. Without candidate proposals, token use and planner steps increase to 273.0k and 31.2. These results suggest that stronger control decisions offset their inference cost by reducing downstream interaction.

Table 4 separates event frequency from post-event behavior. Stall or repetition incidence is similar across Full, `w/o L2S`, and `w/o S2L` at 4.5%, 4.3%, and 4.6%. Conditional on such events, Full reaches a mean evaluator score of 34.3%, compared with 18.8% for `w/o L2S` and 11.8% for `w/o S2L`. These descriptive differences are consistent with improved recovery.

The pattern extends to recovery and termination. Recovery entry and rejected termination are S2L-defined events and are therefore unavailable for `w/o S2L`. Among the remaining variants, Full reaches 34.0% after recovery entry compared with 20.9% for `w/o L2S`, while the scores after rejected termination are 38.9% and 23.1%. These event-conditioned results provide mechanism-specific diagnostics consistent with the grouped ablations, with representative trajectory-level examples provided in Appendix F.

5 CONCLUSION

LocalLSTC addresses the cross-step control burden in locally deployed GUI agents by organizing persistent state and short-term execution commitments. Long-to-Short Planning forms each commitment from persistent state, while Short-to-Long Control integrates execution outcomes back for progress assessment, recovery, and termination. LocalLSTC improves task success and reduces observable cross-step control-failure manifestations, with ablations supporting both mechanisms. With Qwen3.8-27B, it reaches 73.7% SR-100 on OSWorld and 68.5% on WindowsAgentArena, achieving the strongest local OSWorld result and a new state of the art on WindowsAgentArena. These results show that temporal control organization complements backbone scaling.

Reproducibility Statement

We provide documentation and artifacts to support reproduction of both the LocalLSTC framework and the reported experiments. Sections 3 and 4 describe the framework and experimental protocol. Appendix C details the control-failure annotation and human-verification procedures. Appendix A documents implementation details, while Appendix B provides the prompt templates. Code is released through an anonymous repository to support reproducibility.

AI Use Statement

GPT-5.6-sol was used for post-hoc trajectory annotation, language polishing, and LaTeX revision. OpenAI Codex was used to retrieve and identify relevant literature and to draft the Related Work section. The authors independently reviewed the AI-generated content and manually verified all cited works and the claims attributed to them against the original sources. All AI-assisted analyses and revisions were reviewed by the authors, who take full responsibility for the accuracy and content of the final manuscript.

References

Saaket Agashe, Jiuzhou Han, Shuyu Gan, Jiachen Yang, Ang Li, and Xin Eric Wang. Agent s: An open agentic framework that uses computers like a human. In *The Thirteenth International Conference on Learning Representations*, 2025a. URL https://openreview.net/forum?id=lIVRgt4nLv.

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*, 2025b. URL https://openreview.net/forum?id=zg5is4GJ3R.

Anthropic. Claude 3.7 Sonnet and Claude Code, February 2025a. URL https://www.anthropic.com/news/claude-3-7-sonnet.

Anthropic. Introducing Claude 4, May 2025b. URL https://www.anthropic.com/news/claude-4.

Anthropic. Introducing Claude Sonnet 4.5, September 2025c. URL https://www.anthropic.com/news/claude-sonnet-4-5.

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

Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, et al. Qwen2.5-VL technical report. *arXiv preprint arXiv:2502.13923*, 2025. doi: 10.48550/arXiv.2502.13923. URL https://arxiv.org/abs/2502.13923.

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*, volume 267 of *Proceedings of Machine Learning Research*, pp. 4874–4910. PMLR, 2025. URL https://proceedings.mlr.press/v267/bonatti25a.html.

Horia Cristescu, Charles Park, Trong Canh Nguyen, Sergiu Talmacel, Alexandru-Gabriel Ilie, and Stefan Adam. UI-CUBE: Enterprise-grade computer use agent benchmarking beyond task accuracy to operational reliability. *arXiv preprint arXiv:2511.17131*, 2025. doi: 10.48550/arXiv.2511.17131. URL https://arxiv.org/abs/2511.17131.

Tianyu Fu, Anyang Su, Chenxu Zhao, Hanning Wang, Minghui Wu, Zhe Yu, Fei Hu, Mingjia Shi, Wei Dong, Jiayao Wang, et al. Mano technical report. *arXiv preprint arXiv:2509.17336*, 2025. doi: 10.48550/arXiv.2509.17336. URL https://arxiv.org/abs/2509.17336.

Gonzalo Gonzalez-Pumariega, Vincent Tu, Chih-Lun Lee, Jiachen Yang, Ang Li, and Xin Eric Wang. Scaling agents for computer use. *Transactions on Machine Learning Research*, 2026. URL https://openreview.net/forum?id=eve4jBYa8D.

Google DeepMind. Gemma 4, April 2026. URL https://deepmind.google/models/gemma/gemma-4/.

Liangxuan Guo, Bin Zhu, Qingqian Tao, Kangning Liu, Xun Zhao, Xianzhe Qin, Jin Gao, and Guangfu Hao. Agentic lybic: Multi-agent execution system with tiered reasoning and orchestration. *arXiv preprint arXiv:2509.11067*, 2025. doi: 10.48550/arXiv.2509.11067. URL https://arxiv.org/abs/2509.11067.

Linqiang Guo, Wei Liu, Yi Wen Heng, Tse-Hsun (Peter) Chen, and Yang Wang. Agent-sama: State-aware mobile assistant. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 29459–29467, 2026. doi: 10.1609/aaai.v40i35.40187. URL https://ojs.aaai.org/index.php/AAAI/article/view/40187.

Qijun Han, Haoqin Tu, Zijun Wang, Haoyue Dai, Yiyang Zhou, Nancy Lau, Alvaro A. Cardenas, Yuhui Xu, Ran Xu, Caiming Xiong, et al. VLAA-GUI: Knowing when to stop, recover, and search, a modular framework for gui automation. *arXiv preprint arXiv:2604.21375*, 2026. doi: 10.48550/arXiv.2604.21375. URL https://arxiv.org/abs/2604.21375.

Kimi Team, Angang Du, Bohong Yin, Bowei Xing, Bowen Qu, Bowen Wang, Cheng Chen, Chenlin Zhang, Chenzhuang Du, Chu Wei, et al. Kimi-VL technical report. *arXiv preprint arXiv:2504.07491*, 2025. doi: 10.48550/arXiv.2504.07491. URL https://arxiv.org/abs/2504.07491.

Hanyu Lai, Xiao Liu, Yanxiao Zhao, Han Xu, Hanchen Zhang, Bohao Jing, Yanyu Ren, Shuntian Yao, Yuxiao Dong, and Jie Tang. ComputerRL: Scaling end-to-end online reinforcement learning for computer use agents. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://iclr.cc/virtual/2026/poster/10007435.

Aiden Yiliu Li, Xinyue Hao, Shilong Liu, and Mengdi Wang. Avenir-web: Human-experience-imitating multimodal web agents with mixture of grounding experts. *arXiv preprint arXiv:2602.02468*, 2026a. doi: 10.48550/arXiv.2602.02468. URL https://arxiv.org/abs/2602.02468.

Weiming Li, Hye-young Paik, and Yulei Sui. HiSA: Hierarchical state abstraction for scalable GUI agents. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 11965–11985, San Diego, California, United States, July 2026b. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.581. URL https://aclanthology.org/2026.findings-acl.581/.

Zechun Liu, Changsheng Zhao, Forrest Iandola, Chen Lai, Yuandong Tian, Igor Fedorov, Yunyang Xiong, Ernie Chang, Yangyang Shi, Raghuraman Krishnamoorthi, Liangzhen Lai, and Vikas Chandra. MobileLLM: Optimizing sub-billion parameter language models for on-device use cases. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 32431–32454. PMLR, 2024. URL https://proceedings.mlr.press/v235/liu24ce.html.

OpenAI. Computer-using agent, January 2025a. URL https://openai.com/index/computer-using-agent/.

OpenAI. Introducing OpenAI o3 and o4-mini, April 2025b. URL https://openai.com/index/introducing-o3-and-o4-mini/.

Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, et al. UI-TARS: Pioneering automated GUI interaction with native agents. *arXiv preprint arXiv:2501.12326*, 2025. doi: 10.48550/arXiv.2501.12326. URL https://arxiv.org/abs/2501.12326.

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

Qwen Team. Qwen3.6-27B: Flagship-level coding in a 27B dense model, April 2026b. URL https://qwen.ai/blog?id=qwen3.6-27b.

Qwen Team. Qwen3.8-Max: A new bar for coding and cowork, August 2026c. URL https://qwen.ai/blog?id=qwen3.8.

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*, 2026. URL https://iclr.cc/virtual/2026/poster/10007725.

Fei Tang, Zhiqiong Lu, Boxuan Zhang, Weiming Lu, Jun Xiao, Yueting Zhuang, and Yongliang Shen. Clawgui: A unified framework for training, evaluating, and deploying gui agents. *arXiv preprint arXiv:2604.11784*, 2026. doi: 10.48550/arXiv.2604.11784. URL https://arxiv.org/abs/2604.11784.

Shizuo Tian, Hao Wen, Yuxuan Chen, Jiacheng Liu, Shanhui Zhao, Guohong Liu, Ju Ren, Yunxin Liu, and Yuanchun Li. Agentprog: Empowering long-horizon GUI agents with program-guided context management. In *Proceedings of the 24th Annual International Conference on Mobile Systems, Applications and Services*, pp. 883–910. Association for Computing Machinery, 2026. doi: 10.1145/3745756.3809245. URL https://doi.org/10.1145/3745756.3809245.

Madhumitha Venkatesan, Shicheng Wen, Jiajing Guo, Jorge Piazentin Ono, Liu Ren, and Dongyu Liu. Plover: Steering GUI agents through plan-centric interaction. *arXiv preprint arXiv:2607.15193*, 2026. doi: 10.48550/arXiv.2607.15193. URL https://arxiv.org/abs/2607.15193.

Haoming Wang, Haoyang Zou, Huatong Song, Jiazhan Feng, Junjie Fang, Junting Lu, Longxiang Liu, Qinyu Luo, Shihao Liang, Shijue Huang, et al. UI-TARS-2 technical report: Advancing GUI agent with multi-turn reinforcement learning. *arXiv preprint arXiv:2509.02544*, 2025a. doi: 10.48550/arXiv.2509.02544. URL https://arxiv.org/abs/2509.02544.

Taiyi Wang, Sian Gooding, Florian Hartmann, Oriana Riva, and Edward Grefenstette. A subgoal-driven framework for improving long-horizon LLM agents. *arXiv preprint arXiv:2603.19685*, 2026. doi: 10.48550/arXiv.2603.19685. URL https://arxiv.org/abs/2603.19685.

Weiyun Wang, Zhangwei Gao, Lixin Gu, Hengjun Pu, Long Cui, Xingguang Wei, Zhaoyang Liu, Linglin Jing, Shenglong Ye, Jie Shao, et al. InternVL3.5: Advancing open-source multimodal models in versatility, reasoning, and efficiency. *arXiv preprint arXiv:2508.18265*, 2025b. doi: 10.48550/arXiv.2508.18265. URL https://arxiv.org/abs/2508.18265.

Xinyuan Wang, Bowen Wang, Dunjie Lu, Junlin Yang, Tianbao Xie, Junli Wang, Jiaqi Deng, Xiaole Guo, Yiheng Xu, et al. OpenCUA: Open foundations for computer-use agents. In *Advances in Neural Information Processing Systems*, volume 38, pp. 155016–155066, 2025c. doi: 10.52202/085713-4669. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/cc7ae529e945226b0d52ea4ac478c4f3-Abstract-Conference.html.

Zhiyong Wu, Zhenyu Wu, Fangzhi Xu, Yian Wang, Qiushi Sun, Chengyou Jia, Kanzhi Cheng, Zichen Ding, Liheng Chen, Paul Pu Liang, and Yu Qiao. OS-ATLAS: Foundation action model for generalist GUI agents. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=n9PDaFNi8t.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *Advances in Neural Information Processing Systems*, volume 37, pp. 52040–52094, 2024. doi: 10.52202/079017-1650. URL

https://proceedings.neurips.cc/paper_files/paper/2024/hash/5d413e48f84dc61244b6be550f1cd8f5-Abstract-Datasets_and_Benchmarks_Track.html.

Haiyang Xu, Xi Zhang, Haowei Liu, Junyang Wang, Zhaozai Zhu, Shengjie Zhou, Xuhao Hu, Feiyu Gao, Junjie Cao, Zihua Wang, et al. Mobile-Agent-v3.5: Multi-platform fundamental GUI agents. *arXiv preprint arXiv:2602.16855*, 2026a. doi: 10.48550/arXiv.2602.16855. URL https://arxiv.org/abs/2602.16855.

Tianshi Xu, Huifeng Wen, and Meng Li. Adapting the interface, not the model: Runtime harness adaptation for deterministic LLM agents. *arXiv preprint arXiv:2605.22166*, 2026b. doi: 10.48550/arXiv.2605.22166. URL https://arxiv.org/abs/2605.22166.

Yiheng Xu, Zekun Wang, Junli Wang, Dunjie Lu, Tianbao Xie, Amrita Saha, Doyen Sahoo, Tao Yu, and Caiming Xiong. Aguvis: Unified pure vision agents for autonomous GUI interaction. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 69772–69805. PMLR, 2025. URL https://proceedings.mlr.press/v267/xu25ae.html.

Taofeng Xue, Chong Peng, Mianqiu Huang, Linsen Guo, Tiancheng Han, Haozhe Wang, Jianing Wang, Xiaocheng Zhang, Xin Yang, Dengchang Zhao, et al. EvoCUA: Evolving computer use agents via learning from scalable synthetic experience. *arXiv preprint arXiv:2601.15876*, 2026. doi: 10.48550/arXiv.2601.15876. URL https://arxiv.org/abs/2601.15876.

Zhenliang Xue, Yixin Song, Zeyu Mi, Xinrui Zheng, Yubin Xia, and Haibo Chen. Powerinfer-2: Fast large language model inference on a smartphone. *arXiv preprint arXiv:2406.06282*, 2024. doi: 10.48550/arXiv.2406.06282. URL https://arxiv.org/abs/2406.06282.

Bowen Yang, Kaiming Jin, Zhenyu Wu, Zhaoyang Liu, Qiushi Sun, Zehao Li, JingJing Xie, Zhoumianze Liu, Fangzhi Xu, Kanzhi Cheng, Yian Wang, Qingyun Li, Yu Qiao, Zun Wang, and Zichen Ding. OS-symphony: A holistic framework for robust and generalist computer-using agents. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 22300–22330, San Diego, California, United States, July 2026a. Association for Computational Linguistics. doi: 10.18653/v1/2026.acl-long.1021. URL https://aclanthology.org/2026.acl-long.1021/.

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*, 2026b. URL https://iclr.cc/virtual/2026/poster/10011639.

Zhong Zhang, Yaxi Lu, Yikun Fu, Yupeng Huo, Shenzhi Yang, Yesai Wu, Han Si, Xin Cong, Haotian Chen, Yankai Lin, Jie Xie, Wei Zhou, Wang Xu, Yuanheng Zhang, Zhou Su, Zhongwu Zhai, Xiaoming Liu, Yudong Mei, Jianming Xu, Hongyan Tian, Chongyi Wang, Chi Chen, Yuan Yao, Zhiyuan Liu, and Maosong Sun. AgentCPM-GUI: Building mobile-use agents with reinforcement fine-tuning. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 155–180, Suzhou, China, November 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-demos.12. URL https://aclanthology.org/2025.emnlp-demos.12/.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 62138–62160. PMLR, 2024. URL https://proceedings.mlr.press/v235/zhou24r.html.

A Implementation Details

This section specifies how LocalLSTC instantiates the control flow in Section 3. Each planner call constructs a Short-Term Execution Commitment from the current observation and Long-Term Control State. The runtime validates and executes the commitment, converts the resulting observations

into subgoal-aligned evidence, and uses this evidence to update the persistent control state for the next interaction.

A.1 Planner Context and Interface

Each planner call conditions on the task instruction, current screenshot, active subgoal, compact evidence history, runtime feedback, available action channels, and domain-specific operational guidance. The planner receives the GUI interaction instructions, platform-specific Bash instructions, and software API schemas available for the relevant applications in the same call. This unified context allows each commitment to select or combine channels based on the current state.

Planner context combines refined history with recent abstracted steps so that compressed cross-step evidence remains available together with recent execution detail. Runtime feedback records loop detections, no-change outcomes, tool failures, and rejected termination. Application-specific skills and software API schemas are initialized from the task-level `related_apps` set. For tasks involving multiple applications, a recognized foreground application focuses the active set on that application and the operating-system domain when applicable. The full related set remains active when foreground detection does not identify a matching application.

The planner returns a JSON object containing `thought`, `subgoal`, and `actions`. The `subgoal` field specifies the execution stage of the current commitment. The reserved value `continue` retains the active subgoal, while another value proposes a transition to a new subgoal. The runtime canonicalizes the field by stripping surrounding whitespace and collapsing repeated whitespace, then compares the resulting strings exactly. The nonempty `actions` list specifies the ordered actions executed under the commitment.

Executable actions support GUI interaction, software API calls, and Bash execution. GUI interactions are represented as executable `pyautogui` code strings. Software API calls use schema-exposed `Class.method(...)` call strings. The application-specific API schemas and registered execution handlers are adopted directly from ComputerRL (Lai et al., 2026). LocalLSTC changes how these calls are selected and integrated into temporal control. Bash execution uses `bash_execution` strings, which the runtime interprets as Bash commands or Python snippets according to the active platform skill. A commitment can combine actions from these channels when a subgoal requires coordinated interaction across interfaces.

The action space also includes the reserved control actions `WAIT`, `TERMINATE`, and `INFEASIBLE`. `WAIT` retains the active subgoal while the runtime observes for an environment update or timeout. `TERMINATE` requests Final Verification after preceding executable actions complete. `INFEASIBLE` records an infeasibility judgment. Terminal actions are placed at the end of a commitment, and validation rejects a commitment that contains both `TERMINATE` and `INFEASIBLE`.

The runtime recognizes reserved control actions directly and identifies GUI interaction and software API calls from their registered prefixes. Other executable strings enter the Bash channel, where the active platform skill determines their interpretation. Actions are dispatched in list order, and an execution failure stops the remaining actions in the commitment. Candidate generation and selection occur within the same planner completion. The planner autoregressively emits one to three candidate proposals, followed by a `Best` index and selection rationale, and finally the executable JSON object. No additional sampling, planner invocation, or external judge is used. Only the final JSON object is parsed for execution.

A.2 Validation and Canonicalization

Planner outputs are canonicalized before execution through list-level, channel-level, and schema-level validation. List validation enforces terminal-action placement and rejects commitments containing both `TERMINATE` and `INFEASIBLE`. Channel canonicalization removes wrapper artifacts from `bash_execution` input, converts GUI scripts into executable `pyautogui` code, and constructs stable fingerprints used by loop control. Software API validation restricts each action to a single callable expression defined by the exposed schema and implemented by a registered handler. Validated parameter literals are converted to runtime values before execution.

The runtime first applies local JSON repair when strict JSON parsing fails. Remaining parsing or validation failures are returned to the planner together with the original response and the corresponding error message. The repair interface allows up to three attempts for each planner call before the configured runtime failure handling applies.

A.3 Step Abstraction

Step Abstraction evaluates each execution outcome against the intended effect of the current commitment and converts channel-specific observations into a common subgoal-aligned evidence representation. Each evidence record retains the executed action, observed result, task-relevant progress, execution exceptions, and relevant environment changes.

For GUI interaction, planner-generated `pyautogui` code is parsed with an abstract syntax tree (AST) and divided into observation-aligned execution chunks. For each coordinate-dependent statement, the runtime extracts a referring expression from contiguous comments immediately preceding the statement or from its trailing # comment and passes it as the GTA1-7B target description. If no comment is available, a commitment containing one grounding operation falls back to the planner's `thought`. A multi-grounding commitment uses a generic description formed from the operation index and current sub-action, while a single operation lacking either source uses a screenshot-conditioned default target description. Planner-provided coordinates are treated as proposals and are normally replaced through GTA1-7B grounding. Drag endpoints are grounded separately, and scrolling may replace or insert a coordinate identifying the scroll region. Non-spatial operations such as `write`, `press`, and `hotkey` bypass visual grounding. After each chunk, the runtime acquires a stable screenshot and compares it with the pre-action screenshot to determine whether the interface changed. The stable frame becomes the observation for the next chunk.

The runtime measures screenshot stability across consecutive grayscale captures using the normalized area of the bounding box enclosing changed pixels. Captures below the configured change threshold extend the stable sequence, while larger changes restart it. Initial, post-action, and pre-evaluation observations apply the stage-specific stability requirements and timeouts summarized in Table 5. When stabilization times out, the runtime retains the latest available frame and records a warning. Screenshot acquisition failure yields an unavailable observation. Planner observations and Final Verification use screenshot-availability polling.

To localize visual change, the runtime computes the pixel-wise absolute difference between pre-action and post-action screenshots and extracts the bounding box of changed pixels with a 50-pixel margin on each side. When no changed region is detected, Step Abstraction uses the full screenshot. The resulting visual transition is interpreted jointly with the task instruction, active subgoal, executed action, and available recovery feedback.

For `bash_execution`, the evidence record retains the executed command, execution status, exit code, textual output, and available readback evidence. Long outputs are compressed while preserving task-relevant identifiers such as paths, filenames, URLs, IDs, table names, column names, and selected values. Software API evidence similarly retains the callable, execution status, return value, helper observation, and error message. GUI-mediated API execution additionally follows the visual observation policy above. All channels enter compact history through the same evidence interface.

A.4 State and Loop Control

After Step Abstraction, the runtime assigns a transient routing state using a fixed precedence. `INFEASIBLE` is handled as a direct runtime failure before the routing-state ladder. A final `TERMINATE` request maps to `finish`. Execution failure maps to `error`. Repeated no-progress evidence maps to `stall` when the proposed subgoal remains the current subgoal. Otherwise, a changed subgoal maps to `advance`. Remaining executions map to `continue`. State assignment occurs before a proposed subgoal transition is committed, keeping the resulting evidence aligned with the execution subgoal that produced it.

The following pseudocode makes the subgoal resolution, routing precedence, and refinement trigger explicit. Here, `normalize` strips surrounding whitespace and collapses repeated whitespace, and $I_{\min}$ and $I_{\max}$ are the configured minimum and maximum refinement intervals.

```
raw = normalize(decision.subgoal)
proposed = current_subgoal if raw.lower() == "continue" else raw
steps_since_refinement += 1

if INFEASIBLE was reached:
   status = failure
   return

if execution was suppressed by loop control:
   record(loop_stall)
   consecutive_stall_count += 1
   feedback = "Switch target/tool/approach before retrying"
   if steps_since_refinement >= I_max:
      Refine(accumulated_evidence)
      steps_since_refinement = 0
   if consecutive_stall_count >= 3:
      status = failure
      return
   status = running
   return

executed = outcome.executed_prefix
evidence = Update(evidence, SA(proposed, executed, outcome))

if TERMINATE was reached: route = finish
elif execution failed: route = error
elif proposed == current_subgoal and 3 no-change steps under
    current_subgoal:
   route = stall
elif proposed != current_subgoal: route = advance
else: route = continue

if route in {continue, advance}: feedback = None
if route == advance: current_subgoal = proposed
if route == error: feedback = execution_error
if route == stall: feedback = no_progress
if route == finish:
   if Verify(evidence):
      status = success
      return
   route = stall
   feedback = rejected_termination

if route == stall: consecutive_stall_count += 1
else: consecutive_stall_count = 0
if consecutive_stall_count >= 3:
   status = failure
   return

if steps_since_refinement >= I_min and route == advance:
   Refine(accumulated_evidence)
   steps_since_refinement = 0
elif steps_since_refinement >= I_max:
   Refine(accumulated_evidence)
   steps_since_refinement = 0

status = running
```

No-change evidence is aggregated once per planner step, so a multi-action commitment contributes at most one no-change observation. Three consecutive no-change steps under the same subgoal produce `stall`. A proposed transition to a different subgoal takes the `advance` route before the same-subgoal stall rule applies. Steps containing `bash_execution` are excluded from visual no-change aggregation because textual readback may establish progress in the absence of a corresponding interface change. Each `stall` increments the consecutive-stall counter, while any other

route resets it. Three consecutive stalls reach the runtime failure boundary. Runtime feedback is transition-local. Nominal `continue` and `advance` routes clear prior feedback, while `error`, `stall`, and rejected termination write feedback for the next planner call. The refinement counter increments once per planner step and resets after each refinement.

Loop detection compares normalized action and result fingerprints across interactions and suppresses execution when repetition thresholds are reached. Before GUI/API/Bash dispatch, the runtime performs two checks. First, for a single `gui_action`, a third consecutive occurrence of the identical raw `pyautogui` string records the current step as `loop_stall` and skips that GUI execution. Second, repeated action–result pairs suppress the next execution after three repetitions for `bash_execution` and software API actions or five repetitions for GUI actions. A suppressed decision still consumes one planner step, increments the consecutive-stall counter, checks the maximum-interval fallback for Context Refinement, and injects recovery feedback that directs the next planner call to change the target, tool, or approach. This suppression occurs before execution. The no-change rule above applies after execution. Three consecutive stall events terminate the task with failure. Because the inserted `loop_stall` record interrupts the sequence of identical GUI logs, suppression blocks the current repetition and allows the action to be reconsidered after replanning. Repetition counts use at most one result from each planner step, preventing count inflation from multiple actions within one commitment.

The evidence record combines visual observations, execution logs, textual readback, software API returns, and abstracted steps. Textual readback may come from command output or subsequent file inspection. GUI context remains available across text-based actions, allowing confirmed textual evidence to support subsequent planning and Final Verification together with visual evidence.

A.5 Context Refinement

LocalLSTC adopts HiSA's context-refinement operation and schedules it through execution-state triggers in place of a fixed period. An `advance` triggers refinement of accumulated evidence once the minimum refinement interval is reached. The first refinement summarizes the available abstracted history, while later refinements incorporate only steps not already represented in the existing summary. A maximum refinement interval provides a fallback for long trajectories without an earlier trigger.

Subsequent planner calls and Final Verification receive the refined summary together with recent uncompressed steps. This preserves compact cross-step context while retaining local evidence from the most recent interactions.

A.6 Final Verification

A `TERMINATE` request invokes Final Verification using the task instruction, active subgoal, compact execution history, initial screenshot, and latest screenshot. When refined context is available, the verifier combines the summary with abstracted steps not yet represented in it. Otherwise, verification uses the full compact history. For office-style tasks modified through `bash_execution`, the runtime may reopen the target file before verification so that the current interface provides refreshed visual evidence.

The verifier returns `PASS` or `FAIL`, with invalid responses mapped to `FAIL`. A `PASS` confirms termination. A `FAIL` records the rejected termination as a `stall` and returns execution to planning with recovery feedback that requests targeted verification or completion of the remaining requirement. The subsequent planner call receives this feedback together with a newly acquired current screenshot. Rejected termination follows the same consecutive-stall failure boundary as other stall events.

A.7 Runtime Parameters

Table 5 reports the runtime parameters shared across LocalLSTC experiments.

Table 5: Runtime parameters shared across LocalLSTC experiments.

Rule or hyperparameter	Value
Model-inference temperature	0
Model-inference `top_p`	0.95
Model-inference `top_k`	20
Model-inference sampling seed	42
Model-inference thinking token budget	16384
Maximum candidate proposals per inference step	3
Minimum context-refinement interval	5 steps
Maximum context-refinement interval	20 steps
`bash_execution` timeout	180 seconds
Maximum planner parsing or repair attempts	3
Post-action observation timeout	10 seconds
Explicit `WAIT` timeout	20 seconds
Screenshot acquisition timeout	6 seconds
Evaluation retry timeout	25 seconds
Initial / pre-evaluation screenshot stabilization	5 low-change observations, 30-second timeout
Post-action screenshot stability check	2 stable frames, 10-second timeout
Planner-side repeated GUI decision threshold	3 consecutive planner calls
Repeated GUI action–result threshold	5 repetitions
Repeated `bash_execution` or software API action–result threshold	3 repetitions
No-change threshold under the same subgoal	3 consecutive abstracted steps
Consecutive stall-to-fail threshold	3 stalls

B Prompt Templates

This appendix reports the core prompt templates used by the full LocalLSTC configuration in our desktop experiments. Each planner call concatenates the global planner prompt, response-format prompt, enabled software-API guidance, GUI guidance, platform-specific Bash/Python guidance, relevant application API schemas, domain skills, and any active recovery guidance. The API, GUI, and platform-execution guidance remains available together on every call, independent of the preceding action. Prompt changes used by the ablation configurations are summarized in Section B.8.

B.1 Planner System Prompt

System Prompt: Global Planner

```
You are an expert in GUI interaction, execution-side automation, and
software API tools. Always keep the task instruction in mind.

# General Instructions
- GUI, software API, and platform-execution guidance are available
together in every planner call. Choose the shortest reliable channel
for the current state, and combine channels when the task requires it.
- Reassess the best channel from the current screenshot, execution
history, and runtime feedback. Do not continue using a channel merely
because the preceding action used it.

# Action Types
## api
Call one software API exposed for one of the current related apps. Use
 this when the task maps cleanly to an app-specific software operation
. Represent it as a bare `Class.method(...)` action string. Use only
methods explicitly listed in the current api tools section. Never
```

```
invent method names. Use Python literals such as `True`, `False`, and
`None`, not JSON/JavaScript literals like `true`, `false`, or `null`.

## gui_action
Execute pyautogui code against the visible GUI. Use this when the task
 should be completed through visible interface interactions. Represent
 it as a bare `pyautogui...` action string.

## wait
Wait for async operations to complete and observe UI changes.
Represent it as the exact action string `WAIT`.

## bash_execution
Execute execution-side automation through the `bash_execution` tool.
This is a historical tool name: on Linux, output Bash commands or
Python scripts; on Windows, output Python code snippets executed in
the Windows guest. Use this when file-side editing, inspection, or
automation is more reliable than GUI interaction. Follow the platform-
specific skill section for paths, commands, and examples. Represent it
 as a bare command/code action string.

## termination
Request task completion after the required result has been verified.
Represent it as the exact action string `TERMINATE` and explain the
completion evidence in `thought`.

## infeasible
Declare that the task is objectively impossible to complete. Represent
 it as the exact action string `INFEASIBLE`.
- Some tasks may be infeasible by design. If required capabilities,
variables, or app features are unavailable, use `INFEASIBLE` instead
of `TERMINATE` and explain the blocker in `thought`.
- Treat explicit app, method, source, target, format, variable, and
final-state constraints as required. Do not satisfy a similar task
through a workaround and claim completion.
- Do not infer missing required parameters; verify them if possible,
otherwise use `INFEASIBLE`.
```

B.2 Planner Input Payload

The planner receives the current screenshot as multimodal image content together with a text payload that exposes the persistent control state. The schematic logical payload follows the order below; the listing illustrates the fields exposed to the planner rather than a literal static prompt template. Unavailable fields are left empty or marked as none. Application-specific skills and software API schemas are supplied separately as system-side guidance.

Schematic Planner Input Payload

```
Task instruction: {task_instruction}
Current subgoal: {current_subgoal}
Refined history: {refined_history}
Recent abstracted steps: {recent_steps}
Runtime feedback: {recovery_hint}
Foreground application: {foreground_app}
Current screenshot: [attached image content]
```

B.3 PLANNER RESPONSE FORMAT PROMPT

System Prompt: Planner Response Format

````
# Response Format
You may think through up to 3 candidate proposals before the final
answer, but the executable answer must be only the final JSON object.
List only the strongest plausible candidates. Do not invent weak
filler strategies just to reach 3 items.

Use this exact structure:
Candidates:
1. <one-sentence strategy>
[optional] 2. <one-sentence strategy>
[optional] 3. <one-sentence strategy>

Best:
<candidate number>

Why:
<one-sentence selection reason>

```json
{
   "thought": "Brief reasoning about the chosen strategy and current
   action. Check prerequisites and verify previous result.",
   "subgoal": "Meaningful phase-level objective for the current stage
   of work, or 'continue' to keep the current one",
   "actions": [
      "pyautogui.click(695, 514) # click the radio button next to
      Microsoft Bing",
      "pyautogui.click(663, 807) # click Set as default button"
   ]
}
```

Field guide:
- The final JSON object is the only executable output. Everything
before it is optional planning text.
- Output 1 to 3 candidates, not always 3.
- Every candidate must be realistic and worth considering for this
exact state.
- If one approach is clearly dominant, output just 1 candidate.
- If you list candidates, keep them short and plain text only. Do not
put JSON, code blocks, or extra headings inside candidate text.
- End your response with exactly one final JSON object.
- The final JSON object must contain `thought`, `subgoal`, and `
actions`.
- `subgoal`: a phase-level objective for a coherent stage, not an
action title or method. Avoid labels like "Open menu", "Click link", "
Scroll to inspect", or "Run script"; prefer "Navigate to the target
page", "Update the spreadsheet", or "Verify the result".
- Use `continue` while the next action still pursues, verifies, or
strengthens evidence for the current subgoal. Start a new subgoal only
 when the target state changes or the current stage is blocked.
- `actions` must be a non-empty ordered list. The environment changes
after each action, so plan them in execution order.
- Prefer one meaningful interaction per action item. Even though some
action strings can contain multiple low-level operations, you should
usually split sequential interactions into separate `actions` items
for clarity and better replanning.
- Each action item should be one of:
  - a bare `pyautogui...` string for GUI actions
  - a bare `Class.method(...)` string for API calls
````

```
  - a bare command/code string for `bash_execution`
  - the string `WAIT` for `wait`
  - the string `TERMINATE` only when the task is already verified
  complete, with the completion evidence explained in `thought`
  - the string `INFEASIBLE` only when the task is objectively
  impossible, with the blocker reason explained in `thought`
- Use exactly one API call per action item.
- Bash example:
```json
{
  "thought": "Locate the exact workbook before editing it.",
  "subgoal": "Locate the target workbook",
  "actions": [
    "find /home/user -type f -name '*.xlsx' 2>/dev/null | head -20"
  ]
}
```
- Good example:
Candidates:
1. Use a short GUI action sequence to open the search engine menu and
set Bing there.
2. Use one direct browser API call if an API exists.

Best:
1

Why:
The relevant setting is visible in the UI and there is no confirmed
api call for this browser preference.

```json
{
  "thought": "Open the visible browser setting and choose Bing through
  the interface.",
  "subgoal": "Set Bing as the default search engine",
  "actions": [
    "pyautogui.click(695, 514) # click the radio button next to
    Microsoft Bing",
    "pyautogui.click(663, 807) # click Set as default button"
  ]
}
```

- Termination example:
```json
{
  "thought": "The required result is already visible and verified in
  the current screenshot.",
  "subgoal": "Verify task completion",
  "actions": [
    "TERMINATE"
  ]
}
```

- Infeasible example:
```json
{
  "thought": "The requested result cannot be completed because the
  required app or capability is unavailable in the current environment
  .",
  "subgoal": "Confirm the blocker",
  "actions": [
```
```

```
    "INFEASIBLE"
  ]
}
```

B.4 Response Repair Prompt

System Prompt: Response Repair

```
Repair the planner response below and preserve the intended plan.

Repair rules:
- Focus on fixing the concrete parse/schema/tool-syntax error shown
above.
- If the error points to a specific line, column, or token, fix that
exact location.
- Action content must be a bare string, not an object.
- A GUI action must be a valid Python `pyautogui` code string.
- An API action must be exactly one Python call expression such as `
GoogleDriveTools.upload_file(...)`.
- A shell action must contain only raw shell/Python code. Remove any
leading `bash_execution` wrapper, function syntax, colon, or label.
- Return exactly one repaired response.
- The final output must satisfy this response format exactly:

{response_format_prompt}

Original response:
{response}

Parse traceback:
{error_message}
```

B.5 Step Abstraction Prompt

System Prompt: Step Abstraction

```
Compare the latest observations and summarize what happened in 1-2
concise sentences.

Task instruction: {task_instruction}
Current subgoal: {current_subgoal}
Action: {action_description}
Runtime feedback: {recovery_hint}

Rules:
- Describe what changed, or say no visible change.
- Mention clear errors if shown.
- Do not judge subgoal status or long-horizon task completion.
- Keep the summary concise and concrete.
- For bash/API output, preserve exact task-relevant identifiers from
the output, especially paths, filenames, URLs, IDs, sheet/table/column
 names, window titles, counts, and selected values.
- Use the task instruction and current subgoal to decide which output
details are key; do not replace needed paths or IDs with generic
wording.
```

B.6 Context Refinement Prompt

System Prompt: Context Refinement

```
Analyze task execution progress and provide guidance.

You will receive: a task instruction, execution history range and
execution history

Instructions:
- Summarize the full execution history into one unified summary
covering the entire range
- List what was done in order (successes and failures)
- **IMPORTANT**: Preserve coordinates in click actions (e.g., "click
(500,300)") so the summary records the exact prior interactions
- Identify repeated no-progress behavior, task-relevant progress, or
stalled execution.
- Provide actionable suggestions for the next step if there are issues

Return a concise summary string in this format:
Steps X~Y: [ordered list of what was done, keeping coordinates].
Suggestion: [actionable advice, or 'Continue' if progressing well]

Examples:
- Steps 1~5: Opened file, tried to edit (failed 3 times with
permission error), attempted sudo (failed). Suggestion: Try a
different approach - copy file to temp location first.
- Steps 1~5: Clicked Submit button at click(850,620), typed text,
clicked Save at click(920,580). Suggestion: Continue - forms being
filled correctly.
- Steps 1~10: Previously installed package and ran script (steps 1~5).
 Then verified output, tested functionality (steps 6~10). Suggestion:
Continue - good progress.
- Steps 1~15: Clicked the same button 5 times with no response, tried
alternative buttons (failed). Suggestion: Try an alternative method,
or use INFEASIBLE if the task is objectively impossible.
```

B.7 Final Verification Prompt

System Prompt: Final Verification

```
Decide whether the task is fully completed after the observed workflow
.

You will receive:
- the original task
- the current subgoal
- the provided execution history
- the initial screenshot from the start of the task
- the latest screenshot from the end of the task

Return exactly one word:
PASS
or
FAIL

Rules:
- Return only `PASS` or `FAIL`.
- Use `PASS` only if the task requirements appear fully satisfied in
the provided execution history and screenshots.
- Some tasks may be infeasible by design. If required capabilities,
variables, or app features were unavailable, return `FAIL`.
- If there is uncertainty, return `FAIL`.
```

```
- The screenshots are primary evidence for visible state. Use the
initial screenshot as baseline context and the latest screenshot as
the final state to judge.
- If the task outcome is only obvious by comparing before vs after,
explicitly use that comparison before deciding.
- If the latest screenshot does not directly show the requested result
, return `FAIL`.
- Use the provided execution history to confirm what was actually
modified, saved, read back, or verified. If logs only show that a
command or API call ran, but do not confirm the requested final result
, return `FAIL`.
- If a file was modified through `bash_execution` while the desktop
app may still display stale content, prefer explicit readback/
verification evidence from the logs plus the latest screenshot of the
reopened app state.
- Fail if the logs reveal formatting mistakes, header corruption,
partial coverage, or any mismatch with the task requirements.
- Do not treat an explanation of impossibility, a command launch, or a
 transient status/toast as completion for a task that asked for an
actual GUI, file, or configuration result.
- Require exact evidence for explicit app, method, source, target,
format, variable, and final-state constraints; return `FAIL` if the
workflow guessed, changed, or bypassed them.
- For multi-target tasks (`all`, `both`, `each`, `respectively`), fail
 unless every requested target is explicitly covered by the logs.
- For relative-date tasks, fail unless the exact resolved absolute
date is explicitly covered.
```

B.8 Ablation Prompt Variants

The component ablations preserve the full prompts except at the interface being removed. The configuration without a persistent subgoal replaces the reserved `continue` value with an explicit phase-level objective on every planner call. The configuration without candidate proposals removes the candidate list, `Best` index, and selection rationale from the response format. The configuration without a multi-action list replaces `actions` with a single `action` field. Removing Step Abstraction bypasses its semantic prompt and records deterministic raw execution evidence. Removing state-conditioned routing, stall and loop handling, or Final Verification changes runtime control without introducing another planner prompt.

The `w/o L2S` configuration uses a task-level planner interface with `thought` and one `action`. It retains the same GUI, Bash/Python, software API, and reserved control-action channels as Full. Its Step Abstraction prompt omits the current subgoal, and its Final Verification prompt omits the current subgoal from the supplied evidence. The `w/o S2L` configuration retains the full planner commitment interface but uses task-conditioned Step Abstraction without the current subgoal or runtime feedback. It removes state-conditioned routing, routing-dependent recovery feedback, stall and loop handling, and Final Verification, while retaining periodic Context Refinement without state-triggered routing.

C Control-Failure Annotation Protocol

This section presents the annotation protocol used for both GPT-5.6-sol annotation and human verification. Both settings follow the same evidence inputs, label definitions, and decision rules, ensuring consistent assignment of control-failure labels across trajectories.

Control-Failure Annotation Guide

C.1 Annotation Objective and Setup

Annotate unresolved cross-step control failures that remain at trajectory termination. Use the task instruction, complete execution logs, and available evaluator evidence to assess the full tra-

jectory. Problems resolved during subsequent execution are excluded. When evaluator evidence conflicts with the execution record, follow the execution record.

C.2 LABEL DEFINITIONS

The annotation set contains five control-failure labels, STAGE, ASSESS, LOOP, RECOVER, and STOP. Evaluate these labels independently, and record each applicable label at most once per trajectory.

- STAGE records failure to maintain the task stage or its remaining requirements. It applies when execution moves to an incorrect stage, advances before the current stage is satisfied, returns to an earlier stage without justification, or omits a requirement needed to complete the current stage.
- ASSESS records an incorrect assessment of the current execution state or action outcome. It applies when trajectory evidence indicates that an action failed or remained incomplete but the agent treats it as successful, or when a successful action is treated as failed.
- LOOP records repetition of a materially equivalent action after evidence that the previous attempt made no progress, without new information, a relevant change in execution conditions, or an adjustment that could alter the outcome.
- RECOVER records an ineffective response to a recognized failure or no-progress condition. It applies when a recovery, rollback, or alternative execution path does not address the observed problem and leaves execution unrecovered.
- STOP records termination without sufficient evidence that the task has been completed. It applies when required task conditions remain unverified or unsatisfied and execution does not resume.
- OTHER records a final unresolved failure that does not satisfy any of the five control-failure definitions.
- NONE records a trajectory with no unresolved failure at termination.
- ANY indicates that a trajectory contains at least one of the five control failures.

OTHER and NONE are mutually exclusive with the five control-failure labels and with each other.

C.3 DECISION PROCEDURE AND BOUNDARY RULES

1. Review the complete trajectory and identify failures that remain unresolved at termination.
2. Evaluate STAGE, ASSESS, LOOP, RECOVER, and STOP independently, and record every applicable label.
3. Mark ANY if at least one control-failure label applies. If none applies, record OTHER when an unresolved failure remains, or NONE when no unresolved failure remains.
4. Assign multiple control-failure labels when the trajectory independently satisfies multiple definitions.

Unless the execution record provides clear contrary evidence, an evaluator score of 0 indicates that a failure is present. If the trajectory does not provide clear evidence for any of the five control-failure labels, record OTHER. Do not record NONE.

For LOOP, an equivalent retry is labeled only when the previous attempt has explicit no-progress evidence. A retry that introduces new information, changes a relevant execution condition, or makes a corrective adjustment is not labeled as LOOP.

For RECOVER, the failure or no-progress condition must first be recognized. The label applies when the subsequent response does not address that condition.

ASSESS and STOP may co-occur when the agent incorrectly interprets the execution state and subsequently terminates without sufficient completion evidence. These labels describe observable cross-step control-failure manifestations. Upstream perception, grounding, tool, or environment errors may coexist with a control-failure label when the trajectory independently satisfies its definition.

C.4 HUMAN VERIFICATION

We validate the GPT-5.6-sol annotations on a stratified sample of 37 OSWorld tasks shared across all five methods, yielding 185 trajectories. The tasks are sampled without replacement across OSWorld domains using random seed 42.

Table 6: Inter-annotator agreement on 185 trajectories. A1 and A2 report label prevalence for the two independent human annotators. Agreement, F1, and Cohen's κ are computed separately for each label.

Label	A1 prev. (%)	A2 prev. (%)	Agreement ↑ (%)	F1 (%) ↑	κ ↑
STAGE	20.0	20.5	88.6	72.0	0.649
ASSESS	22.2	30.8	84.9	71.4	0.615
LOOP	25.4	22.2	92.4	84.1	0.792
RECOVER	28.1	18.4	87.0	72.1	0.641
STOP	25.4	23.8	87.6	74.7	0.665
OTHER	9.7	10.8	90.3	52.6	0.472
NONE	40.0	36.2	88.6	85.1	0.760
ANY	50.3	53.0	87.6	88.0	0.751

Table 7: Agreement between GPT-5.6-sol and the two independent human annotators on the same 185 trajectories. Precision, recall, F1, and Cohen's κ are computed separately with each annotator as the reference.

	Annotator 1				Annotator 2			
Label	**P (%) ↑**	**R (%) ↑**	**F1 (%) ↑**	**κ ↑**	**P (%) ↑**	**R (%) ↑**	**F1 (%) ↑**	**κ ↑**
STAGE	68.4	70.3	69.3	0.615	60.5	60.5	60.5	0.503
ASSESS	62.5	73.2	67.4	0.572	62.5	52.6	57.1	0.403
LOOP	88.4	80.9	84.4	0.795	74.4	78.0	76.2	0.692
RECOVER	64.1	96.2	76.9	0.652	39.7	91.2	55.4	0.400
STOP	61.3	80.9	69.7	0.574	48.4	68.2	56.6	0.399
OTHER	90.9	55.6	69.0	0.665	54.5	30.0	38.7	0.336
NONE	90.6	64.9	75.6	0.634	88.7	70.1	78.3	0.681
ANY	74.4	96.8	84.1	0.632	76.9	94.9	84.9	0.637

Table 8: Task-level incidence of control-failure labels on OSWorld under the fixed Qwen3.5-9B planner backbone. Each cell reports the percentage of all 369 tasks assigned the corresponding label.

Method	**STAGE ↓**	**ASSESS ↓**	**LOOP ↓**	**RECOVER ↓**	**STOP ↓**	**NONE ↑**	**OTHER ↓**	**ANY ↓**
HiSA	36.3%	19.2%	14.1%	28.2%	51.2%	31.2%	12.7%	56.1%
CoAct-1	10.3%	27.6%	27.4%	38.2%	47.2%	28.5%	3.8%	67.8%
GTA1	11.7%	23.0%	26.3%	31.4%	49.1%	32.2%	5.4%	62.3%
Agent S3	10.8%	23.6%	11.7%	17.3%	64.8%	28.2%	3.3%	68.6%
LocalLSTC	18.2%	21.4%	28.5%	33.6%	26.6%	42.5%	15.7%	41.7%

Two PhD-level AI researchers independently annotate all 185 trajectories using the same label definitions and decision procedure. Each annotator receives the task instruction, complete execution logs, and evaluator evidence, without access to method identity or GPT-5.6-sol labels. The two annotators assign identical complete label sets to 63.2% of trajectories, with a mean Jaccard overlap of 72.3%. For ANY, they achieve 87.6% agreement, 88.0% F1, and Cohen's $\kappa = 0.751$. Across the seven original labels, Cohen's κ ranges from 0.472 for OTHER to 0.792 for LOOP, with a macro average of 0.656. Table 6 reports the complete label-wise inter-annotator agreement.

We further compare the GPT-5.6-sol annotations with each human annotator on the same 185 trajectories. Table 7 reports precision, recall, F1, and Cohen's κ for each label and for ANY, using each annotator independently as the reference. For ANY, GPT-5.6-sol achieves 84.1% F1 with $\kappa = 0.632$ against Annotator 1 and 84.9% F1 with $\kappa = 0.637$ against Annotator 2. The corresponding human–human comparison reaches 88.0% F1 with $\kappa = 0.751$.

C.5 ANNOTATION RESULTS

Table 8 reports the complete task-level label distribution across all 369 OSWorld tasks under the fixed Qwen3.5-9B planner backbone. LocalLSTC has the lowest ANY incidence at 41.7% and the highest NONE incidence at 42.5%, compared with an average ANY incidence of 63.7% across the four baselines.

D WindowsAgentArena Runtime Diagnostics

Table 9 summarizes LocalLSTC's runtime behavior across WindowsAgentArena domains with Qwen3.5-9B and complements the OSWorld analysis in Table 2.

Table 9: Domain-level runtime behavior of LocalLSTC with Qwen3.5-9B on WindowsAgentArena, averaged over two runs. Execution-channel columns report action shares, with the remainder reserved for control actions. Pass, Fail, and Bypass are mutually exclusive trajectory categories based on Final Verification history. Bypass takes precedence for consecutive-stall or maximum-step cutoffs without an explicit `INFEASIBLE` action. Otherwise, Pass indicates no rejected Final Verification, while Fail indicates at least one, even if a later verification succeeds. An explicit `INFEASIBLE` action counts as an accepted Final Verification outcome, with earlier rejections still counted.

Summary			Commitment				Execution Channels			Evidence Control			
Domain	**Tasks**	**SR ↑ (%)**	**steps**	**Subgoals / Task**	**Actions / Subgoal**	**Actions / step**	**GUI (%)**	**Bash (%)**	**API (%)**	**Context Refines**	**Pass (%)**	**Fail (%)**	**Bypass (%)**
Chrome	17	34.7	24.1	8.1	3.3	1.1	84.5	0.9	5.3	2.0	35.3	23.5	41.2
Clock	4	37.5	34.5	10.5	5.5	1.7	96.0	0.0	0.0	3.4	50.0	0.0	50.0
File Explorer	19	63.2	18.5	7.8	3.0	1.3	73.4	14.4	0.0	1.6	50.0	28.9	21.1
LibreOffice Calc	24	8.3	34.7	13.8	4.5	1.8	69.3	1.1	23.3	3.9	12.5	16.7	70.8
LibreOffice Writer	19	7.9	24.4	11.2	3.1	1.4	79.4	0.2	7.3	2.4	18.4	7.9	73.7
Microsoft Paint	3	83.3	22.5	10.0	3.0	1.3	88.3	4.1	0.0	2.3	50.0	33.3	16.7
Microsoft Edge	13	50.0	19.3	8.5	2.5	1.1	88.7	1.8	0.0	1.9	57.7	19.2	23.1
Notepad	2	100.0	15.2	9.0	2.3	1.3	73.1	15.1	0.0	1.8	25.0	75.0	0.0
Settings	5	100.0	6.2	4.4	1.3	0.9	85.1	0.0	0.0	0.6	100.0	0.0	0.0
VLC	21	49.6	33.1	11.5	3.5	1.2	82.9	8.6	2.8	3.4	40.5	23.8	35.7
VS Code	24	62.5	21.8	8.9	3.5	1.4	88.9	4.2	0.9	2.4	62.5	10.4	27.1
Windows Calculator	3	66.7	41.2	8.7	8.2	1.7	91.1	5.9	0.0	2.8	33.3	66.7	0.0
All	154	43.1	25.4	10.0	3.6	1.4	80.1	4.0	8.1	2.6	40.9	19.2	39.9

Averaged over two runs, each task uses 25.4 planner steps and 10.0 subgoal segments, with 3.6 actions per subgoal and 1.4 actions per planner step. GUI actions account for 80.1% of execution, Bash accounts for 4.0%, and software APIs account for 8.1%. The execution mix varies across domains. Bash reaches 15.1% in Notepad and 14.4% in File Explorer, while software API use reaches 23.3% in LibreOffice Calc. GUI interaction remains the primary channel in every domain. LocalLSTC performs 2.6 context refinements per task. Pass, Fail, and Bypass account for 40.9%, 19.2%, and 39.9%, respectively.

E Ablation Configurations

All ablations use the same OSWorld setup, Qwen3.5-9B planner, GTA1-7B visual grounder, 100-step budget, and inference parameters as Full. Each reported variant changes only the listed configuration flag. All unlisted settings remain unchanged.

E.1 Grouped L2S–S2L Ablations

The `w/o L2S` configuration uses `--wo_l2s`. It removes subgoal-conditioned commitment formation and uses task-level single-action planning while retaining all execution channels, task-level post-execution control, and Step Abstraction. The `w/o S2L` configuration uses `--wo_s2l`. It retains the L2S commitment interface but replaces evidence-conditioned post-execution state updates with planner-intent subgoal updates. Table 10 specifies the resulting interfaces.

E.2 Component Ablations

Table 11 specifies the seven component ablations together with the behavior retained or substituted in each configuration. Each comparison measures the effect of removing the named component together with its dependent runtime behavior. Removing state-conditioned routing also bypasses recovery, stall and loop enforcement, and Final Verification, while removing stall and loop handling also disables error routing and its recovery feedback.

Table 10: Grouped L2S and S2L ablation configurations under the fixed Qwen3.5-9B backbone. Full denotes the complete LocalLSTC configuration.

Configuration	Full	w/o L2S	w/o S2L
Planning context	Persistent g_t, subgoal-aligned evidence, refined history, and recent abstracted steps	Task-conditioned evidence and history without a persistent subgoal in planner context	Planner retains a subgoal proposal, while evidence context is task-conditioned
Proposal and commitment	Up to three proposals compared within one inference, followed by an ordered action list	Single proposal with one action per planner step	Same as Full
Step Abstraction	Subgoal-conditioned evidence across execution channels	Retained with task-level conditioning	Retained with task-level conditioning
State update and routing	Evidence-conditioned `continue`, `advance`, `error`, `stall`, and `finish` routing	Task-level error, stall, and finish handling without subgoal-based stage transitions	Planner-intent state updates without S2L routing
Recovery feedback	State-conditioned error and stall feedback	Retained at the task level	Removed
Context Refinement	`advance`-triggered refinement with maximum-interval fallback	Periodic refinement without subgoal advance	Periodic refinement without state routing
Termination	Final Verification after `TERMINATE`, with rejection feedback returned to planning	Task-level Final Verification without subgoal context	Termination without Final Verification

Table 11: Component ablation settings under the fixed Qwen3.5-9B backbone.

Ablation	Removed behavior	Retained or replacement behavior
Persistent-subgoal interface	The planner does not receive the previous subgoal and cannot use `continue`.	Each call generates a new subgoal label. Candidate proposals, multi-action commitments, all execution channels, evidence, routing, refinement, and Final Verification remain enabled.
Candidate proposals	The planner omits the candidate list, `Best` index, and selection rationale.	The planner directly returns one final JSON decision. Persistent subgoals, ordered action lists, all execution channels, and S2L remain enabled.
Multi-action list	The final decision cannot contain an ordered `actions` list.	The response uses one `action` per planner call. Candidate proposals, persistent subgoals, all execution channels, and S2L remain enabled.
Step Abstraction	The semantic Step Abstraction model call is removed.	Deterministic raw evidence records the action and observed change for GUI execution. Bash and API evidence additionally records status, exit code, and raw output. Routing, refinement, and Final Verification remain enabled.
State-conditioned routing	Assigned `continue`, `advance`, `error`, `stall`, and `finish` states are logged but not applied to runtime control.	Step Abstraction and planner subgoal updates remain. Context Refinement becomes periodic. Routing-dependent recovery, stall and loop enforcement, and Final Verification are bypassed.
Stall and loop handling	Repeated-action checks and action–result loop checks are disabled. Execution outcomes are not remapped to `error` or `stall` from tool failure or repeated no-change evidence.	Planner subgoal transitions, Step Abstraction, Context Refinement, and Final Verification remain. Runtime recovery feedback for these removed events is not added to planner context.
Final Verification	The verifier call after `TERMINATE` and its rejection feedback are removed.	A planner `TERMINATE` request ends the trajectory directly. All planning, execution, abstraction, routing, refinement, and stall and loop handling before termination remain enabled.

F Runtime Control Examples

The runtime-event analysis in Table 4 examines repetition, recovery entry, and rejected termination under the Qwen3.5-9B planner backbone. The six successful OSWorld trajectories below illustrate the model behavior associated with these events and the control mechanisms that guide subsequent execution.

F.1 Recovery after Repeated Actions

Task ID `01b269ae-2111-4a07-81fd-3fcd711993b0`

Instruction. Fill all the blank cells in B1:E30 with the value in the cell above it. Finish the work and don't touch irrelevant regions, even if they are blank.

Model Behavior. The planner generates three equivalent file-editing commands that produce no visible spreadsheet change. File readback then localizes the unresolved cells to column B. The planner revises the command, reloads Calc, and requests termination before sufficient visual evidence of the target range is available.

Mechanism. The persistent subgoal retains the original range and the constraint against modifying irrelevant regions. Step Abstraction incorporates the no-progress and file-readback evidence. L2S revises execution around the unresolved cells instead of restarting the task. Final Verification rejects the premature termination. Subsequent execution refreshes the spreadsheet and obtains sufficient evidence for completion.

Task ID `4127319a-8b79-4410-b58a-7a151e15f3d7`

Instruction. Use terminal command to count all the lines of all php files in current directory recursively, show the result on the terminal

Model Behavior. The planner repeatedly uses the Bash channel, which returns the same 54-line result without producing the required visible terminal evidence. It subsequently switches to GUI execution, opens a terminal, and enters the counting command there.

Mechanism. The persistent subgoal retains both the recursive counting requirement and the requirement to show the result in the terminal. L2S changes the execution channel while preserving the task stage. Final Verification accepts termination after the visible terminal establishes both the command and its result.

F.2 Stage-Preserving Recovery across Execution Channels

Task ID `3c8f201a-009d-4bbe-8b65-a6f8b35bb57f`

Instruction. Download the image from https://huggingface.co/datasets/xlangai/ubuntu_osworld_file_cache/resolve/main/multi_apps/3c8f201a-009d-4bbe-8b65-a6f8b35bb57f/kingbird.jpeg, and then use GIMP to compress it to under 600KB as "compressed.jpeg" on the Desktop. Resize if needed.

Model Behavior. The planner completes the download through Bash and performs the export through the GUI, but repeated GUI attempts produce incorrect filenames. File readback then identifies the candidate files and their sizes, including a valid 115 KB compressed file.

Mechanism. The persistent state preserves the completed download while keeping the compression subgoal active. L2S switches execution back to Bash and normalizes the valid file to the required filename. The accumulated evidence establishes the required location, filename, and size without repeating completed stages.

Task ID `c867c42d-a52d-4a24-8ae3-f75d256b5618`

Instruction. Please assist me in exporting my contacts of Personal Address Book from Thunderbird into contacts.csv file in the desktop and convert it to .xlsx with Libreoffice Calc.

Model Behavior. The planner exports the address book through the GUI, producing `Personal Address Book.csv`. File readback reveals the filename mismatch. The planner then uses Bash to rename the file and completes the conversion in Calc.

Mechanism. S2L retains the successful export as completed evidence while keeping the conversion stage active. L2S repairs the filename through Bash without repeating the export and then continues the pending conversion. Bash readback confirms that `contacts.xlsx` exists on the desktop.

F.3 Evidence-Grounded Termination

Task ID `1e8df695-bd1b-45b3-b557-e7d599cf7597`

Instruction. Add a new column named "Profit" right next to the 'CGOS' column and calculate the profit for each week by subtracting "COGS" from "Sales" in that column.

Model Behavior. After initial numeric-type and file-state errors, the planner uses a corrected script that completes the edit. Readback confirms the column name and sample calculations. The planner then requests termination before visual confirmation is available.

Mechanism. Final Verification rejects the first termination request because the accumulated evidence does not yet establish the visible spreadsheet state. L2S retains the active task stage, resolves the Calc recovery dialog, and verifies the Profit column and its values. Final Verification accepts the subsequent request after the required visual evidence becomes available.

Task ID `6ada715d-3aae-4a32-a6a7-429b2e43fb93`

Instruction. Copy the screenshot 1.png from the desktop to where my cursor is located

Model Behavior. The planner requests termination before the available evidence establishes the insertion result. It then inspects the inserted image and its document position. A subsequent termination request occurs while a context menu obscures the relevant content.

Mechanism. Final Verification rejects both requests because the available evidence is insufficient to establish the required document state. The persistent image-insertion subgoal keeps execution focused on the unresolved evidence requirement. Subsequent interaction exposes the inserted image and its position, after which Final Verification accepts completion.