

LOCALLSTC: A LONG SHORT-TERM CONTROL ARCHITECTURE FOR LOCALLY DEPLOYED GUI AGENTS

Weiming Li, Helen Paik & Yulei Sui*

School of Computer Science and Engineering
The University of New South Wales
Sydney, Australia
{weiming.li1,h.paik,y.sui}@unsw.edu.au

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 37.7%. Trajectory annotation further identifies at least one control failure in 91.6% 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. Long-Term Control maintains the active subgoal, subgoal-aligned evidence, and runtime feedback across interactions, while Short-Term Execution realizes bounded commitments for the current step. Long-to-Short Planning forms each commitment from persistent state, and Short-to-Long Control integrates execution outcomes back into that state for progress assessment, recovery, and termination. With Qwen3.6-27B, LocalLSTC reaches 64.7% SR-100 on OSWorld and 65.3% on WindowsAgentArena, outperforming the strongest prior local results on both benchmarks. Ablations further support contributions from mechanisms on both sides of execution. These findings identify temporal organization of control information as a distinct architectural dimension for 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 achieve strong desktop task performance through hierarchical planning, specialized components, code execution, and test-time search (Agashe et al., 2025a;b; Gonzalez-Pumariega et al., 2025; Yang et al., 2026b; Song et al., 2026). Yet persistent control information often remains implicit in the interaction trajectory. At each new decision, the planner needs to reconstruct the active task stage, accumulated evidence, and relevant runtime feedback before choosing the next action.

Local deployment changes this assumption. Keeping inference local allows screenshots, files, credentials, browser state, and application data to remain within the local deployment boundary. It also reduces dependence on external-service latency, availability, and recurring invocation costs (Liu et al., 2024; Xue et al., 2024). Public benchmarks and open-model studies continue to report a 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; Google DeepMind, 2026), while local hardware capacity, compute, and inference

*Corresponding author: y.sui@unsw.edu.au

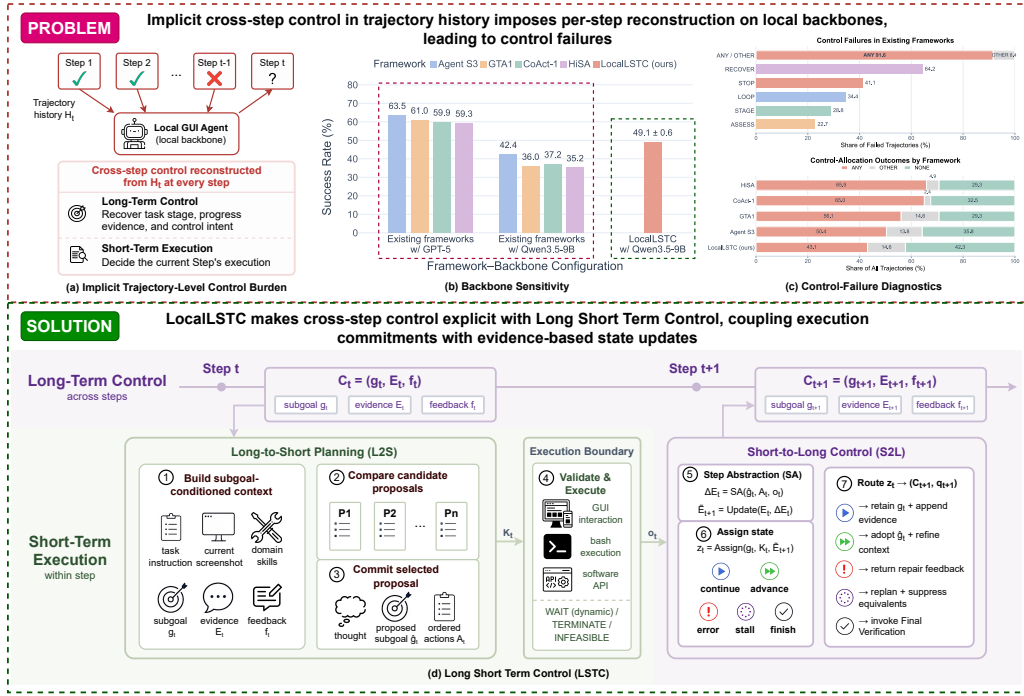


Figure 1: Overview of the control problem, empirical evidence, and LocalLSTC. Panel (a) illustrates the implicit trajectory-level control burden. Panel (b) shows the SR-100 drop after replacing GPT-5 with Qwen3.5-9B across four GUI-agent frameworks. Panel (c) reports cross-step control-failure incidence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

throughput still constrain deployable model scale (Liu et al., 2024; Xue et al., 2024). Existing approaches mainly rely on more complex system composition or GUI-specific training that changes model parameters (Agashe et al., 2025b; Song et al., 2026; Qin et al., 2025; Xu et al., 2025). Runtime interfaces for adapting off-the-shelf general backbones to open-ended desktop tasks remain less systematically studied (Xu et al., 2026b).

Long-horizon GUI control requires information to persist beyond a single decision. Existing trajectory-conditioned interfaces leave the active task stage, accumulated evidence, and runtime feedback embedded in a growing interaction history. At each step, the backbone reconstructs this cross-step state before determining how to proceed. We call this repeated reconstruction the *implicit trajectory-level control burden*. Panel (a) of Figure 1 depicts this burden. As trajectories grow, reconstruction errors can manifest as loss of stage commitment, incorrect progress assessment, repeated equivalent actions after no-progress evidence, mismatched recovery, or premature termination without sufficient completion evidence. We refer to these observable manifestations as *control failures*.

Controlled backbone replacement exposes this sensitivity. Across four representative state-of-the-art frameworks, replacing GPT-5 with local Qwen3.5-9B reduces average OSWorld SR-100 from 60.9% to 37.7%. Among failed trajectories from these local-backbone baselines, 91.6% contain at least one control failure. Under the same Qwen3.5-9B backbone, LocalLSTC reaches 49.1% SR-100 and reduces ANY control-failure incidence from the 59.4% baseline average to 43.1%. Panels (b) and (c) of Figure 1 summarize the backbone replacement and control-failure diagnostics. These results motivate explicit cross-step control state under constrained local reasoning backbones.

To address this problem, we introduce LocalLSTC, a Long Short-Term Control Architecture for Locally Deployed GUI Agents. LocalLSTC separates control by temporal scope. Long-Term Control preserves the active subgoal, subgoal-aligned evidence, and runtime feedback across interactions. Short-Term Execution represents a bounded commitment for the current step. This separation keeps

persistent control information available across decisions while limiting each execution commitment to the immediate interface state.

Long-to-Short Planning (L2S) converts the persistent state and current observation into a subgoal-conditioned execution commitment. Short-to-Long Control (S2L) then abstracts the execution outcome into subgoal-aligned evidence and updates the persistent state for subsequent progress assessment, recovery, and termination. The resulting L2S–S2L loop carries the active subgoal, accumulated evidence, and runtime feedback across interactions without requiring each planner call to recover them from the full trajectory.

With Qwen3.6-27B, LocalLSTC reaches 64.7% SR-100 on OSWorld and 65.3% on WindowsAgentArena, achieving the strongest literature-reported local results in our comparison on both benchmarks. Under the 50-step WindowsAgentArena setting, it reaches 63.4%, improving the previous local result by 18.1 percentage points and coming within 0.1 points of the strongest reported API result. With Qwen3.5-9B fixed, removing L2S or S2L reduces OSWorld SR-100 from 49.1% to 36.4% and 41.5%, respectively, supporting contributions from mechanisms on both sides of execution. Our contributions are as follows.

- We quantify 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 separates persistent control from step-local execution by temporal scope. Long-Term Control maintains cross-step state, Short-Term Execution realizes bounded commitments, and L2S and S2L connect the two scopes across execution.
- LocalLSTC achieves the strongest local results on OSWorld and WindowsAgentArena. Fixed-backbone analysis and ablations further isolate contributions from mechanisms on both sides of execution.

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 methods encode GUI-specific capabilities into model parameters. LocalLSTC instead operates with an off-the-shelf backbone and focuses on how task stage, execution evidence, and runtime feedback are maintained across interactions.

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-Pumariiega et al., 2025; 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 frameworks improve agent capability through richer system composition, execution interfaces, or additional inference-time computation. LocalLSTC instead focuses on how persistent control information is organized across interactions around a fixed backbone.

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). Unlike these approaches, LocalLSTC treats the temporal organization of control information across interactions as the primary architectural abstraction.

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 ,

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

where x_t contains the task instruction and current observation, H_t is the interaction trajectory accumulated through step t , a_t is the action selected at the current step, and o_t is the resulting execution outcome. 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*

$$C_t = (g_t, E_t, f_t), \quad C_0 = (\perp, \emptyset, \perp),$$

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. The commitment expires after its execution outcomes have been incorporated into the next control state.

To connect these two temporal scopes across execution, LocalLSTC uses two complementary mechanisms. Long-to-Short Planning (L2S) forms a Short-Term Execution Commitment from the persistent control state and current observation. Short-to-Long Control (S2L) incorporates the execution outcome into the next Long-Term Control State. The interaction between the two temporal scopes is

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

where $q_{t+1} \in \{\text{running}, \text{success}, \text{failure}\}$ denotes task status after the state update.

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. Step 1 builds a subgoal-conditioned context from the task instruction, current observation, domain skills, and persistent state C_t . Step 2 compares candidate proposals under this shared context, and Step 3 commits the selected proposal as $K_t = (\hat{g}_t, A_t)$. Step 4 validates and executes the ordered actions in K_t , producing the execution outcome o_t . S2L then updates persistent control from this outcome. Step 5 applies Step Abstraction to convert o_t into subgoal-aligned evidence, Step 6 assigns a transient routing state according to the resulting execution evidence, and Step 7 routes this state into C_{t+1} and task status q_{t+1} . The updated Long-Term Control State then conditions the next interaction.

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 according to the active subgoal so that operational guidance remains aligned with the current task stage. Appendix D.1 details prompt construction.

In step 2, the planner evaluates alternative proposals before committing to the next execution. A common inference-time approach improves selection by sampling candidates across multiple planner calls and ranking them with a separate judge, as in GTA1 (Yang et al., 2026b). L2S instead performs proposal comparison within a single planner inference, with all candidates conditioned on the same persistent control state.

In step 3, the selected proposal is serialized as a JSON object containing `thought`, `subgoal`, and `actions`. The proposal specifies the execution subgoal $\hat{g}_t$, which may retain g_t or propose 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. LocalLSTC adopts programmatic Bash execution from CoAct-1 (Song et al., 2026), while its software API follows a schema-constrained interface similar to ComputerRL (Lai et al., 2026). Appendices D.2 and D.3 specify the action space and validation rules.

3.2.2 EXECUTION

Step 4 validates $K_t = (\hat{g}_t, A_t)$ before execution. The runtime checks the final JSON object, action schemas, and channel-specific parameters, then dispatches valid actions in list order through the corresponding GUI, Bash, or software API executor. Pre-execution fingerprint checks suppress repeated equivalent actions when the current state provides no basis for repetition. Reserved actions `WAIT`, `TERMINATE`, and `INFEASIBLE` are handled directly by the runtime. The resulting outcomes o_t , together with K_t , are passed to S2L. Appendices D.3 and D specify the validation and execution rules.

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

$$\Delta E_t = \text{SA}(\hat{g}_t, A_t, o_t), \quad \tilde{E}_{t+1} = \text{Update}(E_t, \Delta E_t).$$

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 D.4 details the channel-specific observation and abstraction mechanisms.

In step 6, the runtime assigns a transient routing state

$$z_t = \text{Assign}(g_t, K_t, \tilde{E}_{t+1}),$$

where $z_t \in \{\text{continue}, \text{advance}, \text{error}, \text{stall}, \text{finish}\}$. Assignment follows a fixed precedence. A `TERMINATE` request maps to `finish`. Execution failure maps to `error`, repeated no-progress maps to `stall`, and a changed execution subgoal maps to `advance`. All remaining cases map to `continue`. Appendix D.5 specifies the detection rules and thresholds.

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

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

The `continue` route retains the current subgoal and appends new evidence. The `advance` route adopts the proposed subgoal after execution provides supporting evidence and can trigger context

refinement, which compresses completed evidence into compact history while retaining recent evidence for subsequent planning and verification. LocalLSTC inherits this refinement mechanism from HiSA (Li et al., 2026b). The `error` route writes execution-specific feedback into the persistent state, while `stall` records no-progress or rejected-termination feedback for the next planner call. The `finish` route invokes Final Verification, which evaluates task completion against accumulated visual, textual, and software API evidence. A `PASS` sets task status to `success`, while a `FAIL` records the rejected termination, returns `stall` feedback, and resumes planning with status `running`. The `INFEASIBLE` action and predefined runtime failure boundaries set task status to `failure`.

Step Abstraction and Final Verification use the planner backbone under dedicated prompts, while state assignment and routing are deterministic runtime operations. Appendices D.6, D.7, and D provide the implementation details.

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 isolate the contributions of L2S, S2L, and their constituent mechanisms, with runtime-event analysis examining how these mechanisms affect recovery after execution problems.

4.1 EXPERIMENTAL SETUP

Benchmarks. We evaluate LocalLSTC on OSWorld (Xie et al., 2024) and WindowsAgentArena (Bonatti et al., 2025). OSWorld contains 369 Linux tasks spanning single applications, file operations, and cross-application workflows. Backbone-sensitivity experiments, control-failure annotation, fixed-backbone ablations, and runtime-event analyses are conducted on OSWorld, together with the main benchmark evaluation. WindowsAgentArena contains 154 Windows tasks covering applications, system utilities, and browsers, and is used to evaluate performance in a distinct operating-system and application 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 and GTA1-7B (Yang et al., 2026b) as the visual grounder. Additional experiments evaluate Qwen3.5-4B, Qwen3.6-27B, Gemma-4-E4B-it, and Gemma-4-26B-A4B-it (Qwen Team, 2026a;b; Google DeepMind, 2026). The planner backbone is reused for SA, context refinement, and Final Verification under dedicated prompts. OSWorld backbone configurations are run independently twice with the same parameter settings while other results use one run. Appendix D.8 lists runtime parameters, Appendix D.9 specifies model roles and token accounting, and Appendix E 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.

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 and statistics. 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. Task-level 95% confidence intervals are estimated with a percentile bootstrap using 10,000 resamples and random seed 2027 (Efron, 1979).

4.2 RESULTS

4.2.1 BACKBONE SENSITIVITY AND CROSS-STEP CONTROL FAILURES

Existing GUI-agent frameworks show a consistent sensitivity to reasoning-backbone strength. We evaluate this sensitivity on Agent S3, GTA1, CoAct-1, and HiSA through controlled reasoning-backbone replacement, substituting GPT-5 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 decreases by 21.1–25.0 points across the four frameworks, lowering their average from 60.9% to 37.7%. The consistency of this drop across different system compositions and inference-time strategies indicates that planner-backbone sensitivity is not specific to a single framework design. Under the same backbone, LocalLSTC reaches 49.1% SR-100, 11.4 points above the four-framework average.

Trajectory annotation further localizes this sensitivity to cross-step control behavior. Among failed trajectories from the four baselines, 91.6% contain at least one control failure. RECOVER, STOP, and LOOP are the three most frequent categories, occurring in 64.2%, 41.1%, and 34.4% of failed trajectories, respectively. The dominant category varies across frameworks, indicating that the aggregate burden manifests through multiple failure modes.

Human verification confirms the reliability of the aggregate annotation. The two annotators agree on 86.5% of ANY labels, with Cohen’s $\kappa = 0.730$. Against the two annotators, GPT-5.6-sol achieves ANY F1 scores of 86.1% and 85.2%, with $\kappa = 0.686$ and 0.647, respectively. Appendix A.4 reports the complete validation results.

Across all OSWorld tasks, LocalLSTC reduces ANY control-failure incidence to 43.1%, compared with a 59.4% average across the four baselines. The 16.3-point reduction has a task-level bootstrap 95% CI of [8.3, 24.2]. LocalLSTC also achieves the highest NONE incidence at 42.3%. Appendix A.5 reports the complete label distribution.

4.2.2 BENCHMARK PERFORMANCE

On OSWorld, LocalLSTC with Qwen3.6-27B and GTA1-7B reaches 64.7% SR-100, 62.7% SR-50, and 51.4% SR-15, achieving the strongest local results at all three step budgets. These results improve the previous strongest local scores by 10.8, 6.0, and 5.1 points, respectively. The advantage persists as the step budget decreases, indicating that the gain does not depend on long trajectories or a large interaction budget.

On WindowsAgentArena, the same configuration reaches 65.3% SR-100, 63.4% SR-50, and 48.5% SR-15, achieving the strongest local results on the benchmark. SR-50 improves the previous strongest local result by 18.1 points and matches the strongest API result within 0.1 points. The strong results across Ubuntu and Windows environments indicate that the control architecture transfers across distinct application ecosystems without benchmark-specific training.

LocalLSTC also benefits consistently from stronger local reasoning backbones. Within both the Qwen3.5 and Gemma-4 families, increasing backbone scale improves SR at every evaluated step budget on both benchmarks. This scaling pattern places temporal control organization and backbone capability on complementary axes.

Table 2 examines how the control structure is used across OSWorld domains with Qwen3.5-9B. Across the two runs, LocalLSTC averages 28.1 planner steps and 10.1 subgoal segments per task, with 3.7 actions per subgoal and 1.3 actions per planner step. The multi-action interface therefore supports short bounded commitments while each persistent subgoal spans multiple interactions. Execution channels also adapt to task structure. GUI actions account for 72.7% of execution overall, while Bash accounts for 15.1% and software APIs for 4.6%. OS tasks use Bash for 52.5% of actions, whereas Chrome and GIMP use GUI interaction for 86.1% and 85.5%, respectively.

Table 1: Success rates on OSWorld and WindowsAgentArena under step budgets of 100, 50, and 15. OSWorld results for LocalLSTC report mean $\pm$ SD over two runs, and WindowsAgentArena results report one run. 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 $\uparrow$	SR-50 $\uparrow$	SR-15 $\uparrow$	SR-100 $\uparrow$	SR-50 $\uparrow$	SR-15 $\uparrow$
API	o3 (OpenAI, 2025b)	23.0	17.2	9.1	-	-	-
	Agent S / GPT-4o (Agashe et al., 2025a)	-	-	20.6	-	-	18.2
	computer-use-preview (OpenAI, 2025a)	30.5	31.3	26.0	-	-	-
	Claude 3.7 Sonnet (Anthropic, 2025a)	35.6	35.8	27.1	-	-	-
	Seed1.5-VL / Doubao-1.5-thinking-vision-pro (Guo et al., 2025a)	40.0	-	31.9	-	-	-
	Claude Sonnet 4 (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., 2025b)	61.9	56.9	-	-	-	-
	Claude Sonnet 4.5 (Anthropic, 2025c)	62.9	58.1	42.9	-	-	-
	GTA1 / GPT-5 (Yang et al., 2026b)	63.4	-	-	-	-	-
	Agent S3 / GPT-5 (Gonzalez-Pumariaga et al., 2025)	63.5	-	-	50.2	49.0	-
	Agent S3 / Opus 4.5 bBoN(N=1)	66.0	-	-	-	-	-
	(Gonzalez-Pumariaga et al., 2025)	67.1	<u>64.4</u>	-	-	-	-
	UiPath Screen Agent / Opus 4.5 (Cristescu et al., 2025)	69.9	-	-	56.6	54.1	-
	Agent S3 / GPT-5 bBoN(N=10)	-	-	-	-	-	-
	(Gonzalez-Pumariaga et al., 2025)	-	-	-	<u>61.0</u>	60.4	-
	VLAA-GUI / Gemini 3 Flash (Han et al., 2026)	-	-	-	-	63.5	-
LOCAL	OS-Symphony / GPT-5 (Yang et al., 2026a)	-	65.8	-	-	-	-
	Claude Sonnet 4.6 (Anthropic, 2025d)	72.1	-	-	-	-	-
	Agent S3 / Opus 4.5 + GPT-5 bBoN(N=10)	<u>72.6</u>	-	-	-	-	-
	(Gonzalez-Pumariaga et al., 2025)	76.3	-	-	-	-	-
	VLAA-GUI / Opus 4.5 (Han et al., 2026)	-	-	-	-	-	-
	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 (Du 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	<u>46.3</u>	-	-	-
Ours	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	-	-	-	-
	LocalLSTC / Gemma-4-E4B-it + GTA1-7B	28.1 $\pm$ 2.1	26.9 $\pm$ 2.7	20.3 $\pm$ 0.6	26.1	26.1	19.6
	LocalLSTC / Qwen3.5-4B + GTA1-7B	37.3 $\pm$ 4.8	35.5 $\pm$ 3.3	24.7 $\pm$ 1.2	30.4	23.9	21.7
	LocalLSTC / Qwen3.5-9B + GTA1-7B	49.1 $\pm$ 0.6	46.2 $\pm$ 1.1	34.8 $\pm$ 2.2	42.9	35.8	25.4
Ours	LocalLSTC / Gemma-4-26B-A4B-it + GTA1-7B	60.3 $\pm$ 0.2	54.7 $\pm$ 2.4	42.2 $\pm$ 1.4	41.3	34.8	<u>28.3</u>
	LocalLSTC / Qwen3.6-27B + GTA1-7B	64.7 $\pm$ 0.6	62.7 $\pm$ 0.4	51.4 $\pm$ 1.9	65.3	<u>63.4</u>	48.5

Table 2: Domain-level runtime behavior of LocalLSTC with Qwen3.5-9B on OSWorld, averaged over two runs. Execution-channel columns report action shares. Pass, Fail, and Bypass denote accepted termination, rejected termination followed by recovery, and cutoff without verification. Recovery statistics are conditional on recovery entry.

Summary			Commitment			Execution Channels				Evidence Control					
Domain	Tasks	SR (%)	steps	Planning Structure			Channel Use			Verification				Recovery	
				Subgoals / Task	Actions / Subgoal	Actions / step	GUI (%)	Bash (%)	API (%)	Context Refines	Pass (%)	Fail (%)	Bypass (%)	SR after recovery (%)	steps after recovery entry
Chrome	46	57.0	23.9	8.2	3.4	1.1	86.1	1.8	3.1	2.3	58.2	17.6	24.2	61.0	2.1
GIMP	26	81.2	12.0	4.5	3.7	1.4	85.5	5.9	0.0	1.0	75.5	3.9	20.7	50.0	8.8
LibreOffice Calc	47	44.1	28.5	10.6	3.9	1.5	74.4	8.5	9.3	2.9	39.0	24.1	36.9	87.5	6.9
LibreOffice Impress	47	30.9	23.8	8.9	4.0	1.5	74.5	1.2	14.1	2.3	44.3	16.8	39.0	45.4	5.3
LibreOffice Writer	23	47.5	31.5	10.8	4.3	1.5	80.8	6.8	4.4	3.0	43.2	25.2	31.7	65.0	3.9
Multi-apps	101	32.7	38.1	14.0	3.4	1.3	67.2	25.2	1.5	4.1	52.3	19.7	28.1	37.2	5.5
OS	24	72.9	24.0	6.4	4.4	1.1	40.9	52.5	0.0	2.2	66.7	16.7	16.7	100.0	9.5
Thunderbird	15	60.0	25.5	6.9	4.3	1.2	87.6	4.3	0.0	2.1	70.0	3.4	26.7	50.0	0.5
VLC	17	62.1	31.7	11.4	3.2	1.1	71.7	14.9	3.4	3.4	68.7	14.3	17.2	74.7	11.5
VS Code	23	77.7	19.5	8.1	3.3	1.4	69.9	20.4	0.2	1.8	38.2	29.5	32.3	85.7	1.6
All	369	49.1	28.1	10.1	3.7	1.3	72.7	15.1	4.6	2.8	53.0	18.3	28.8	63.8	6.0

The persistent-state mechanisms are exercised repeatedly during successful execution. LocalLSTC performs 2.8 context refinements per OSWorld task. Final Verification rejects termination on 18.3%

Table 3: Execution-boundary 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 $\uparrow$	Tokens (k) $\downarrow$	Prompt (k) $\downarrow$	Completion (k) $\downarrow$	Steps $\downarrow$	Time (s) $\downarrow$
Full	LocalLSTC	49.1	256.3	240.6	15.7	28.1	403.2
BOUNDARY	w/o L2S	36.4	192.9	179.1	13.8	31.4	382.2
	w/o S2L	41.5	246.9	229.4	17.4	27.1	473.6
LONG-TERM CONTROL	w/o persistent subgoal	38.0	276.6	257.3	19.3	29.6	554.4
	w/o Step Abstraction	31.6	270.4	251.9	18.5	33.0	550.8
	w/o state-conditioned routing	40.7	215.2	199.0	16.2	25.4	431.3
	w/o stall/loop handling	45.1	285.0	266.9	18.1	29.8	517.2
	w/o Final Verification	40.3	268.7	251.7	16.9	28.7	510.7
SHORT-TERM EXECUTION	w/o candidate proposals	40.7	273.0	259.1	13.9	31.2	412.6
	w/o multi-action list	43.5	260.7	243.0	17.8	28.8	445.5

Table 4: Runtime-event diagnostics for Full, w/o L2S, and w/o S2L on OSWorld. Rate reports event incidence, and Score reports mean evaluator score conditional on entering the corresponding event. Bold marks the column maximum, and “–” indicates an event unavailable after removing the corresponding mechanism.

Method	Stall / repetition		Recovery entry		Rejected termination	
	Rate (%)	Score (%) $\uparrow$	Rate (%)	Score (%) $\uparrow$	Rate (%)	Score (%) $\uparrow$
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	–	–	–	–

of tasks, and 63.8% of those tasks subsequently succeed after returning to execution. The same behavior appears on WindowsAgentArena, where verification enters recovery on 21.4% of tasks and 51.5% of those tasks subsequently succeed after an average of 5.2 additional steps. Appendix C reports the complete WindowsAgentArena runtime statistics.

4.2.3 ABLATIONS AND MECHANISM ANALYSIS

We next examine which mechanisms account for LocalLSTC’s gains under a fixed Qwen3.5-9B backbone. We first ablate L2S and S2L as grouped mechanisms, then remove individual components within Long-Term Control and Short-Term Execution. Appendix B specifies the exact w/o L2S and w/o S2L configurations.

Removing L2S lowers SR-100 from 49.1% to 36.4%, and S2L to 41.5%. Both sides of the execution boundary contribute under the same Qwen3.5-9B backbone. Among individual mechanisms, SA produces the largest effect, reducing SR-100 by 17.5 points. Removing the persistent subgoal reduces SR by 11.1 points, while Final Verification, state-conditioned routing, candidate proposals, the multi-action list, and stall/loop handling contribute 8.8, 8.4, 8.4, 5.6, and 4.0 points, respectively. The largest effects occur in mechanisms that maintain or interpret cross-step evidence, consistent with the identified control burden.

Runtime-event diagnostics separate 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. The difference reflects how the agent responds after encountering no-progress behavior.

The pattern extends to recovery and termination. After recovery entry, Full reaches 34.0% compared with 20.9% for w/o L2S. After rejected termination, the scores are 38.9% and 23.1%. L2S supplies the stage and evidence context for the next commitment, while S2L converts execution outcomes into recovery state. The event-conditioned results connect the grouped ablations to the runtime behavior producing the final SR differences. Appendix F provides trajectory-level examples.

5 CONCLUSION

LocalLSTC addresses the cross-step control burden in locally deployed GUI agents by separating persistent Long-Term Control from Short-Term Execution. L2S forms execution commitments from persistent state and current observations, while S2L incorporates outcomes into state for subsequent

planning, recovery, and termination. LocalLSTC improves task success and reduces control failures; ablations confirm contributions from both sides of the execution boundary. With Qwen3.6-27B, LocalLSTC reaches 64.7% SR-100 on OSWorld and 65.3% on WindowsAgentArena, the strongest local results on both benchmarks. These results establish temporal control organization as an architectural dimension for locally deployed GUI agents.

REPRODUCIBILITY STATEMENT

Sections 3 and 4 describe the framework and experimental protocol. Appendix A details the control-failure annotation and human-verification procedures, and Appendix A.5 provides the corresponding task-level bootstrap analysis. Appendix D documents implementation details, while Appendix E includes the prompts for planning, repair, Step Abstraction, verification, and context refinement. The implementation, task-level outputs, and evaluation artifacts are released through an anonymous code repository for reproducibility.

AI USE STATEMENT

GPT-5.6-sol was used for post-hoc trajectory annotation. OpenAI Codex was used for language editing and L^AT_EX revision. All AI-assisted analyses and revisions were reviewed by the authors, who take responsibility for 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, 2025d. URL <https://www.anthropic.com/news/claude-sonnet-4-6>.
- Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibao 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>.
- Rogério 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>.

-
- Angang Du, Bohong Yin, Bowei Xing, Bowen Qu, Bowen Wang, Cheng Chen, Chenlin Zhang, Chenzhuang Du, Chu Wei, Congcong Wang, 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>.
- Bradley Efron. Bootstrap methods: Another look at the jackknife. *The Annals of Statistics*, 7(1): 1–26, 1979. doi: 10.1214/aos/1176344552. URL <https://doi.org/10.1214/aos/1176344552>.
- 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-Pumariaga, Vincent Tu, Chih-Lun Lee, Jiachen Yang, Ang Li, and Xin Eric Wang. The unreasonable effectiveness of scaling agents for computer use. *arXiv preprint arXiv:2510.02250*, 2025.
- Google DeepMind. Gemma 4. <https://deepmind.google/models/gemma/gemma-4/>, 2026. Accessed: 2026-05-30.
- Dong Guo, Faming Wu, Feida Zhu, Fuxing Leng, Guang Shi, Haobin Chen, Haoqi Fan, Jian Wang, Jianyu Jiang, Jiawei Wang, et al. Seed1.5-VL technical report. *arXiv preprint arXiv:2505.07062*, 2025a. doi: 10.48550/arXiv.2505.07062. URL <https://arxiv.org/abs/2505.07062>.
- 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*, 2025b. 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>.
- 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.
- 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>.
- 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 Piazzentin 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, Juntong 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, Chen Henry Wu, Zhennan Shen, Zhuokai Li, Ryan Li, Xiaochuan Li, Junda Chen, Boyuan Zheng, Peihang Li, Fangyu Lei, Ruisheng Cao, Yeqiao Fu, Dongchan Shin, Martin Shin, Jiarui Hu, Yuyan Wang, Jixuan Chen, Yuxiao Ye, Danyang Zhang, Yipu Wang, Heng Wang, Diyi Yang, Victor Zhong, Y. Charles, Zhilin Yang, and Tao Yu. OpenCUA: Open foundations for computer-use agents. In *Advances in Neural Information Processing Systems*, volume 38, 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://mlanthology.org/neurips/2024/xie2024neurips-osworld/>.
- 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. Aguis: 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>.

APPENDIX

A 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

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

A.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.

A.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.

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.

Table 5: 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 $\uparrow$ (%)	F1 (%) $\uparrow$	κ $\uparrow$
STAGE	20.0	20.5	88.6	72.0	0.649
ASSESS	22.2	31.9	83.8	70.0	0.594
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	9.2	8.6	88.6	36.4	0.301
NONE	40.5	37.3	87.0	83.3	0.727
ANY	50.3	54.1	86.5	87.0	0.730

Table 6: 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.

Label	Annotator 1				Annotator 2			
	P (%) $\uparrow$	R (%) $\uparrow$	F1 (%) $\uparrow$	κ $\uparrow$	P (%) $\uparrow$	R (%) $\uparrow$	F1 (%) $\uparrow$	κ $\uparrow$
STAGE	68.4	70.3	69.3	0.615	60.5	60.5	60.5	0.503
ASSESS	69.8	73.2	71.4	0.630	67.4	49.2	56.9	0.410
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	66.7	80.9	73.1	0.627	52.6	68.2	59.4	0.445
OTHER	63.6	82.4	71.8	0.685	31.8	43.8	36.8	0.298
NONE	93.6	58.7	72.1	0.595	93.6	63.8	75.9	0.654
ANY	77.6	96.8	86.1	0.686	79.3	92.0	85.2	0.647
Macro	74.0	79.9	75.7	0.661	62.4	68.3	63.3	0.506

A.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, with 10 multi-application tasks and 27 tasks from the nine single-domain strata.

To prevent method identity from influencing annotation, we recode each trajectory and remove identifying fields, including method names and result directories. 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 61.1% of trajectories, with a mean Jaccard overlap of 70.1%. For ANY, they achieve 86.5% agreement, 87.0% F1, and Cohen’s $\kappa = 0.730$. Across the seven original labels, Cohen’s κ ranges from 0.301 for OTHER to 0.792 for LOOP, with a macro average of 0.624. Table 5 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 6 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 86.1% F1 with $\kappa = 0.686$ against Annotator 1 and 85.2% F1 with $\kappa = 0.647$ against Annotator 2. The corresponding human–human comparison reaches 87.0% F1 with $\kappa = 0.730$.

A.5 COMPLETE ANNOTATION RESULTS

Table 7 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 43.1% and the

Table 7: 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. Bold marks the highest NONE incidence and the lowest ANY incidence.

Method	STAGE↓	ASSESS↓	LOOP↓	RECOVER↓	STOP↓	NONE↑	OTHER↓	ANY↓
HiSA	32.5%	5.7%	20.3%	46.3%	52.0%	29.3%	4.9%	65.9%
CoAct-1	4.9%	4.9%	35.0%	60.2%	3.3%	32.5%	2.4%	65.0%
GTA1	16.3%	16.3%	25.2%	39.0%	16.3%	29.3%	14.6%	56.1%
Agent S3	22.0%	39.0%	6.5%	17.1%	39.0%	35.8%	13.8%	50.4%
LocalLSTC	9.8%	18.7%	28.5%	35.0%	21.1%	42.3%	14.6%	43.1%

Table 8: 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
Execution channels	GUI, Bash/Python, software API, and reserved control actions	GUI, Bash/Python, and reserved control actions	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 <code>continue</code> , <code>advance</code> , <code>error</code> , <code>stall</code> , and <code>finish</code> 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	Evidence-supported <code>advance</code> with maximum-interval fallback	Periodic or forced fallback without subgoal advance	Periodic or forced fallback without state routing
Termination	Final Verification after <code>TERMINATE</code> , with rejection feedback returned to planning	Task-level Final Verification without subgoal context	Termination without Final Verification

highest NONE incidence at 42.3%, compared with an average ANY incidence of 59.4% across the four baselines.

B GROUPED L2S–S2L ABLATION CONFIGURATIONS

The grouped ablations isolate the two sides of the L2S–S2L execution boundary under the fixed Qwen3.5-9B backbone. *w/o L2S* removes subgoal-conditioned commitment formation and uses task-level single-action planning, while retaining task-level post-execution control and Step Abstraction. *w/o S2L* retains the L2S commitment interface but replaces evidence-conditioned post-execution state updates with planner-intent updates. Table 8 specifies the corresponding configurations.

C WINDOWSAGENTARENA RUNTIME DIAGNOSTICS

Table 9 complements the OSWorld analysis in Table 2 with domain-level runtime statistics on WindowsAgentArena under the Qwen3.5-9B planner backbone.

Across all 154 tasks, LocalLSTC averages 30.5 planner steps, 8.7 subgoals, and 2.8 context refinements per task. Final Verification rejects termination on 21.4% of tasks, and 51.5% of these tasks subsequently succeed after returning to execution, with an average of 5.2 additional planner steps.

D 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

Table 9: Domain-level runtime behavior of LocalLSTC with Qwen3.5-9B on WindowsAgentArena. Execution-channel columns report action shares. Pass, Fail, and Bypass denote accepted termination, rejected termination followed by recovery, and cutoff without verification. Recovery statistics are conditional on recovery entry.

Summary			Commitment				Execution Channels				Evidence Control					
Domain	Tasks	SR ↑ (%)	Planning Structure				Channel Use			Context Refines	Verification			Recovery		
			steps	Subgoals /Task	Actions /Subgoal	Actions /step	GUI (%)	Bash (%)	API (%)		Pass (%)	Fail (%)	Bypass (%)	SR after ↑ recovery (%)	steps after recovery entry	
Chrome	17	64.1	17.8	4.5	5.0	1.3	92.9	0.5	0.0	1.2	70.6	23.5	5.9	75.0	2.7	
Clock	4	50.0	38.8	8.8	5.6	1.3	87.1	0.0	0.0	3.5	50.0	0.0	50.0	0.0	0.0	
File Explorer	19	68.4	20.2	7.3	3.3	1.2	64.8	22.7	0.0	1.8	63.2	26.3	10.5	60.0	2.3	
LibreOffice Calc	24	0.0	41.3	10.9	7.3	1.9	95.1	1.3	0.0	4.2	16.7	8.3	75.0	0.0	0.0	
LibreOffice Writer	19	10.5	43.5	13.2	5.0	1.5	90.5	2.8	0.0	4.6	5.3	15.8	78.9	33.3	33.0	
Microsoft Paint	3	66.7	36.7	4.0	12.3	1.3	93.6	0.0	0.0	2.0	33.3	33.3	33.3	100.0	0.0	
Microsoft Edge	13	30.8	24.2	6.6	4.3	1.2	93.1	1.5	0.0	2.2	53.8	15.4	30.8	0.0	0.0	
Notepad	2	100.0	7.5	3.0	3.3	1.3	65.4	11.5	0.0	0.0	0.0	100.0	0.0	100.0	3.0	
Settings	5	100.0	4.8	2.6	1.6	0.9	72.4	0.0	0.0	0.4	100.0	0.0	0.0	0.0	0.0	
VLC	21	43.7	32.1	10.2	3.8	1.2	79.3	11.4	0.0	3.0	28.6	28.6	42.9	33.3	4.5	
VS Code	24	58.3	31.8	9.3	4.6	1.3	86.0	6.5	0.0	2.8	41.7	29.2	29.2	57.1	3.0	
Windows Calculator	3	66.7	45.3	6.7	13.1	1.9	87.3	7.6	0.0	2.3	66.7	33.3	0.0	100.0	11.0	
All	154	42.9	30.5	8.7	5.1	1.5	87.8	5.3	0.0	2.8	40.3	21.4	38.3	51.5	5.2	

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

D.1 PROMPT CONSTRUCTION AND EXECUTION MODE

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. Prompt construction adapts this context to the current execution mode. In GUI mode, the context emphasizes visual state, interface-operation guidance, and available software API schemas. In `bash_execution` mode, it emphasizes file-system state, textual execution evidence, and platform-specific operations.

The runtime initializes the execution mode from task-level lexical cues and related-application priors, then updates it according to the channels used during execution. GUI and software API actions select GUI for the next planner call, while Bash/Python actions select `bash_execution`. For commitments spanning multiple channels, the final executed channel determines the subsequent mode.

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 selected from the foreground application and the task-level `related_apps` set, allowing multi-application tasks to retain guidance for relevant domains.

D.2 PLANNER INTERFACE AND ACTION SPACE

The planner returns a JSON object containing `thought`, `subgoal`, and `actions`. The `subgoal` field specifies the execution stage for the current commitment. The sentinel value `continue` retains the active subgoal, while any other value proposes a transition to a new subgoal. The nonempty `actions` list contains the ordered actions to execute under that commitment and may combine GUI interaction, Bash/Python execution, and software API calls.

The interface also provides the reserved actions `WAIT`, `TERMINATE`, and `INFEASIBLE`. `WAIT` retains the active subgoal while observing for an environment update or timeout. `TERMINATE` requests Final Verification after the preceding executable actions complete. `INFEASIBLE` records an infeasibility judgment. Terminal actions appear only at the end of a commitment, and validation rejects commitments containing both `TERMINATE` and `INFEASIBLE`. Table 10 summarizes the executable representation of each action type.

The runtime identifies reserved control actions directly and distinguishes GUI interaction and software API calls from their registered prefixes. Remaining executable strings enter the Bash/Python channel, where the active platform skill determines whether they are interpreted as shell com-

Table 10: Action types exposed to the planner and their executable representations.

Action type	Executable representation
GUI interaction	Executable <code>pyautogui</code> code string
Software API call	Schema-exposed <code>Class.method(...)</code> call string
Bash execution	A <code>bash_execution</code> command or code string interpreted by the runtime as a Bash or Python snippet
Event-driven wait	Reserved control action <code>WAIT</code>
Completion request	Reserved control action <code>TERMINATE</code>
Infeasibility report	Reserved control action <code>INFEASIBLE</code>

mands or Python code. Candidate proposals remain internal to the planner inference, and only the `actions` list from the selected proposal is executed.

D.3 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.

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.

D.4 OBSERVATION AND 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 divided into observation-aligned execution chunks. Actions requiring coordinate localization are grounded before execution unless executable coordinates are already available. After each chunk, the runtime waits for environment changes and passes the latest stable screenshot to subsequent execution. Screenshots acquired before planning and after execution also undergo stability checks, with the latest available frame used when the timeout expires.

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.

D.5 STATE ASSIGNMENT AND EVIDENCE POLICY

After Step Abstraction, the runtime assigns a transient routing state using a fixed precedence. A final `TERMINATE` request maps to `finish`. Execution failure maps to `error`, repeated no-progress evidence maps to `stall`, and a proposed subgoal different from the persistent 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.

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`. Steps containing `bash_execution` are excluded from visual no-change aggregation because textual readback may establish progress without 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.

Loop detection compares normalized action and result fingerprints across interactions. GUI and software API fingerprints encode normalized executable requests, while `bash_execution` fingerprints encode normalized commands or code. Result fingerprints represent GUI execution status, `bash_execution` status and textual output, or software API values and errors. Before execution, the runtime suppresses repeated equivalent GUI commitments and repeated action–result pairs according to the thresholds in Table 11.

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.

D.6 CONTEXT REFINEMENT

LocalLSTC extends the context-refinement mechanism of HiSA (Li et al., 2026b) with execution-state triggering. An evidence-supported `advance` makes completed subgoal evidence eligible for refinement once sufficient new evidence has accumulated. 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.

D.7 FINAL VERIFICATION AND EVIDENCE REFRESH

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 a recent sliding window. 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 and returns execution to planning with feedback about the remaining completion conditions. The latest screenshot and corresponding verifier feedback are retained as recovery evidence for the subsequent planner call.

D.8 RUNTIME PARAMETERS

Table 11 reports the runtime parameters shared across LocalLSTC experiments. Planner sampling uses `temperature`, `top_p`, and `top_k`, while the remaining parameters govern proposal generation, context refinement, execution timeouts, observation stability, repair, and runtime control.

D.9 MODEL RESPONSIBILITIES AND ACCOUNTING

LocalLSTC uses two model roles during execution. The planner backbone produces candidate proposals and the selected execution commitment, and the same backbone performs Step Abstraction,

Table 11: Runtime parameters shared across LocalLSTC experiments.

Rule or hyperparameter	Value
Model-inference temperature	0
Model-inference <code>top_p</code>	0.95
Model-inference <code>top_k</code>	20
Maximum candidate proposals per inference step	3
Minimum context-refinement interval	5 steps
Maximum context-refinement interval	20 steps
<code>bash_execution</code> timeout	180 seconds
Maximum planner parsing or repair attempts	3
Post-action observation timeout	10 seconds
Explicit <code>WAIT</code> timeout	20 seconds
Screenshot acquisition timeout	6 seconds
Evaluation retry timeout	25 seconds
Initial screenshot stabilization	5 low-change observations
Post-action screenshot stability check	2 stable frames
Step Abstraction ROI margin	50 pixels
Planner-side repeated GUI decision threshold	3 consecutive planner calls
Repeated GUI action-result threshold	5 repetitions
Repeated <code>bash_execution</code> 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

context refinement, and Final Verification under dedicated prompts. The visual grounder localizes GUI actions when executable coordinates are unavailable.

Output parsing, schema validation, action fingerprinting, timeout handling, state assignment, and routing are deterministic runtime operations. GPT-5.6-sol is used only for post-hoc control-failure annotation and does not participate in agent execution. Token accounting includes planner and visual-grounding calls when the corresponding model invocation is triggered.

D.10 EXECUTION BOUNDARY

All benchmark actions execute inside the benchmark-defined virtual machine. Planner outputs are validated before execution, software API calls are restricted to schema-exposed handlers, and GUI and Bash/Python actions operate within the permissions of the benchmark environment.

E PROMPT TEMPLATES

This appendix reports the prompt templates used by LocalLSTC. The blocks use the paper-facing `api` action name for software API calls.

E.1 PLANNER SYSTEM PROMPT

This prompt corresponds to `GLOBAL_PLANNER_PROMPT`.

System Prompt: Global Planner

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

General Instructions

...

Action Types

```

## api
Call one software-level 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.
- Never use 'pyautogui.hotkey('alt', 'f4')' or any Alt+F4 variant to
close a window. On Windows this can open the Shut Down Windows dialog
and can terminate the VM session. Use a visible in-app close/cancel
button, 'pyautogui.press('esc')' for dialogs, or execution-side app-
specific cleanup instead.

## 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.

## 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'.

```

E.2 PLANNER RESPONSE FORMAT PROMPT

This prompt corresponds to `PLANNER_RESPONSE_FORMAT_PROMPT`.

System Prompt: Planner Response Format

```

# Response Format
You may think through up to 3 candidate strategies 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 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 blocker reason 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"
]
}
```

```

E.3 REPAIR PROMPT

This prompt corresponds to `FIX_RESPONSE_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.
- Return exactly one final JSON object with 'thought', 'subgoal', and 'actions'.
- Each 'actions' item must be a bare string or number, not an object.
- A GUI action must be a valid Python 'pyautogui' code string.
- A software API action must be exactly one Python call expression such as 'GoogleDriveTools.upload_file(...)'
- Return exactly one repaired response.
- The final output must satisfy this response format exactly:

```
{response_format_prompt}
```

Original response:

```
{response}
```

Parse traceback:

```
{error_message}
```

E.4 STEP ABSTRACTION PROMPT

This prompt corresponds to 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}

Recovery hint: {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 subgoal to decide which output details are key; do not replace needed paths or IDs with generic wording.

E.5 FINAL VERIFICATION PROMPT

This prompt corresponds to 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 subgoal
- the full 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 full execution history and the provided screenshots.
- Some tasks may be infeasible by design. If required capabilities,
  variables, or app features were unavailable, return 'FAIL' rather than
  accepting a workaround.
- 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 full 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.

```

E.6 CONTEXT REFINEMENT PROMPT

This prompt corresponds to `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)") - these can be reused later
- Identify if we're stuck in loops, making progress, or stalled
- Provide actionable suggestions for the next step if there are issues
- If 'bash_execution' or software API already verified the requested file/data state, do not suggest GUI Save, reopen, reload, or refresh just to sync a stale window; suggest termination or another text-level verification instead.

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: This approach isn't working - try an alternative method or termination as infeasible.

F EXAMPLES OF RUNTIME CONTROL

The runtime-event analysis in Table 4 shows how LocalLSTC behaves after repetition, recovery entry, and rejected termination. The six successful OSWorld trajectories below illustrate how persistent state and execution evidence guide subsequent commitments after these events. The examples are summarized from the corresponding benchmark task configurations, `execution_log.json`, and `model.trace.txt`.

F.1 RECOVERY AFTER REPEATED ACTIONS

Task 01b269ae-2111-4a07-81fd-3fcd711993b0 requires filling blank cells in B1:E30 with the value from the cell above. Three equivalent file-editing commands produce no visible spreadsheet change. File readback then localizes the unresolved cells to column B while retaining the original range and the constraint against modifying unrelated cells. Under this evidence, L2S revises the execution command, reloads Calc, and verifies column B separately from columns C:E. Final Verification initially rejects termination because the visual evidence is insufficient. The runtime subsequently refreshes the spreadsheet, verifies the target range, and accepts completion.

Task 4127319a-8b79-4410-b58a-7a151e15f3d7 requires recursively counting lines in all PHP files under the current directory and displaying the result in a terminal. The Bash channel repeatedly returns the same 54-line result without producing the required visible terminal evidence. The active subgoal retains both the counting scope and the display requirement. L2S then switches execution to the GUI, opens a terminal, and enters the command there. Final Verification accepts termination once the visible terminal establishes both the command and its result.

F.2 STAGE-PRESERVING RECOVERY ACROSS EXECUTION CHANNELS

Task 3c8f201a-009d-4bbe-8b65-a6f8b35bb57f requires downloading an image and exporting a compressed copy as `compressed.jpeg` on the desktop. The trajectory completes

the download through Bash and performs the export through the GUI, but repeated GUI attempts produce incorrect filenames. File readback identifies the candidate files and their sizes. The compression subgoal remains active while L2S switches back to Bash, retains the valid 115 KB file, and normalizes its filename. The resulting evidence establishes the required location, filename, and size without restarting the completed stages.

Task `c867c42d-a52d-4a24-8ae3-f75d256b5618` requires exporting the Personal Address Book from Thunderbird and converting the resulting CSV to XLSX in LibreOffice Calc. GUI export produces `Personal Address Book.csv`, and file readback reveals that the filename does not match the task requirement. S2L retains the completed export evidence while keeping the conversion stage active. L2S then uses Bash to rename the file to `contacts.csv`, completes the conversion in Calc, and confirms through Bash readback that `contacts.xlsx` exists on the desktop.

F.3 EVIDENCE-GROUNDED TERMINATION

Task `1e8df695-bd1b-45b3-b557-e7d599cf7597` requires adding a Profit column beside Sales and COGS and computing the weekly difference. After initial numeric-type and file-state errors, a corrected script completes the edit, and readback confirms the column name and sample calculations. Final Verification rejects the first termination request because visual confirmation is still unavailable. L2S then resolves the Calc recovery dialog and verifies the Profit column and its values in the visible spreadsheet. Final Verification accepts the subsequent termination request once the required visual evidence is available.

Task `6ada715d-3aae-4a32-a6a7-429b2e43fb93` requires inserting `1.png` from the desktop at the current cursor position in Writer. The first termination request is rejected because the available evidence does not establish the insertion result. L2S retains the image-insertion subgoal and inspects both the inserted image and its document position. A second termination request is rejected after a context menu obscures the relevant content. After subsequent interaction exposes the required visual evidence, Final Verification accepts completion.