

Agent Alpha: Unifying Generation, Exploration and Evaluation via Tree Search for Computer-Use Solution Discovery

Sizhe Tang Rongqian Chen Tian Lan
The George Washington University

Abstract

Discovering solutions for complex computer-use tasks remains a fundamental challenge due to vast action spaces, non-reversible state transitions, and sparse rewards. Existing approaches for Graphical User Interface (GUI) agents operate as unidirectional processes, where the lack of regressive ability prevents the reuse of partial successes and recovery from early missteps. We introduce Agent Alpha, a unified framework that synthesis generation, exploration, and evaluation through step-level Monte Carlo Tree Search (MCTS) for efficient solution discovery. By integrating Alpha-UCT guided search into the interaction loop, Agent Alpha enables deliberate planning with early pruning of suboptimal branches and efficient prefix reuse. We further employ comparison-driven evaluation to mitigate absolute scoring biases and diversity-constrained expansion to maintain a compact, informative search space. We analyze the regret bound of Alpha-UCT and show it achieves tighter confidence bounds that accelerate identification of successful trajectories. On the OSWorld benchmark, Agent Alpha achieves a state-of-the-art success rate of $\sim 77\%$, significantly outperforming trajectory-level baselines. Code is available at: github.com/sizhe-tang/AgentAlpha.

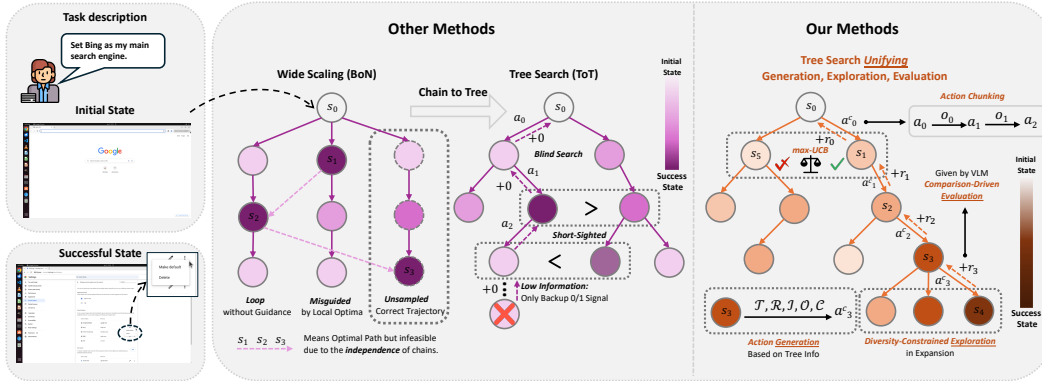


Figure 1: Overview of Agent Alpha. Existing approaches operate as unidirectional processes without the ability to model or exploit the structure of the planning space, limiting solution discovery efficiency. Agent Alpha addresses this by synergizing generation, exploration, and evaluation through step-level MCTS under the Alpha-UCT bound, enabling regressive planning with refined exploration and prefix reuse. Nodes with deeper color represent states with higher potential. $\mathcal{T}, \mathcal{R}, \mathcal{I}, \mathcal{O}, \mathcal{C}$ denote trajectory, reflection, task instruction, observation, and operation context respectively.

1 Introduction

Solving complex computer-use tasks requires discovering multi-step action sequences across vast, dynamic environments—a challenge that demands both strategic exploration

Correspondence to: s.tang1@gwu.edu.

and robust error recovery. Recent methods such as Chain of Thoughts (CoT) (Wei et al., 2022), Tree of Thoughts (ToT) (Yao et al., 2023), and Behavior Best of N (bBoN) (Gonzalez-Pumariega et al., 2025) have demonstrated remarkable potential in enhancing computer-use agents (CUAs) based on Multimodal Large Language Models (MLLMs). Yet these approaches operate primarily as unidirectional processes (Snell et al., 2024) without the ability to model or exploit the structure of the planning space. In the absence of a mechanism to adapt to dynamic value feedback, they cannot retrospectively evaluate the effectiveness of past actions—neither to rectify the course when a suboptimal branch has been chosen, nor to focus compute on promising prefixes. Consequently, a single early misstep often cascades into an irreversible failure, severely limiting the efficiency of solution discovery.

To address this, we introduce Agent Alpha, a unified framework that synergizes the generation, exploration, and evaluation capabilities of MLLMs through step-level Monte Carlo Tree Search (MCTS). By transforming linear execution into a regressive planning process, Agent Alpha recovers from early and compounding errors autonomously, reuses promising action prefixes, and terminates stalled branches early. The search is guided by Alpha-UCT, a new selection bound that uses maximum values on search paths to augment exploration and accounts for the dependent samples that arise from comparative evaluation and cross-iteration reflection (Ma et al., 2025; Wang & Wang, 2025). By modeling the predictive residual as a martingale difference sequence, we show that Alpha-UCT achieves a tighter confidence bound (Freedman, 1975; Kato, 2020) than standard UCT, enabling the algorithm to identify successful trajectories with significantly fewer samples.

Our framework further introduces three methodological innovations tailored to the computer-use domain. First, we propose search-aware action generation, where the agent aggregates information from previously explored branches, including failed ones, to refine its proposal distribution via cross-trajectory reflection. Second, we introduce diversity-constrained expansion that enforces semantic uniqueness among sibling nodes through lexical normalization, countering the mode-seeking tendency of aligned models and yielding a compact yet diverse search space. Third, we employ comparison-driven consistent evaluation, in which sibling actions are assessed jointly rather than in isolation, producing discriminative relative signals that mitigate the estimation bias and anchoring effects inherent in absolute scoring.

We evaluate Agent Alpha on the OSWorld benchmark (Xie et al., 2024), a comprehensive evaluation suite for multimodal computer control spanning diverse task scenarios. Agent Alpha achieves a new state-of-the-art success rate of $\sim 77\%$, surpassing human-level performance and significantly outperforming existing methods under comparable computational budgets. These results demonstrate that unifying generation, exploration, and evaluation within a structured search process leads to substantially more effective solution discovery in complex computer-use environments.

Our contributions are four-fold:

- We propose Agent Alpha, a unified MCTS-based framework that synergizes generation, exploration, and evaluation of MLLMs, transforming solution discovery from a unidirectional process into regressive, structure-aware planning.
- We introduce three designs tailored to computer-use solution discovery: tree-informed action generation, diversity-constrained expansion, and comparison-driven consistent evaluation.
- We derive the regret bound of Alpha-UCT and show it achieves tighter confidence than standard UCT, enabling more sample-efficient solution discovery.
- Agent Alpha achieves a state-of-the-art success rate of $\sim 77\%$ on OSWorld, surpassing human-level performance.

2 Background

Computer-Use Agents General-purpose computer use is modeled as a Partially Observable Markov Decision Process (POMDP) $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \Omega, \mathcal{O} \rangle$. The environment state $s_t \in \mathcal{S}$ is not fully visible; instead, the agent receives an observation $o_t \in \Omega$ via $\mathcal{O}(s_t, \mathcal{I})$, comprising the current screenshot and structured accessibility metadata (e.g., the accessibility tree), where $\mathcal{I}$ is the user instruction. Recent work spans both general-purpose agentic

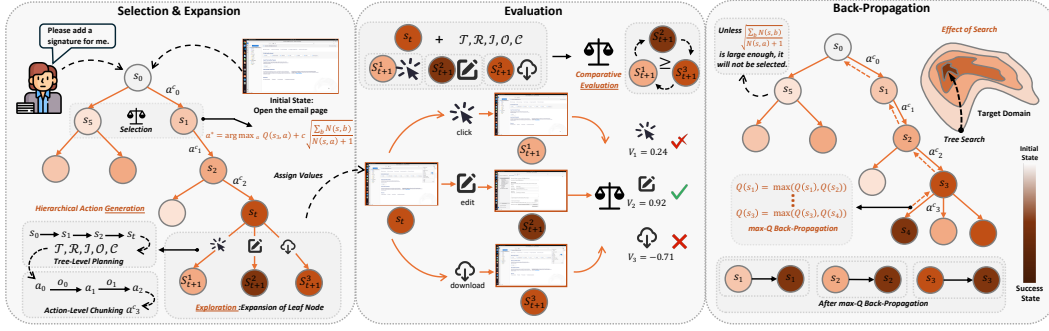


Figure 2: The detailed search process applied in Agent Alpha. The process consists of *Selection*, *Expansion*, *Evaluation*, and *Back-Propagation*. The search process will stop if the budget runs out or the task is successful.

frameworks (Song et al., 2024; Wang et al., 2024) and specialized GUI agents (Niu et al., 2024; Rawles et al., 2024; Xie et al., 2024).

Test-Time Compute for Solution Discovery. Allocating additional compute at test time has proven effective for large models (Muennighoff et al., 2025), typically via parallel sampling (Zhu et al., 2025; Gonzalez-Pumariaga et al., 2025) or iterative refinement (Lightman et al., 2024). For computer-use agents, trajectory-level methods such as bBoN (Gonzalez-Pumariaga et al., 2025) prioritize breadth but incur high cost with no cross-trajectory information sharing, while step-wise parallel strategies often fall into repetitive cycles. These approaches share three fundamental limitations: unidirectional generation without error recovery, isolated execution that prevents knowledge transfer across trajectories, and undirected exploration that scales poorly with vast action spaces.

Monte Carlo Tree Search Monte Carlo Tree Search (MCTS) (Kocsis & Szepesvári, 2006) builds a lookahead tree to estimate the optimal policy (Silver et al., 2017; Tang et al., 2025) through iterative selection, expansion, evaluation, and back-propagation. Starting from the root s_0 , the algorithm selects child nodes by maximizing the Upper Confidence Bound applied to Trees (UCT) (Kocsis & Szepesvári, 2006): $a_t = \arg \max_{a \in \mathcal{A}(s)} Q(s, a) + c_{uct} \sqrt{\frac{\ln N(s)}{N(s, a)}}$, where $Q(s, a)$ is the estimated action value, $N(s)$ and $N(s, a)$ are the visit counts of the parent and child nodes, and c_{uct} is an exploration constant. Upon reaching a leaf node s_L , one or more children are expanded, and the estimated value v is propagated backward along the traversed path: $N(s, a) \leftarrow N(s, a) + 1$, $Q(s, a) \leftarrow Q(s, a) + \frac{v - Q(s, a)}{N(s, a)}$.

3 Methodology

3.1 Overview of Agent Alpha

Problem Statement We formalize computer-use tasks as a solution discovery problem. The transition function $g(s_{t+1}|s_t, a_t)$ is deterministic but unknown, and rewards are sparse: $r = 0$ for $t < T$ with a non-zero signal only upon task completion. The objective is to discover an action sequence $(a_0, a_1, \dots, a_T)$ that reaches a success state. Following recent GUI agents (Peng et al., 2024; Qin et al., 2025), we use a Vision-Language Model π_θ over an action space $\mathcal{A}$ of computer commands (e.g., `click(x, y)`, `type(text)`, `scroll(dir, len)`). In standard reactive paradigms (Yao et al., 2022), the agent generates actions autoregressively from the history $h_t = (o_0, a_0, \dots, o_t)$: $p(a_t|h_t, \mathcal{I}) = \sum_{\tau_t} \pi_\theta(a_t|\tau_t, h_t, \mathcal{I}) \pi_\theta(\tau_t|h_t, \mathcal{I})$ where τ_t is the reasoning trace. This unidirectional process offers no mechanism to revisit earlier decisions when the chosen path proves suboptimal.

Unifying Generation, Exploration, and Evaluation. Agent Alpha constructs a search tree iteratively from the initial task state until the budget is exhausted or the task is completed. Each iteration consists of four phases. During *Selection*, the agent traverses the tree by

maximizing the Alpha-UCT metric to identify the most promising node for expansion:

$$a' = \arg \max_{a \in \mathcal{A}(v)} \text{Alpha-UCT}(v, Q, N). \quad (1)$$

In *Expansion*, the policy π_θ samples candidate actions $\mathcal{A}^{(i)} = \{a_1^{(i)}, a_2^{(i)}, \dots, a_k^{(i)}\}$, which are filtered through a normalization operator $\phi(\cdot)$ to remove semantic duplicates, yielding the de-duplicated set $\bar{\mathcal{A}}^{(i)} = \phi(\mathcal{A}^{(i)})$. Each admitted node v is then assigned a value $Q(v)$ via comparative judgment. In *Back-Propagation*, node statistics initialized by *Evaluation* along the search path are updated via max-value propagation.

Novel Search Designs. Our framework introduces four innovations tailored to computer-use search. First, action generation is driven by tree-level dynamic reflection, which aggregates information across explored branches rather than relying on single-pass reasoning. Second, diversity-constrained expansion eliminates semantically redundant sibling nodes, focusing compute on distinct and promising paths. Third, comparison-driven evaluation assesses sibling actions jointly instead of scoring them independently, yielding consistent and discriminative guidance for the search. Fourth, we adopt action chunking, which groups a sequence of atomic actions into a single tree node, enabling longer-horizon planning and more meaningful evaluation per step. Detailed implementation refers to Appendix B.

Alpha-UCT Bound. Standard MCTS algorithms typically view all actions/rewards as independent samples in a multi-armed bandit formulation (Kocsis & Szepesvári, 2006). However, samples in Agent Alpha could be correlated. The tree-level reflection and comparative-driven judgment applied lead to dependent evaluations and action generations (e.g., due to evolving context and dialog history (Ma et al., 2025; Wang & Wang, 2025)), thus embedding additional unconfirmed knowledge (Kato, 2020) across search. We take into account such dependent samples in MCTS and develop a new Alpha-UCT bound that ensures more efficient search compared with the standard UCT (Kocsis & Szepesvári, 2006). We further leverage the maximum value on search paths to augment the exploration term, to more quickly identify missteps and promising prefixes. The regret of Alpha-UCT is analyzed.

3.2 Search-Aware Action Generation via Tree-Informed Reflection

Standard paradigms such as CoT (Wei et al., 2022) condition both action and reasoning generation on the history in a single pass. Agent Alpha transcends this by employing an informed action generation mechanism that leverages interaction information gathered across the entire search tree.

The reflection $\mathcal{R}$ is not a static descriptor but dynamically aggregates trial-and-error information over the tree topology. As Agent Alpha updates node statistics through iterations, the information from all explored paths is distilled into the selection process. Let $\mathcal{T}_t^{(i)} = \{(a_{1:k}^{(i)}, o_{1:k}^{(i)}, Q_{1:k}^{(i)})\}_{k=1}^t$ be the trajectory of actions, observations, and node values from root to step t in iteration i . The trajectory at iteration i is derived from all prior iterations:

$$\mathcal{T}_t^{(i)} = f_{\text{search}}(\mathcal{T}_{\text{leaf.depth}}^{(i-1)}), \quad (2)$$

implying that trajectory information is inherited across iterations. Let $\mathcal{R}^{(i)}$ denote the reflection used in the i -th iteration. Since reflection is generated only at the leaf node:

$$\mathcal{R}^{(i)} = f_{\text{reflect}}(\mathcal{T}_{\text{leaf.depth}}^{(i-1)}, \mathcal{I}). \quad (3)$$

Thus, even when the agent follows the same search path, the reflection evolves across iterations. The action generator synthesizes the collective intelligence $\mathcal{R}_t^{(i)}$ by aggregating insights from diverse explored branches, refining the proposal distribution against known failure modes identified during search.

Let $\text{ctx}^{(i)}$ denote the workspace context (e.g., clickable elements). The planning $\mathcal{P}_t^{(i)}$ at the leaf node in iteration i is:

$$\mathcal{P}_t^{(i)} = f_{\text{plan}}(\mathcal{I}, o_t^{(i)}, \mathcal{R}_t^{(i)}, \text{ctx}_t^{(i)}). \quad (4)$$

The planning and observation are then fed into the grounding model to produce the specific action: $a_t = f_{\text{predict}}(\mathcal{P}_t^{(i)}, o_t^{(i)})$. To form an action chunk, this atomic action-observation cycle is repeated sequentially; the resulting actions are grouped into a single chunk and the final state is assigned to the expanded node. Action chunking enables longer-horizon planning and prevents individual atomic actions from being prematurely judged as unproductive.

3.3 Diversity-Constrained Exploration

To expand a tree, Agent Alpha needs to do the above action generation for the same leaf node, which incurs the problem of efficient exploration. That is, given identical state $s_t^{(i)}$, independent sampling $a_{t,1}^{(i)}, \dots, a_{t,k}^{(i)} \sim \pi_\theta(\cdot | s_t^{(i)})$ often collapses onto a narrow subset of high-probability tokens. This results in structural redundancy where sibling branches explore overlapping regions of the action space, creating an exploration illusion where the search tree grows in size without a corresponding increase in state-space coverage.

We address this through a constrained exploration operator that enforces uniqueness among sibling nodes via online semantic de-duplication. Let $\phi : \mathcal{A}^{(i)} \rightarrow \bar{\mathcal{A}}^{(i)}$ be a normalization function that maps a raw action set to its functional representation set by stripping superficial formatting variations while preserving essential elements such as coordinates and command types. For example, if the planner indicates that agent needs to click the download button, then the action `click(300, 450)` and `click(300, 452)` should share the same function and one of them should be rejected for sampling. During the expansion of node v , candidate actions are sampled in parallel from π_θ . The normalization operator ϕ then removes semantic duplicates: an action $a_k^{(i)}$ is retained if and only if its normalized representation is unique among all retained siblings:

$$\bar{\mathcal{A}}^{(i)} = \{a_k^{(i)} \sim \pi_\theta \mid \forall j < k, \phi(a_k^{(i)}) \neq \phi(a_j^{(i)})\} \quad (5)$$

This mechanism yields an adaptive branching factor $b^*(v) = |\phi(\bar{\mathcal{A}}^{(i)})| \leq K$. If a state presents a narrow decision bottleneck, the exploration gracefully yields fewer branches, thereby concentrating the search compute on genuinely distinct decision paths. This ensures that the search budget is allocated proportionally to actual uncertainty, transforming the expansion phase from a blind sampling process into a diversity-aware construction of a compact and informative search space.

3.4 Comparison-Driven Consistent Evaluation

Evaluating the quality of candidate actions is central to effective tree search, yet language models exhibit significant range instability and scale-interpretation bias when tasked with independent scalar scoring (Lorentz, 2016). In the context of GUI navigation, absolute judgments often suffer from overconfidence or excessive hedging, where minor prompt perturbations lead to unpredictable score shifts (Gu et al., 2024). This lack of calibration is particularly detrimental when the search algorithm must resolve subtle decision boundaries between semantically similar sibling actions, as independent queries fail to provide a consistent reference frame.

Agent Alpha addresses this through comparison-driven consistent evaluation, a relative assessment paradigm that transforms the evaluation phase into a joint assessment task. Instead of scoring nodes in isolation, the MLLM-based evaluator jointly assesses the set of sibling nodes expanded from a common parent. Let v be the parent node at state s_t , and $\{v'_1, \dots, v'_k\}$ be its children generated by actions $\{a_{t,1}, \dots, a_{t,k}\}$. The evaluator f_{judge} computes the node values as a normalized vector:

$$[V(v'_1), \dots, V(v'_k)] = f_{\text{judge}}(s_t, \{(a_{t,j}, o'_{t,j})\}_{j=1}^k, \mathcal{I}) \quad (6)$$

where each $(a_{t,j}, o'_{t,j})$ pair represents the proposed action and its resulting interaction consequence, grounded in the user instruction $\mathcal{I}$.

In standard MCTS, the predicted node values will be updated as the cumulative mean value through the search path. However, slight operation differences in computer-use scenarios can lead to vastly different outcomes. Thus, mean-value aggregation delays the exposure of erroneous operations with low values since the negative feedback is easily hidden by the high Q -value of the previous path. To this end, we update the exploitation term in UCT-type bound using the max-value of nodes in the search path. For each node v in the search path except for the leaf, the Q value is updated as

$$Q(v) = \max(Q(v), V_{\text{new}}) \quad (7)$$

where V_{new} is the new value to be updated into the tree. In this way, the agent can select the most potential action from the perspective of the whole tree, which prevents fatal actions from going undetected. To better distinguish the meaningful state-action pair and the wrongly guided one, we set the judgment score for each node within $[-1, 1]$.

3.5 Alpha-UCT and Regret Analysis

Standard MCTS algorithms typically view actions/rewards as independent samples (Kocsis & Szepesvári, 2006). However, Agent Alpha leverages the dynamic reflection and comparative evaluation mechanism, resulting in correlated samples during search, due to evolving context and dialog history (Ma et al., 2025; Wang & Wang, 2025)). More precisely, reward correlation/dependence arises in the evaluation since rewards are evaluated in a comparative manner; the reflection mechanism introduces a temporal dependency across search iterations, altering the independence assumption of exploration and action sampling. Such correlation/dependence may serve as unconfirmed knowledge (Kato, 2020) in MCTS.

We model the value estimation process as a martingale concentration problem. Let X_t be the evaluation score and $\hat{\theta}_t$ be the reflection prior generated by the agent. The efficiency of the search is fundamentally governed by the residual variance of the prediction error rather than the raw variance of the environment: $\sigma_{\text{res}}^2 = \mathbb{E}[(X_t - \hat{\theta}_t)^2 \mid \mathcal{F}_{t-1}]$. Here, the filtration $\mathcal{F}_{t-1}$ represents the history of observed trajectories and reflections. Unlike independent samples where variance is constant, the conditional variance here decreases as the reflection $\hat{\theta}_t$ becomes more accurate.

Leveraging the concentration inequality (Kato, 2020) for martingales, we obtain Alpha-UCT bound as action our selection policy as follows:

$$a^* = \arg \max_{a \in \mathcal{A}(v)} \left[Q_{\max}(v, a) + c \sqrt{\frac{\sum_b N(v, b)}{N(v, a) + 1}} \right], \quad Q_{\max}(v, a) = \max_{k \in \mathcal{T}(v, a)} V_k. \quad (8)$$

where $\mathcal{T}(v, a)$ is the set of all completed trajectories passing through node (v, a) , $Q_{\max}(v, a)$ denotes the maximum evaluation score observed in the subtree of action a , and V_k is the comparative judgment score. Eq. 8 combines max-value backup with reflection-conditioned estimation. The analysis below characterizes the variance-dependent regret benefit of the latter, while the max-backup design is empirically validated in Table. 4. Next, we analyze the regret bound of Alpha-UCT.

Theorem 3.1. (Regret Bound with Unconfirmed Knowledge). *The cumulative regret R_T for an agent leveraging unconfirmed knowledge satisfies:*

$$R_T \leq \sum_{i \in \mathcal{A} \setminus \{i^*\}} \left(\frac{8\sigma_{\text{res},i}^2 \ln T}{\Delta_i} + \frac{16 \ln T}{3} + 2\Delta_i \right). \quad (9)$$

This theorem leads to a critical insight regarding the complexity of our framework compared to standard methods.

Corollary 3.2. (Asymptotic Regret Complexity). *Under the reflection-guided framework of Agent Alpha, the cumulative regret R_T over horizon T with branching factor K satisfies:*

$$R_T = \mathcal{O} \left(K \cdot \sigma_{\text{res}}^2 \cdot \ln T \right). \quad (10)$$

Table 1: Success rate comparison on OSWorld tasks. **Chr.**: Chrome, **Calc**: LibreOffice Calc, **Imp.**: LibreOffice Impress, **Wri.**: LibreOffice Writer, **GIMP**: GNU Image Manipulation Program, **VSC**: VSCode, **Multi**: Multi-apps, **Thu.**: Thunderbird, **OS**: OS, **VLC**: VLC Media Player. **Red** indicates the best performance, and **Blue** indicates the second best.

Methods	Chr.	Calc	Imp.	Wri.	GIMP	VSC	Multi	Thu.	OS	VLC	Avg.
Agent s3 (N=10)	62.96	87.23	72.26	86.83	73.08	82.61	63.90	73.33	79.17	62.94	72.58
UiPath Screen Agent	69.48	80.43	70.19	69.57	69.23	78.26	52.97	73.33	70.83	60.82	67.14
OS-Symphony	62.96	76.60	48.81	78.13	80.77	78.26	55.30	80.00	78.26	61.53	65.77
GBOX Agent	62.96	74.47	63.74	68.18	76.92	91.30	49.73	66.67	70.83	46.29	64.22
GTA1	58.61	63.83	65.53	60.74	76.92	82.61	50.91	80.00	79.17	57.88	63.41
claude-sonnet-4-5	62.96	72.34	68.00	82.48	53.85	73.91	49.54	60.00	70.83	58.18	62.88
CoACT-1	54.26	70.21	50.28	73.91	65.38	78.26	47.87	73.33	75.00	71.94	60.76
Agent Alpha (Ours)	67.39	87.23	78.72	91.30	96.15	100	59.14	80.00	95.83	70.58	77.29

Table 2: Improvement Analysis: Breakdown of tasks failed by Agent S3 but successfully solved by Agent Alpha across 10 domains.

Metric	Chr.	Calc	Imp.	Wri.	GIMP	VSC	Multi	Thu.	OS	VLC	Total
# S3 Failed	21	16	23	7	12	6	58	5	6	11	165
# Solved by Alpha	5	8	4	3	11	6	7	1	5	6	56
improvement Rate (%)	23.8	50.0	17.4	42.9	91.7	100.0	12.1	20.0	83.3	54.5	33.9

This implies that the quality of the agent’s reflection directly dictates search speed: as the reflection $\hat{\theta}$ aligns closer to the outcome X , $\sigma_{\text{res}}^2 \rightarrow 0$, effectively minimizing the regret. To quantify this advantage, we compare the sample complexity of Alpha-UCT against the standard UCT baseline that considers all actions are independent.

Corollary 3.3 (Efficiency Gain via Variance Reduction). *Let $\sigma_H^2 = 1/4$ denote the Hoeffding variance proxy for evaluation scores in $[0, 1]$. For each suboptimal action a , the ratio between the variance-dependent regret terms of Alpha-UCT and standard UCT is*

$$\frac{R_{T,\text{var},a}^{\text{Alpha}}}{R_{T,\text{var},a}^{\text{UCT}}} = \frac{\sigma_{\text{res},a}^2}{\sigma_H^2}.$$

Thus, whenever $\sigma_{\text{res},a}^2 < 1/4$, reflection-conditioned estimation yields a strictly smaller variance-dependent regret term.

This corollary theoretically validates that Agent Alpha converts predictive accuracy into search efficiency. The ratio $\sigma_{\text{res}}^2/\sigma_H^2$ quantifies the reduction of the variance-dependent regret term relative to the standard Hoeffding-based UCT baseline. The proof is detailed in Appendix A.

4 Experiments and Analysis

4.1 Experiment Setting.

Benchmark. We conduct our experiments on the OSWorld benchmark (Xie et al., 2024), a comprehensive evaluation platform designed for multimodal agents in real computer environments. The benchmark encompasses diverse task scenarios, ranging from fundamental OS operations and daily office workflows to professional software manipulation and complex cross-application tasks.

Baselines. We compare the performance and efficiency of Agent Alpha against state-of-the-art methods listed on the OSWorld leaderboard, including UiPath Screen Agent, Claude Sonnet 4.5, Agent S3 (Gonzalez-Pumariega et al., 2025), CoAct-1 (Song et al., 2025), GTA1 (Yang et al., 2025), OS-symphony (Yang et al., 2026) and GBOX Agent.

Table 3: Performance comparison between Agent Alpha(Exp. Node=5, Action Chunking =1) and Agent S3(N=3), both use gpt-5-mini as base model.

Method	Success Rate (%)	Alignment Score(%)	Average Step	Avg Time (s)
Agent S3	54.29	69.5	8.88	313.37
Agent Alpha	64.27	82.2	7.98	1116.5

Table 4: Ablation study on different module configurations. We analyze the impact of each sub-module choice on success rate and runtime. Par. here is the abbreviation for Parallel.

Setting	Judge Mech.		Backup Mech.		Tree Par.		Results	
	Comp.	Indep.	Max	Mean	Action	Env.	SR (%)	Speed
w/o Comp. Judge		✓	✓		✓	✓	57.96	-
w/o Max Backup	✓			✓	✓	✓	45.42	-
w/ Action Par.	✓		✓		✓		-	4.4x
w/ Env. Par.	✓		✓			✓	-	2.1x
Alpha Agent (Full)	✓		✓		✓	✓	64.27	-

4.2 Main Results

In this section, we evaluate the performance of Agent Alpha against seven baseline models on the OSWorld benchmark. The Agent Alpha experimental setup utilizes GPT-5.2 with optimized hyperparameters: expansion factor of 5 and maximum MCTS iterations of 20. For complex, long-horizon tasks (such as Multi-apps), we utilize action chunking number of 5 and reduce iterations to 15 to balance efficiency.

As illustrated in Table 1, Agent Alpha establishes a new state-of-the-art with an average Success Rate (SR) of 77.29%. This outperforms the strongest baseline, Agent s3 w/ Opus 4.5 + GPT-5 bBoN ($N = 10$) (Gonzalez-Pumariega et al., 2025), by a margin of 4.71%. Notably, given that human performance on this benchmark is approximately 72% (Xie et al., 2024), Agent Alpha successfully masters how to discover solutions for computer-use tasks.

4.3 Comparison with SOTA

Tables 2 and 3 compare Agent Alpha against the SOTA baseline Agent S3 both with GPT-5-mini on successful tasks. The primary distinction lies in the search strategy: Agent S3 uses Best-of-N ($N = 3$) and maximum trajectory depth of 30 steps, while Agent Alpha employs MCTS with an expansion factor of 5, max iteration of 20 times and no action chunking. We evaluate performance using SR and Alignment Score, which measures adherence to human logic. Alignment Score is the fraction of executed steps judged consistent with reasonable human progress toward the task goal. We also report Average Step and Average Time to assess efficiency.

As illustrated in Table 3, Agent Alpha demonstrates significant improvements over the baseline, achieving a Success Rate of 64.27% compared to 54.29% for Agent S3. Our model also exhibits superior solution discovery capabilities. To further investigate the robustness of our approach, Table 2 provides a breakdown of tasks that Agent S3 failed to solve but were successfully completed by Agent Alpha. Out of 165 tasks failed by the baseline, Agent Alpha successfully solved 56, yielding an overall improvement rate of 33.9%.

We further compare the inference-time efficiency of Agent Alpha with representative test-time scaling and single-pass baselines in Appendix D, including wall-clock time, LLM/API calls, and token usage under a matched GPT-5-mini backbone.

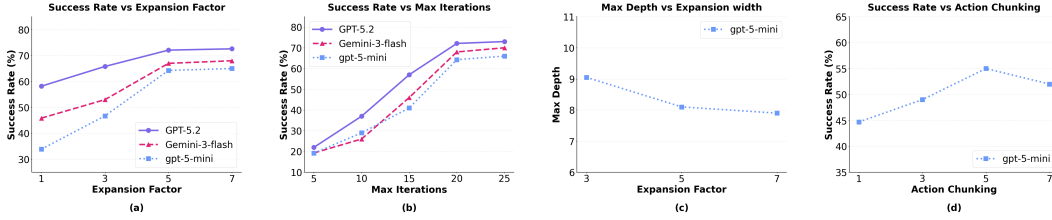


Figure 3: Analysis of hyperparameters: (a) Success rate vs. Expansion Nodes, (b) Success rate vs. Maximum Iterations, (c) Max Depth vs. Expansion factor, and (d) Success rate vs. Action chunking on long-horizon tasks.

4.4 Analysis of Hyperparameters

Impact of Expansion factor (N): In Figure 3(a), we analyze the effect of varying the expansion factor while fixing the maximum MCTS iterations to 20. Performance improves significantly as the node count increases from 1 to 5. However, the gain saturates beyond $N = 5$. This suggests that for most OSWorld tasks, the optimal action lies within the top-5 probabilities generated by the policy model. Consequently, increasing N beyond 5 offers diminishing returns with unnecessary computational overhead.

Impact of MCTS Iterations: Figure 3(b) displays the success rate as a function of maximum iterations (with $N = 5$ and chunking=1). The results show a steep performance increase up to 20 iterations, after which the curve plateaus. Lower iteration counts fail to explore the search tree sufficiently for complex tasks, while 20 iterations provide a robust balance between search depth and inference time.

Search Breadth vs. Trajectory Length: Figure 3(c) highlights an inverse relationship between expansion width and trajectory depth. As the expansion factor increases, the average depth of successful trajectories decreases. This indicates that a broader search allows the agent to discover more efficient shortcuts and optimal paths, rather than relying on sub-optimal, longer sequences.

Action Chunking for Long-Horizon Tasks: In Figure 3(d), we evaluate action chunking on long-horizon domains (Chrome and Multi-apps). While increasing the chunk size to 5 improves performance by allowing the agent to plan macro-actions, increasing it further to 7 causes a performance drop. This degradation is likely due to the loss of granular control and the increased difficulty in error recovery when too many actions are committed simultaneously without intermediate feedback.

4.5 Ablation Study

We evaluate individual component contributions via an ablation study (Table 4) using GPT-5-mini with fixed hyperparameters: 20 MCTS iterations, expansion factor $N = 5$, and action chunk size 1.

Impact of Judge Mechanism. We first examine the effectiveness of the Comparative Judge. As shown in the results, replacing the Comparative Judge with an Independent Judge leads to a performance drop from 64.27% to 57.96%. This indicates that comparative reasoning/evaluation is crucial to accurately distinguish between superior and inferior actions, whereas independent evaluation struggles to capture the relative quality of candidate steps in complex computer-use environments.

Impact of Backup Mechanism. Replacing the *Max* backup with a *Mean* backup results in a significant performance degradation of 18.85% (dropping to 45.42%). This sharp decline suggests that for open-ended OS tasks, an "optimistic" update strategy is essential. Since the goal is to find *one* successful trajectory, the Max backup correctly prioritizes high-potential paths, without diluting the signal by sub-optimal siblings as in Mean backup.

5 Conclusion

We presented Agent Alpha, an MCTS-based framework that unifies generation, exploration, and evaluation for efficient solution discovery in computer-use tasks. Alpha-UCT guides the search with tighter regret bounds than standard UCT by leveraging maximum search-path values and accounting for dependent samples. Step-level regressive planning enables error recovery and prefix reuse, while comparison-driven evaluation and diversity-constrained expansion further improve search quality. On the OSWorld benchmark, Agent Alpha achieves a $\sim 77\%$ success rate, surpassing both state-of-the-art baselines and human-level performance. The discovered solution trajectories can further serve as high-quality demonstrations for downstream policy improvement, pointing toward a promising direction for building stronger computer-use agents.

References

- David A Freedman. On tail probabilities for martingales. *the Annals of Probability*, pp. 100–118, 1975.
- 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.
- Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, et al. A survey on llm-as-a-judge. *The Innovation*, 2024.
- Go Kato. Concentration inequality using unconfirmed knowledge. *arXiv preprint arXiv:2002.04357*, 2020.
- Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pp. 282–293. Springer, 2006.
- Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let’s verify step by step. In *International Conference on Learning Representations (ICLR)*, 2024.
- Richard Lorentz. Using evaluation functions in monte-carlo tree search. *Theoretical computer science*, 644:106–113, 2016.
- Wei Ma, Yixiao Yang, Qiang Hu, Shi Ying, Zhi Jin, Bo Du, Zhenchang Xing, Tianlin Li, Junjie Shi, Yang Liu, et al. Rethinking testing for llm applications: Characteristics, challenges, and a lightweight interaction protocol. *arXiv preprint arXiv:2508.20737*, 2025.
- Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori B Hashimoto. s1: Simple test-time scaling. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 20286–20332, 2025.
- Runliang Niu, Jindong Li, Shiqi Wang, Yali Fu, Xiyu Hu, Xueyuan Leng, He Kong, Yi Chang, and Qi Wang. Screenagent: A vision language model-driven computer control agent. *arXiv preprint arXiv:2402.07945*, 2024.
- Zhiliang Peng, Wenhui Wang, Li Dong, Yaru Hao, Shaohan Huang, Shuming Ma, Qixiang Ye, and Furu Wei. Grounding multimodal large language models to the world. In *The Twelfth International Conference on Learning Representations*, 2024.
- 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.

- Christopher Rawles, Sarah Clinckemaiellie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, et al. Android-world: A dynamic benchmarking environment for autonomous agents. *arXiv preprint arXiv:2405.14573*, 2024.
- David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dhharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.
- Charlie Snell et al. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.
- Chan Hee Song, Jiaman Wu, Clayton Washington, Brian M Sadler, Wei-Lun Chao, and Yu Su. Trial and error: Exploration-based trajectory optimization for llm agents. *arXiv preprint arXiv:2403.02502*, 2024.
- Linxin Song, Yutong Dai, Viraj Prabhu, Jieyu Zhang, Taiwei Shi, Li Li, Junnan Li, Silvio Savarese, Zeyuan Chen, Jieyu Zhao, et al. Coact-1: Computer-using agents with coding as actions. *arXiv preprint arXiv:2508.03923*, 2025.
- Sizhe Tang, Jiayu Chen, and Tian Lan. Malinzero: Efficient low-dimensional search for mastering complex multi-agent planning. *arXiv preprint arXiv:2511.06142*, 2025.
- Julian Junyan Wang and Victor Xiaoqi Wang. Assessing consistency and reproducibility in the outputs of large language models: Evidence across diverse finance and accounting tasks. *arXiv preprint arXiv:2503.16974*, 2025.
- Xingyao Wang, Hao Yang, Hany Hassan, and Ahmed Awadallah. Executable code actions elicit better llm agents. *arXiv preprint arXiv:2402.01030*, 2024.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.
- Tianbao Xie, Danyang Zhang, Jixuan Chen, François Li, Siheng Zhao, Kunsheng Ruan, Bo Wang, Sarah Nowak, Lisa O’Connell, Maneesh Agrawal, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.
- Bowen Yang, Kaiming Jin, Zhenyu Wu, Zhaoyang Liu, Qiushi Sun, Zehao Li, JingJing Xie, Zhoumianze Liu, Fangzhi Xu, Kanzhi Cheng, et al. Os-symphony: A holistic framework for robust and generalist computer-using agent. *arXiv preprint arXiv:2601.07779*, 2026.
- Yan Yang, Dongxu Li, Yutong Dai, Yuhao Yang, Ziyang Luo, Zirui Zhao, Zhiyuan Hu, Junzhe Huang, Amrita Saha, Zeyuan Chen, et al. Gta1: Gui test-time scaling agent. *arXiv preprint arXiv:2507.05791*, 2025.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.
- Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023.
- King Zhu, Hanhao Li, Siwei Wu, Tianshun Xing, Dehua Ma, Xiangru Tang, Minghao Liu, Jian Yang, Jiaheng Liu, Yuchen Eleanor Jiang, et al. Scaling test-time compute for llm agents. *arXiv preprint arXiv:2506.12928*, 2025.

A Theoretical Proofs

In this appendix, we provide the detailed derivation of the regret bound for Agent Alpha (Theorem 3.1). Our analysis relies on the concentration inequalities for martingales, explicitly leveraging the *unconfirmed knowledge* framework introduced by Kato (2020) to quantify the variance reduction effect of the reflection mechanism.

Alignment with Methodology: As discussed in Section 3.5, while Agent Alpha utilizes a robust max-UCB selection criterion (Eq. 8) for engineering stability, the core efficiency gain stems from the reflection and evaluation mechanism reducing the effective variance of the value estimation. The following proof quantifies this information-theoretic advantage.

A.1 Preliminaries and Definitions

Let $\mathcal{K} = \{1, \dots, K\}$ be the set of actions (arms) at a given expansion step. We denote the true expected value of action a as $\mu_a = \mathbb{E}[X | a]$. Let $a^* = \arg \max_{a \in \mathcal{K}} \mu_a$ be the optimal action with mean μ^* , and for any suboptimal action $a \neq a^*$, let $\Delta_a = \mu^* - \mu_a$ be the optimality gap.

We assume the evaluation scores $X_t \in [0, 1]$. Agent Alpha generates a reflection prior $\hat{\theta}_t$ for each step, serving as a predictor for X_t . The uncertainty of the estimation is governed by the conditional residual variance:

$$\sigma_{\text{res},a}^2 = \mathbb{E} \left[(X_t - \hat{\theta}_t)^2 \mid a_t = a, \mathcal{F}_{t-1} \right]. \quad (11)$$

The comparative evaluator produces scores in $[-1, 1]$ (Section 3.4). We apply the standard affine rescaling $X_t \leftarrow (X_t + 1)/2$ to map scores to $[0, 1]$, ensuring the bounded difference condition $|X_{k,a} - \mu_a| \leq 1$ required by Freedman's Inequality. This rescaling preserves action ordering and the efficiency ratio in Corollary 3.3.

A.2 Proof of Theorem 3.1

The proof proceeds in three steps: establishing the concentration bound via Freedman's Inequality, deriving the sample complexity for suboptimal arms, and summing the regret.

A.2.1 Step 1: Concentration with Residual Variance

Let $S_{n,a} = \sum_{k=1}^n (X_{k,a} - \mu_a)$ be the sum of martingale differences for arm a after n visits. Note that $|X_{k,a} - \mu_a| \leq 1$. Let $V_{n,a}$ be the cumulative conditional variance. Following Kato (2020), the predictable quadratic variation process is bounded by the residual variance due to the presence of the predictor $\hat{\theta}$:

$$V_{n,a} = \sum_{k=1}^n \text{Var}(X_{k,a} \mid \mathcal{F}_{k-1}) \leq \sum_{k=1}^n \mathbb{E}[(X_{k,a} - \hat{\theta}_{k,a})^2 \mid \mathcal{F}_{k-1}] \approx n\sigma_{\text{res},a}^2. \quad (12)$$

Lemma 1 (Freedman's Inequality (Freedman, 1975)). For any real numbers $\epsilon > 0$ and $v > 0$,

$$\mathbb{P}(S_{n,a} \geq \epsilon \cap V_{n,a} \leq v) \leq \exp \left(-\frac{\epsilon^2}{2v + 2\epsilon/3} \right). \quad (13)$$

We seek a confidence width $C_{n,a}$ such that the probability of deviation is bounded by a small δ . Setting the RHS to δ :

$$\exp \left(-\frac{\epsilon^2}{2v + 2\epsilon/3} \right) = \delta \implies \frac{\epsilon^2}{2v + 2\epsilon/3} = \ln(1/\delta). \quad (14)$$

Rearranging terms leads to a quadratic inequality in ϵ :

$$\epsilon^2 - \frac{2}{3} \ln(1/\delta) \epsilon - 2v \ln(1/\delta) = 0. \quad (15)$$

Solving for the positive root ϵ using the quadratic formula:

$$\epsilon = \frac{\frac{2}{3} \ln(1/\delta) + \sqrt{\frac{4}{9} (\ln(1/\delta))^2 + 8v \ln(1/\delta)}}{2} = \frac{\ln(1/\delta)}{3} + \sqrt{\frac{(\ln(1/\delta))^2}{9} + 2v \ln(1/\delta)}. \quad (16)$$

Using the inequality $\sqrt{A+B} \leq \sqrt{A} + \sqrt{B}$, we bound ϵ :

$$\epsilon \leq \frac{\ln(1/\delta)}{3} + \frac{\ln(1/\delta)}{3} + \sqrt{2v \ln(1/\delta)} = \sqrt{2v \ln(1/\delta)} + \frac{2}{3} \ln(1/\delta). \quad (17)$$

Substituting $v \approx n\sigma_{\text{res},a}^2$ and dividing by n to convert the sum $S_{n,a}$ to the mean $\bar{X}_{n,a}$, we obtain the confidence radius $C_{n,a}(\delta) = \epsilon/n$:

$$C_{n,a}(\delta) = \sqrt{\frac{2\sigma_{\text{res},a}^2 \ln(1/\delta)}{n}} + \frac{2 \ln(1/\delta)}{3n}. \quad (18)$$

By setting $\delta = T^{-2}$, we establish that with high probability, $|\bar{X}_{n,a} - \mu_a| \leq C_{n,a}(T^{-2})$.

A.2.2 Step 2: Bounding the Number of Suboptimal Selections

Consider a suboptimal action a with optimality gap $\Delta_a > 0$. The algorithm selects action a at time t only if its Upper Confidence Bound exceeds that of the optimal action a^* . This implies:

$$\bar{X}_{T_a(t),a} + C_{T_a(t),a} \geq \bar{X}_{T_{a^*}(t),a^*} + C_{T_{a^*}(t),a^*}. \quad (19)$$

This condition holds only if at least one of the following three failure events occurs (standard UCT decomposition (Kocsis & Szepesvári, 2006)):

$$E_1 : \bar{X}_{T_{a^*}(t),a^*} \leq \mu^* - C_{T_{a^*}(t),a^*} \quad (\text{Optimal arm under-estimated}) \quad (20)$$

$$E_2 : \bar{X}_{T_a(t),a} \geq \mu_a + C_{T_a(t),a} \quad (\text{Suboptimal arm over-estimated}) \quad (21)$$

$$E_3 : \mu^* < \mu_a + 2C_{T_a(t),a} \quad (\text{True means are close relative to confidence width}) \quad (22)$$

The probabilities of E_1 and E_2 are bounded by $\delta = T^{-2}$ via Step 1. The dominant contribution to regret comes from E_3 , which determines the necessary samples N_a to resolve the gap. E_3 implies:

$$\Delta_a < 2C_{N_a,a} = 2\sqrt{\frac{2\sigma_{\text{res},a}^2 \ln T}{N_a}} + \frac{4 \ln T}{3N_a}. \quad (23)$$

Solving for $\sqrt{N_a}$, we treat this as a quadratic inequality $\Delta_a(\sqrt{N_a})^2 - \alpha\sqrt{N_a} - \beta < 0$. For asymptotic behavior ($T \rightarrow \infty$), the first order term dominates:

$$\sqrt{N_a} < \frac{2\sqrt{2\sigma_{\text{res},a}^2 \ln T}}{\Delta_a}. \quad (24)$$

Squaring provides the bound on the number of pulls:

$$N_a(T) \leq \frac{8\sigma_{\text{res},a}^2 \ln T}{\Delta_a^2} + \Psi(T), \quad (25)$$

where $\Psi(T)$ represents lower-order terms $O(\ln T / \Delta_a)$.

A.2.3 Step 3: Cumulative Regret

The cumulative regret R_T is defined as the sum of gaps weighted by expected pulls:

$$R_T = \sum_{a \neq a^*} \Delta_a \mathbb{E}[N_a(T)]. \quad (26)$$

Substituting Eq. (25):

$$R_T \leq \sum_{a \neq a^*} \Delta_a \left(\frac{8\sigma_{\text{res},a}^2 \ln T}{\Delta_a^2} + \frac{16 \ln T}{3\Delta_a} + 2 \right). \quad (27)$$

(Note: The constant 2 accounts for the finite summation of probabilities of E_1, E_2). Simplifying, we obtain the final bound:

$$R_T \leq \sum_{a \neq a^*} \left(\frac{8\sigma_{\text{res},a}^2 \ln T}{\Delta_a} + \frac{16 \ln T}{3} + 2\Delta_a \right). \quad (28)$$

□

A.3 Proof of Corollary 3.2 (The Order of Regret Bound)

To derive the asymptotic complexity order, we analyze the dominant terms in the regret bound established in Theorem 3.1.

Recall the bound:

$$R_T \leq \sum_{a \in \mathcal{K}, a \neq a^*} \left(\frac{8\sigma_{\text{res},a}^2 \ln T}{\Delta_a} + \frac{16 \ln T}{3} + 2\Delta_a \right). \quad (29)$$

Let $K = |\mathcal{K}|$ be the total number of arms (branching factor). There are $K - 1$ suboptimal arms in the summation. We define the minimal optimality gap $\Delta_{\min} = \min_{a \neq a^*} \Delta_a > 0$ and the maximum expected residual variance $\sigma_{\max}^2 = \max_a \sigma_{\text{res},a}^2$.

We proceed by bounding each term in the summation:

1. Dominant Term (Variance-dependent):

$$\sum_{a \neq a^*} \frac{8\sigma_{\text{res},a}^2 \ln T}{\Delta_a} \leq (K - 1) \frac{8\sigma_{\max}^2 \ln T}{\Delta_{\min}}. \quad (30)$$

2. Secondary Term (Bernstein Correction):

$$\sum_{a \neq a^*} \frac{16 \ln T}{3} = (K - 1) \frac{16 \ln T}{3}. \quad (31)$$

3. Constant Term:

$$\sum_{a \neq a^*} 2\Delta_a \leq 2(K - 1)\Delta_{\max}, \quad \text{where } \Delta_{\max} \leq 1. \quad (32)$$

Combining these, the regret is bounded by:

$$R_T \leq (K - 1) \ln T \left(\frac{8\sigma_{\max}^2}{\Delta_{\min}} + \frac{16}{3} \right) + 2(K - 1). \quad (33)$$

We treat the gap parameters $\Delta_{\min}$ as problem-dependent constants. We focus on the scaling with respect to the horizon T , the branching factor K , and the core variance characteristic σ_{res}^2 (represented here by $\sigma_{\max}^2$).

Dropping the lower-order constant terms and absorbing the numerical constants $(8/\Delta_{\min}, 16/3)$ into the asymptotic notation, we obtain:

$$R_T = \mathcal{O} \left(K \cdot \sigma_{\text{res}}^2 \cdot \ln T \right). \quad (34)$$

This confirms that the regret grows logarithmically with time T , linearly with the action space size K , and linearly with the residual variance σ_{res}^2 . Crucially, if $\sigma_{\text{res}}^2 \rightarrow 0$ (perfect reflection), the constant factor pre-multiplying $\ln T$ vanishes, significantly reducing the regret compared to standard UCT where σ^2 is a fixed constant.

□

A.4 Proof of Corollary 3.3 (Efficiency Gain)

We explicitly compare the asymptotic regret terms. Let R_T^{UCT} be the regret of standard UCT using Hoeffding's inequality, which assumes a variance proxy $\sigma_{\text{raw}}^2 = 1/4$ (bounded range $[0, 1]$):

$$R_T^{\text{UCT}} \approx \sum_{a \neq a^*} \frac{8\sigma_{\text{raw}}^2 \ln T}{\Delta_a}. \quad (35)$$

Let R_T^{Alpha} be the regret derived above using residual variance σ_{res}^2 :

$$R_T^{\text{Alpha}} \approx \sum_{a \neq a^*} \frac{8\sigma_{\text{res},a}^2 \ln T}{\Delta_a}. \quad (36)$$

The efficiency ratio is:

$$\eta_{\text{var}} = \frac{R_{T,\text{var}}^{\text{Alpha}}}{R_{T,\text{var}}^{\text{UCT}}} \approx \frac{\sigma_{\text{res}}^2}{\sigma_{\text{raw}}^2} = \frac{\sigma_{\text{res}}^2}{1/4}. \quad (37)$$

Whenever $\sigma_{\text{res}}^2 < 1/4$, the variance-dependent term is strictly smaller than that of the standard Hoeffding-based UCT baseline. $\square$

B Implementation Details

This appendix provides a comprehensive account of the engineering choices underlying Agent Alpha’s deployment on the OSWorld benchmark. We describe the environment parallelism architecture (§B.1), the state management strategy during tree search (§B.2), the action chunking mechanism and its error handling (§B.3), and our parallel computing strategies (§B.4).

B.1 Environment Parallelism Architecture

A central challenge in applying tree search to GUI agents is that real-world operating system environments are *stateful and non-reversible*: one cannot “undo” a file deletion or a browser navigation by simply restoring a tensor. We address this through a **multi-VM environment pool** architecture.

VM Instance Pool. At initialization, Agent Alpha provisions a pool of K independent virtual machine (VM) instances in addition to the primary (main) environment. Each VM is a fully isolated OS instance (Ubuntu desktop) running on cloud infrastructure (e.g., AWS EC2). The pool size K is set equal to the branching factor b (i.e., `num_actions_per_expansion`), so that every candidate action generated during expansion can be executed on a dedicated VM without cross-contamination.

Concretely, when the system starts, the `create_expand_env_pool()` function instantiates K new `DesktopEnv` objects, each backed by its own cloud VM instance:

- Each sub-environment receives `path_to_vm=None`, causing the provider to allocate a fresh VM from the cloud’s instance pool (via snapshot/AMI restore).
- All VMs share the same OS image, screen resolution, and configuration as the main environment, ensuring behavioral consistency.
- The pool is created once per worker process and **reused across all tasks** in the evaluation suite, amortizing the VM startup cost over the entire benchmark.

Pool Management. The environment pool is managed through a thread-safe acquire/release protocol:

- **Acquire:** Before each expansion, the searcher acquires b environments from the pool. Each acquired environment is tagged with a `current_node_id` indicating which tree node state it currently holds. The acquire logic preferentially returns environments already synchronized to the target node, avoiding redundant replays.
- **Release:** After expansion and judging are complete, environments are returned to the pool with updated node-state tags.
- **Thread Safety:** A `threading.Lock` guards all pool mutations, ensuring correctness when expansion and judging operations overlap across threads.

How Isolation Is Achieved. Unlike game environments that support copy-on-write state cloning, a full OS desktop cannot be cheaply duplicated at runtime. Our isolation relies on:

1. **Separate VM instances:** Each sub-environment runs on its own VM. Actions executed on VM_i have no effect on VM_j .
2. **Snapshot-based reset:** Returning to a prior state is achieved by reverting the VM to a cloud snapshot (AMI) and replaying the action prefix. This is functionally equivalent to a full state rollback, at the cost of replay time.
3. **Application-level backup:** Before search begins, critical application state (browser profiles, config directories, user data folders) is backed up within each VM. A “soft reset” restores these backups, kills relevant processes, and re-runs the task’s setup script—avoiding the full snapshot revert when possible.

B.2 State Management During Tree Search

The Replay Paradigm. Since OS environments are non-reversible, Agent Alpha uses a **reset-and-replay** paradigm to navigate the search tree. At each MCTS iteration, after the SELECT phase identifies a leaf node, the system must bring the environments to that leaf’s state before expanding. This is accomplished as follows:

1. **Reset:** Each environment (main + all sub-environments) is reverted to the initial task state via `env.reset(task_config=...)`, which triggers a VM snapshot restore.
2. **Replay:** The complete action sequence from the root to the selected leaf node is re-executed in each environment. Actions are stored in grounded pyautogui format (e.g., `pyautogui.click(635, 697)`), so replaying is deterministic and does not require re-grounding from natural language.
3. **Observation Capture:** After replay completes, a fresh screenshot is captured from the main environment as the leaf node’s observation. This screenshot—not a cached one—is used for subsequent expansion and judging, ensuring the search operates on ground-truth visual state.

Parallel Replay. A naive sequential approach would reset and replay each VM one after another, creating a bottleneck proportional to $(K + 1) \times T_{\text{replay}}$. Instead, Agent Alpha performs **parallel replay**: the main environment and all K sub-environments execute their reset-and-replay sequences concurrently using a `ThreadPoolExecutor`. Since each VM is an independent networked instance, these operations are truly parallel (not merely concurrent), reducing the effective replay time to approximately $\max(T_{\text{replay},i})$ rather than $\sum_i T_{\text{replay},i}$.

Lazy Synchronization. Each sub-environment maintains a `current_node_id` tag. When the search revisits a subtree, environments already at the target node state can skip the reset-and-replay step entirely. This optimization is particularly effective when MCTS exploits a promising branch repeatedly, as the sub-environments remain synchronized from the previous iteration.

Observation Handling in the Search Tree. Given that “rolling back” an OS state is infeasible, Agent Alpha obtains observations (screenshots) during search by:

- **Before expansion:** The parent node’s screenshot is obtained by replaying to the parent state in the main environment. This is a *live* screenshot, not a cached approximation.
- **During expansion:** Each candidate action is executed on its dedicated sub-environment (which has been synchronized to the parent state). The resulting *after-screenshot* is captured from that sub-environment. This pair of (before, after) screenshots is fed to the comparative judge.
- **Tree storage:** Each node stores its observation (screenshot) in memory. For aggregate (chunked) nodes, per-step intermediate screenshots are also stored to support full-trajectory reflection.

Thus, every observation in Equations (4) and (5) of the main paper corresponds to a *real screenshot captured from a live VM* at the appropriate state—not a simulated or predicted image.

B.3 Action Chunking Mechanism

Mechanism. The action chunking (or node aggregation) feature groups C consecutive action cycles into a single tree node, where C is controlled by the `node_aggregation_count` parameter. Within one chunked node, the agent executes C rounds of the reflection→plan→execute cycle on the same sub-environment, producing C grounded actions. All C actions are stored in the node’s `aggregated_actions` list, and intermediate screenshots are preserved in `aggregated_screenshots` for downstream reflection.

Error Handling and Early Termination. Each chunked execution runs C cycles of the reflection→plan→execute loop sequentially on the same sub-environment. The error handling behavior depends on which phase fails:

- **Action execution failure** (e.g., `env.step()` throws an exception): The exception is caught and the cycle continues with the *previous* observation as a fallback. The action is still recorded in the trajectory. In other words, a single failed execution does not abort the entire chunk—the agent proceeds to the next cycle using the last valid screenshot as input. This design prioritizes robustness: a transient UI glitch at step k does not discard the progress of steps 1 through $k-1$.
- **Environment signals done=True**: If the environment returns a terminal signal (e.g., due to a `TASK_SUCCEED` or `FAIL` action), execution halts immediately via a break. The partial trajectory (actions 1 through k) and corresponding observations are returned.
- **TASK_SUCCEED detection**: When the agent’s generated action is `TASK_SUCCEED` at any step within the chunk, the MCTS searcher detects this during expansion and immediately returns a success trajectory, terminating the search.

There is no explicit “state rollback” mechanism within a chunk, because each chunk executes on a dedicated sub-environment that is discarded (or reassigned) after judging. The parent node’s state remains accessible through the reset-and-replay mechanism described in §B.2. This means that if a chunk produces a poor trajectory, it does not corrupt the search: the tree simply assigns a low value to that node, and subsequent iterations explore alternative branches from the parent.

Chunk Size Selection. The chunk size C is set as a hyperparameter. In the comparison experiments (Table 3), the default configuration uses $C = 1$ (no chunking) with expansion factor $N = 5$ and 20 MCTS iterations. For complex, long-horizon tasks (such as Chrome and Multi-apps), we increase the chunk size to $C = 5$ and reduce iterations to 15 to balance efficiency (as noted in §4.2 of the main paper). As shown in Figure 3(d), increasing chunk size to 5 improves performance on long-horizon domains by enabling the agent to plan macro-actions, but further increases (e.g., $C = 7$) cause performance degradation due to loss of granular control and increased difficulty in error recovery when too many actions are committed without intermediate evaluation feedback. We acknowledge that dynamic chunk size selection—e.g., adapting C based on task complexity or search progress—is a promising direction for future work.

B.4 Parallel Computing Strategies

Agent Alpha employs parallelism at three distinct levels:

Level 1: Task-Level Parallelism. Multiple worker processes run concurrently, each handling a different task from the evaluation suite. Each worker owns its own main environment and sub-environment pool. Tasks are distributed via a shared queue (`multiprocessing.Queue`), and results are collected into a shared score list.

Level 2: Intra-Iteration VM Parallelism. Within a single MCTS iteration, the most time-consuming operations—reset, replay, action execution, and evaluation—are parallelized across the $K + 1$ VM instances:

- **Parallel reset-and-replay** (`_parallel_replay_to_node`): The main environment and all K sub-environments simultaneously reset to the initial state and replay the action prefix to reach the selected leaf node. This uses a `ThreadPoolExecutor` with $K + 1$ workers.
- **Parallel expansion** (`expand_new`): Each of the b candidate action sequences is executed on its own sub-environment concurrently. The agent’s reflection-plan-execute cycles run in parallel threads, each operating on an isolated environment.

- **Parallel judging** (`_score_sibling_nodes_parallel`): After expansion, the comparative judge collects after-screenshots from all b sub-environments in parallel, then issues a single comparative scoring call (or b parallel independent scoring calls).

Level 3: LLM Call Parallelism. Within the expansion phase, b candidate actions are generated by b parallel calls to the planner LLM. Each thread restores the agent’s internal state from a snapshot taken at the start of expansion, ensuring that sibling candidates are generated from identical context (no cross-contamination from one candidate’s reflection polluting another’s generation). This snapshot-and-restore mechanism guarantees that the b candidates are independently sampled despite sharing the same underlying agent object.

Agent State Isolation. A subtle but critical implementation detail: the agent (Worker) maintains mutable internal state including turn count, action history, reflection buffers, and generator/reflection agent message histories. Naively sharing this state across parallel expansion threads would cause candidate actions to interfere with each other. Agent Alpha addresses this through a **snapshot-restore protocol**:

1. Before expansion begins, the agent’s full mutable state is deep-copied into a snapshot.
2. Before each candidate’s generation, the snapshot is restored, ensuring all candidates start from identical context.
3. After expansion completes, the snapshot is restored one final time, preventing expansion side-effects from leaking into subsequent search iterations.

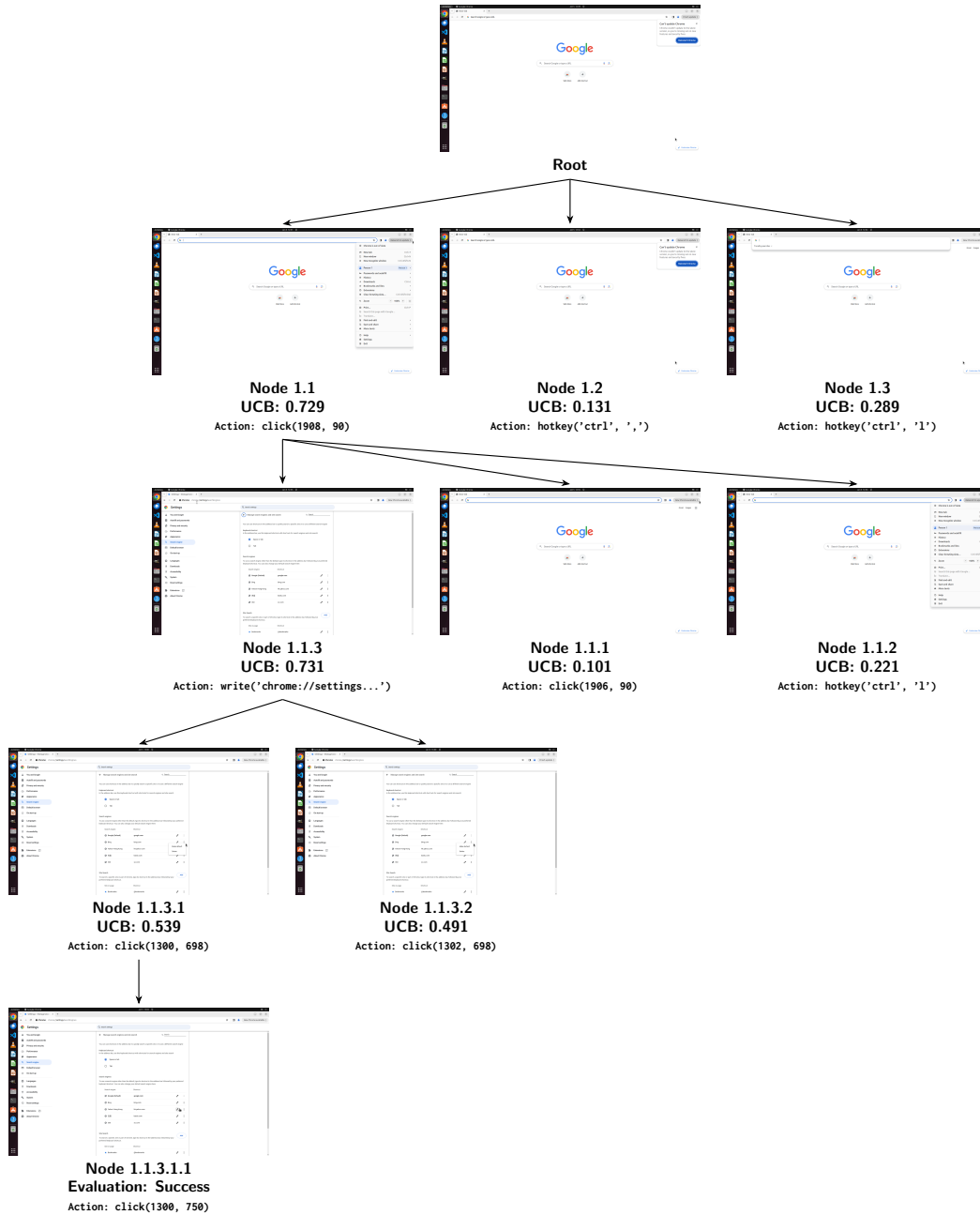
C Case Studies

C.1 Task1: Can you make Bing the main search engine when I look stuff up on the internet?

For Task 1 (a relatively simple task in the Chrome domain), the hyperparameters were configured as follows: max MCTS iterations = 20, expansion factor = 3, and action chunking = 1. The search process successfully located the solution at depth 5.

C.2 Task2: Help me export the first image from the doc file attached in the most recent email in Notes folder, and set this image as the new desktop background.

In Task 2, a long-horizon task involving multi-app operations, we set the action chunking = 5, the maximum MCTS iterations = 15, and the expansion factor = 5. The generated search tree is illustrated in the following figure, while the subsequent figure displays screenshots of the successful path.



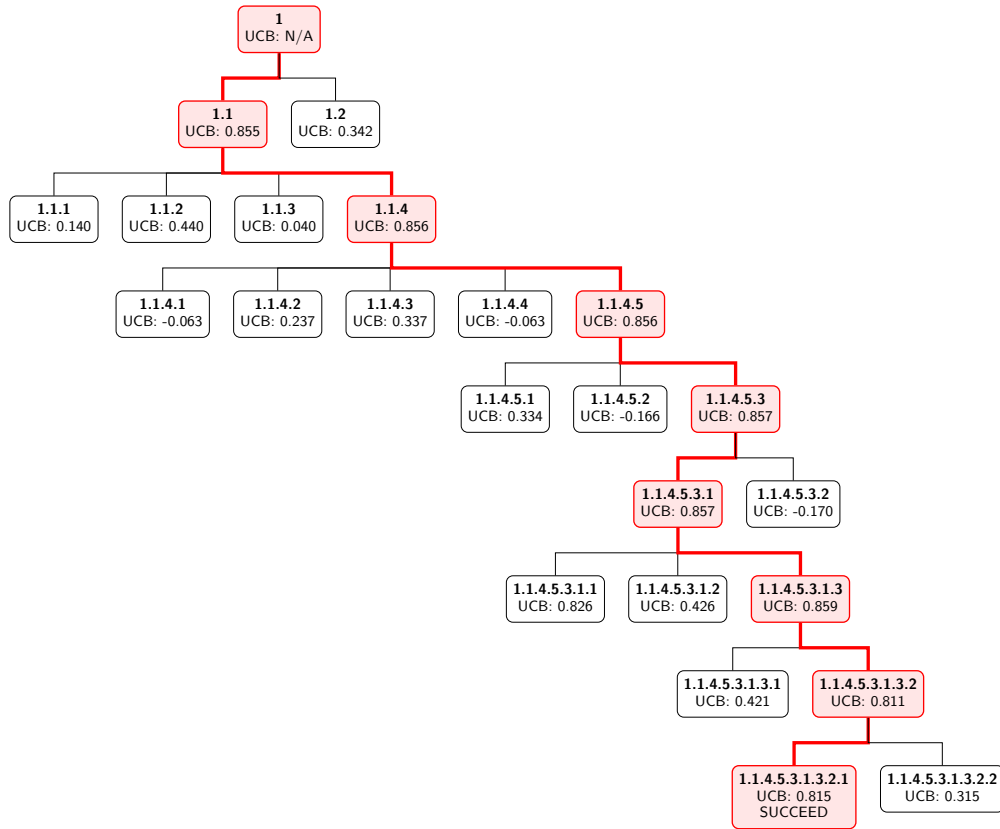


Figure 4: MCTS Tree Structure for Task2(Optimal Path Highlighted)

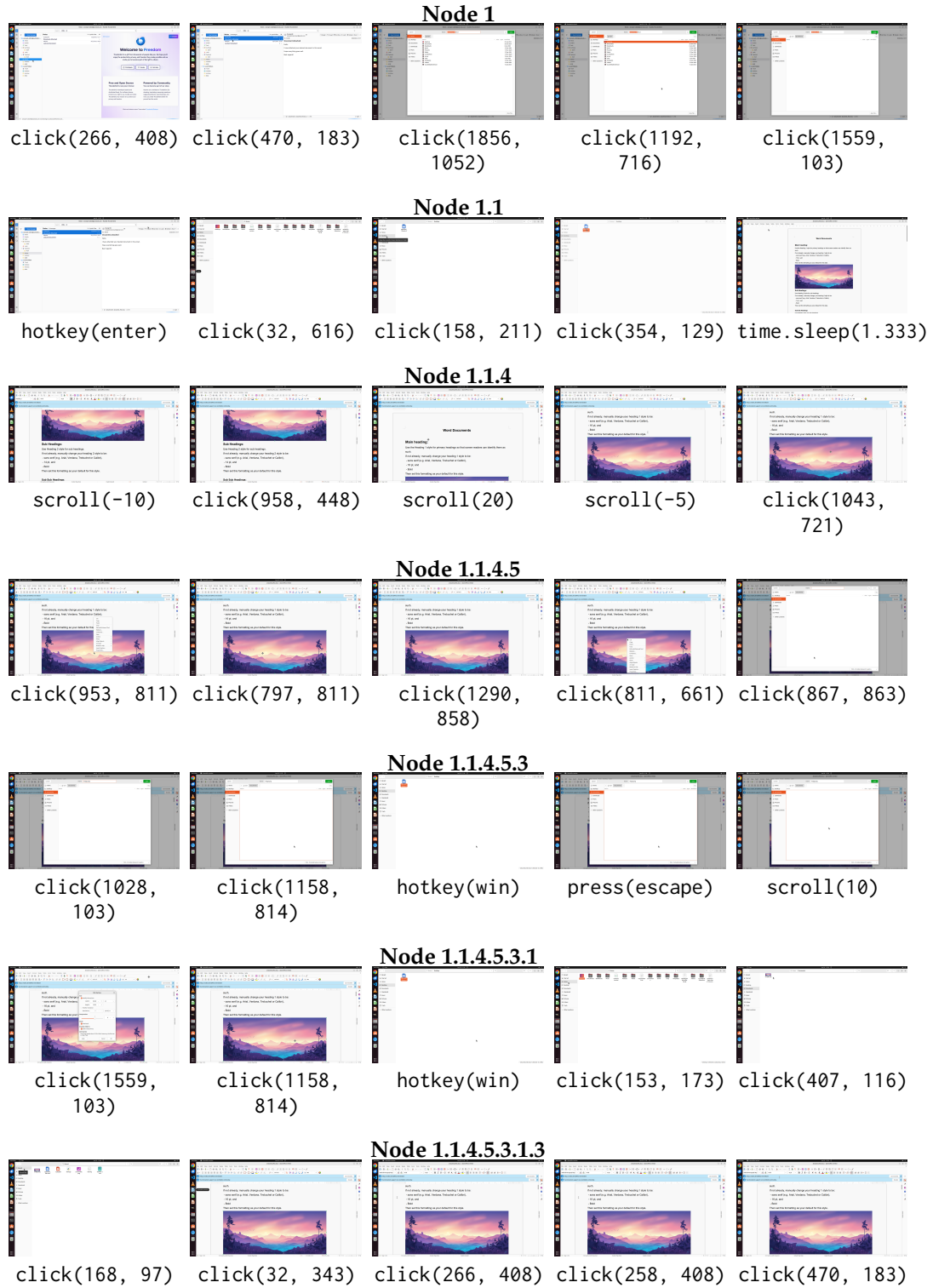


Figure 5: Trajectory Visualization of Task2: Part 1

Table 5: Per-task inference efficiency under a matched GPT-5-mini backbone. Agent S3 executes $N = 10$ rollouts in parallel; its LLM/API calls and tokens are totals across all rollouts.

Method	Time (s)	LLM/ API Calls	Tokens
CoAct-1 (single-pass)	602	22	0.33M
OS-Symphony (single-pass)	894	51	0.79M
Agent S3 (bBoN, $N = 10$)	661	380	3.10M
Agent Alpha (ours, MCTS)	1447	157	2.17M

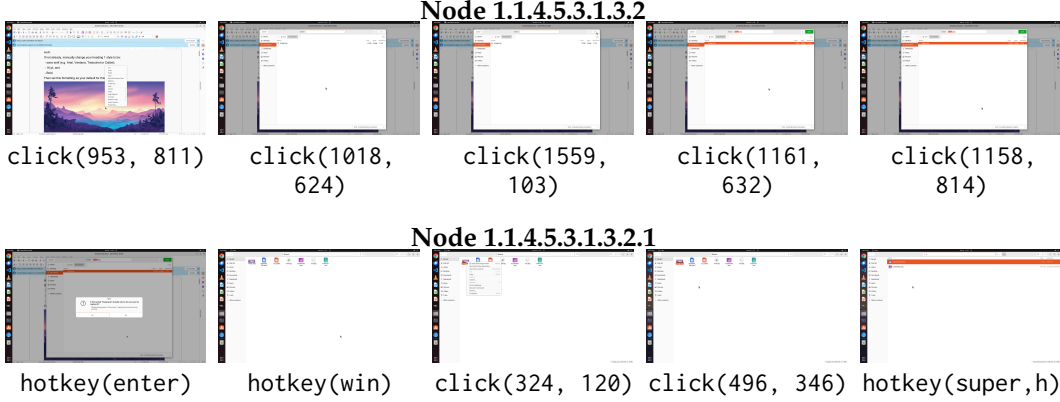


Figure 6: Trajectory Visualization of Task2: Part 2

D Inference-Time Efficiency

Since Agent Alpha is an inference-time scaling method, we further evaluate its computational efficiency in terms of wall-clock time, the number of LLM/API calls, and token usage. We compare Agent Alpha with three representative baselines covering both inference paradigms: Agent S3 with best-of- N scaling ($N = 10$), and two single-pass agentic frameworks, CoAct-1 and OS-Symphony.

To isolate the computational cost of the inference algorithm from that of a stronger backbone, all methods use the same GPT-5-mini model as the unified policy and grounding model, while retaining their respective standard search configurations. For Agent S3, the $N = 10$ rollouts are executed in parallel. Therefore, its reported wall-clock time corresponds to the parallel execution latency, whereas its LLM/API calls and token usage are accumulated over all ten rollouts. Agent Alpha is evaluated with sequential single-machine execution. Table 5 reports the per-task averages, and Tables 6–8 provide the corresponding per-domain breakdowns.

As expected, inference-time scaling methods consume more total compute than single-pass agents in exchange for higher task-completion capability. Among the inference-time scaling approaches, however, Agent Alpha requires substantially less total inference compute than Agent S3. Specifically, Agent Alpha uses 157 LLM/API calls and 2.17M tokens per task on average, compared with 380 calls and 3.10M tokens for Agent S3, corresponding to approximately $2.4\times$ fewer calls and $1.4\times$ fewer tokens. This efficiency gain follows from the structured search mechanism in Agent Alpha: unpromising branches can be terminated early, while successful action prefixes are reused across subsequent search iterations. In contrast, best-of- N executes multiple independent trajectories without cross-trajectory prefix reuse.

Agent S3 nevertheless exhibits lower observed wall-clock latency (661 s vs. 1447 s), since its ten rollouts are executed in parallel, whereas the reported Agent Alpha runtime corresponds to sequential single-machine execution. Parallel execution reduces latency but does not

reduce the total number of model calls or tokens consumed. Agent Alpha also supports action- and environment-level parallelism, as analyzed in the ablation study in the main paper.

Table 6: Per-domain wall-clock time per task in seconds. Agent S3 executes $N = 10$ rollouts in parallel, while Agent Alpha is measured under sequential single-machine execution.

Method	Chr.	Calc	Imp.	Wri.	GIMP	VSC	Multi	Thu.	OS	VLC	Avg.
Agent S3 ($N = 10$)	979	741	608	924	350	495	402	775	581	866	661
OS-Symphony	859	1291	472	1002	787	1090	251	728	875	1423	894
CoAct-1	319	405	846	1130	1413	133	272	2113	262	634	602
Agent Alpha	1435	798	3435	3583	995	238	647	1438	743	346	1447

Table 7: Per-domain number of LLM/API calls per task. For Agent S3, the reported values are accumulated across all $N = 10$ rollouts.

Method	Chr.	Calc	Imp.	Wri.	GIMP	VSC	Multi	Thu.	OS	VLC	Avg.
Agent S3 ($N = 10$)	627	460	450	397	210	300	185	530	360	430	380
OS-Symphony	53	64	36	63	49	63	13	52	39	80	51
CoAct-1	27	10	24	24	69	7	11	105	11	17	22
Agent Alpha	200	95	361	359	75	45	63	185	99	40	157

Table 8: Per-domain total token usage per task. For Agent S3, token usage is accumulated across all $N = 10$ rollouts. All methods use the same GPT-5-mini backbone.

Method	Chr.	Calc	Imp.	Wri.	GIMP	VSC	Multi	Thu.	OS	VLC	Avg.
Agent S3 ($N = 10$)	7.77M	3.61M	3.29M	3.40M	1.15M	1.57M	1.12M	4.34M	2.46M	3.55M	3.10M
OS-Symphony	0.84M	1.09M	0.49M	0.98M	0.82M	0.99M	0.12M	0.84M	0.52M	1.23M	0.79M
CoAct-1	0.23M	0.13M	0.57M	0.73M	0.77M	0.05M	0.12M	1.61M	0.09M	0.30M	0.33M
Agent Alpha	3.01M	1.26M	5.18M	5.80M	0.70M	0.22M	0.63M	2.59M	0.65M	0.94M	2.17M

E System Prompts

User Input

Instruction: Can you help me change the default search engine from Chrome to Bing?

Prompt for Reflection Generation

REFLECTION ON TRAJECTORY: You are an expert computer use agent designed to reflect on the trajectory of a task and provide feedback on what has happened so far.

You have access to the Task Description and the Current Trajectory of another computer agent. The Current Trajectory is a sequence of a desktop image, chain-of-thought reasoning, and a desktop action for each time step. The last image is the screen's display after the last action. **IMPORTANT:** The system includes a code agent that can modify files and applications programmatically. When you see:

- Files with different content than expected
- Applications being closed and reopened
- Documents with fewer lines or modified content

These may be **LEGITIMATE** results of code agent execution, not errors or corruption.

Your task is to generate a reflection. Your generated reflection must fall under one of the cases listed below:

Case 1. The trajectory is not going according to plan. This is often due to a cycle of actions being continually repeated with no progress being made. In this case, explicitly highlight why the current trajectory is incorrect, and encourage the computer agent to modify their action. However, **DO NOT** encourage a specific action in particular.

Case 2. The trajectory is going according to plan. In this case, simply tell the agent to continue proceeding as planned. **DO NOT** encourage a specific action in particular.

Case 3. You believe the current task has been completed. In this case, tell the agent that the task has been successfully completed.

To be successful, you must follow the rules below:

- ****Your output MUST be based on one of the case options above**.**
- **DO NOT** suggest any specific future plans or actions. Your only goal is to provide a reflection, not an actual plan or action.
- Any response that falls under Case 1 should explain why the trajectory is not going according to plan. You should especially lookout for cycles of actions that are continually repeated with no progress.
- Any response that falls under Case 2 should be concise, since you just need to affirm the agent to continue with the current trajectory.
- **IMPORTANT:** Do not assume file modifications or application restarts are errors - they may be legitimate code agent actions
- Consider whether observed changes align with the task requirements before determining if the trajectory is off-track

Main prompt for Planning Generation

You are an expert in graphical user interfaces and Python code. You are responsible for executing the task: 'TASK_DESCRIPTION'. You are working in CURRENT_OS.

GUIDELINES

Agent Usage Guidelines

You have access to both GUI and code agents. Choose the appropriate agent based on the task requirements:

GUI Agent

- **Use for**: clicking, typing, navigation, file operations, tasks requiring specific application features...

Code Agent

code.agent.section

Visual Confirmation for Evaluation

The evaluator judges success based on what is **VISIBLY** shown on the screen in the final screenshot.

- If you modified an image, you **MUST** open it in an image viewer to show the result.

- **Invisible changes count as failures.**

Environment-Instruction Alignment Analysis

Before planning your first action, analyze the relationship between the User Instruction and the Current Environment State.

1. **Initial State Check**: What application or website is currently open?

2. **Constraint Prediction**: Do **NOT** navigate away from the initial website unless explicitly asked.

3. **Action Alignment**: Use the search bar *of the current website*, not the browser's address bar.

Evaluation Awareness & Success Criteria Analysis

The evaluator for this task is strict and programmatic. It often checks for specific URL parameters, file names, or HTML elements.

Pre-Completion Checklist

Before generating 'agent.done()', you **MUST** complete this checklist:

1. **Task Requirements Check**: Read the original task instruction word-for-word.

2. **Visual Proof Check**: Is the result clearly visible in the current screenshot?

3. **File Name Check**: If the task specified a filename, verify it **EXACTLY** matches.

4. **Core Task Verification**: Are you solving the **RIGHT** task?

State Stagnation & Loop Detection (CRITICAL)

- **Compare Screenshots**: Look closely at the current screenshot versus the previous step's screenshot.

- **Identical State = Action Failed**: If the screen hasn't changed, your last action was **INEFFECTIVE**.

- **DO NOT REPEAT**: Never repeat an action that just failed.

END OF GUIDELINES

You are provided with:

1. A screenshot of the current time step.

2. The history of your previous interactions with the UI.

3. Access to the following class and methods to interact with the UI:

class Agent:

Your response should be formatted like this:

(Previous action verification)

Carefully analyze based on the screenshot if the previous action was successful.

(Screenshot Analysis)

Closely examine and describe the current state of the desktop.

(Next Action)

Based on the current screenshot and the history, decide on the next action in natural language.

(Grounded Action)

Translate the next action into code using the provided API methods:

```python

agent.click("The menu button at the top right of the window", 1, "left") ```

#### Dynamically added prompt for Planning Generation

CRITICAL REFLECTION - YOU MUST READ AND ACT ON THIS

{reflection}

MANDATORY INSTRUCTIONS BASED ON ABOVE REFLECTION:

1. If reflection says "NOT going according to plan" → You MUST try a DIFFERENT action
2. If reflection identifies a problem → You MUST address that specific problem
3. DO NOT repeat the same action if it has failed before
4. If clicking hasn't worked → Try scrolling, keyboard shortcuts, or different UI elements
5. Explore alternative paths: search boxes, filters, side panels, menus
6. Read the explanation(if available) of button or clickable element in the screenshot carefully.
7. Ignore options/settings that are NOT related to the instruction. Only focus on satisfying the specific conditions mentioned in the instruction.
8. If the task is about Chrome, DO NOT use the code agent. And do the task on initial webpage.
9. If you can't find the exact option/setting, try to find the similar or related option/setting that might reach the goal.
10. If the option list doesn't shows the target option in the first page, try to scroll the list to look for it.

#### Dynamically added prompt for Planning Generation(deep search alert)

DEEP SEARCH ALERT - CONSIDER ALTERNATIVE APPROACH

Current {depth.label}: {depth.value} (threshold exceeded: > 8) {iter\_line}

You have been exploring this path for a long time without success. This often indicates:

- The GUI approach is too complex or unreliable for this task
- The task might be better solved with code/command-line tools
- You are stuck in unproductive GUI navigation

MANDATORY ACTION REQUIRED: If you have NOT already tried the code agent for this task, you MUST try it now:

1. **\*\*For application settings/config tasks\*\***: Use agent.call\_code\_agent() to edit config files
2. **\*\*For file/data manipulation\*\***: Use agent.call\_code\_agent() for text editing, file processing
3. **\*\*For extension installation\*\***: Use code -install-extension instead of GUI navigation
4. **\*\*EXCEPTION FOR CHROME TASKS\*\***: For Chrome-related tasks, DO NOT use the code agent.

DO NOT continue with GUI operations if coding agent could solve this more reliably.

#### Dynamically added prompt for Planning Generation(task infeasible alert)

CRITICAL: TASK MAY BE INFEASIBLE

Current {depth.label}: {depth.value} (CRITICAL THRESHOLD: 17)

You have explored this task for an EXTREMELY long time without success. This is a STRONG indicator that the task may be IMPOSSIBLE or INFEASIBLE.

CRITICAL ANALYSIS REQUIRED - Ask yourself:

1. **\*\*Application Capability Check\*\***:

- Does this application have the required feature?

Examples: VS Code cannot open multiple workspaces in one window GIMP cannot edit/trim videos

2. **\*\*Fundamental Constraint Check\*\***:

- Is there a fundamental limitation preventing this task?

- Have you tried multiple completely different approaches?

3. **\*\*Repeated Failure Pattern\*\***:

- Are you stuck in a cycle trying the same approaches?

IF YOU DETERMINE THE TASK MAYBE IMPOSSIBLE: You MUST call agent.fail() immediately.

**Prompt for Comparative Evaluation(fact generation)**

**BEHAVIOR\_NARRATOR\_SYSTEM\_PROMPT:** You are an expert in computer usage responsible for analyzing what happened after a computer action is taken.

**\*\*Reasoning Guidelines:\*\***

You will analyze the before and after screenshots given an action and provide a clear summary of the changes observed. Some things to note:

- Pay attention to any circular visual markers that may suggest where clicks, mouse movements, or drags occurred.
  - Clicks will be marked with a red circle and labeled Click
  - Moving the mouse without clicking will be marked with a blue circle and labeled MoveTo
  - Drag and drops will have an initial blue circle labeled MoveTo, a green circle labeled DragTo, and a green line connecting the two circles.
  - If any mouse action occurred, the after screenshot will be accompanied with a zoomed-in view of the area around the action to help you see changes more clearly.
  - Focus on the changes that were induced by the action, rather than irrelevant details (e.g. the time change in the system clock).
  - Note that even if the action is expected to cause a change, it may have not. Never assume that the action was successful without clear evidence in the screenshots.
  - Do not rely on the coordinates of the action to determine what changed; always refer to the visual marker as the true location of the action.
  - Your response will be used to caption the differences between before and after screenshots so they must be extremely precise.
- Please format your response as follows below.

(thoughts)

[Your detailed reasoning about the before screenshot and any visual markers, the action being taken, and the changes in the after screenshot and zoomed-in view (if present).]

(thoughts)

(answer)

[An unordered list of the relevant changes induced by the action]

(answer)

### Prompt for Comparative Evaluation(Comparative Scoring Prompt)

You are evaluating {num.trajectories} different action sequences to determine which one makes the most progress toward completing this task:

**\*\*Task:\*\*** {instruction}

**\*\*CRITICAL:** Show, Don't Tell

You must base your judgment ONLY on the provided screenshots and explicit text output visible in them. Do NOT assume the task is done just because the agent says so (or because the facts claim so). If the agent ran a verification command (like ls or cat), you MUST see the output in the terminal screenshot confirming the result. If the proof is not visible, do not give a high score.

**\*\*Evaluation Criteria:\*\***

1. Does the sequence move closer to completing the task?
2. Are the actions relevant and purposeful?
3. Does the final state show VISIBLE progress toward the goal?

**\*\*Differentiation Requirement:\*\***

For nodes that appear very similar (in terms of action content, trajectory history, and produced results), please **\*\*carefully compare their subtle differences\*\***. Look for efficiency, unnecessary steps, or minor visual improvements. **\*\*You must assign differentiated scores based on this careful analysis.\*\*** Avoid giving identical scores unless the trajectories are absolutely indistinguishable. Ideally, the scores should clearly reflect which trajectory is better, even if only slightly.

You will be provided:

- Initial screenshot (starting state)
- For each trajectory: fact captions describing what changed + final screenshot

**\*\*Output Format:\*\***

Provide scores from -1.0 to 1.0 for each trajectory, where:

- 1.0 = Completes or nearly completes the task
- 0.5 to 0.9 = Makes significant progress
- 0 to 0.5 = Makes some progress
- -0.5 to 0 = Little or no progress / wrong direction
- -1.0 to -0.5 = Harmful or completely wrong

Return ONLY a JSON array of scores, e.g., [0.8, -0.5, -0.6]