MULTI-AGENT COMPUTER USE

Anonymous authors
Paper under double-blind review

ABSTRACT

Computer use agents (CUAs) today are primarily deployed as single serial agents. This setup is suboptimal for complex long-horizon tasks that benefit from task decomposition, parallel execution, and consistent re-planning based on new information. In this paper, we argue that we should instead move towards evaluating and building *multi-agent computer use* (MACU) systems. These systems, which emphasize planning and parallel execution, alleviate many of the shortcomings of single-agent CUAs. We propose a general multi-agent setup in which a manager model decomposes computer use tasks as a directed acyclic graph (DAG) of subtasks, encoding relevant dependencies and goals for subagents. At each iteration, the manager dispatches parallel CUA subagents to carry out nodes on the ready frontier of the DAG, and continuously revises the DAG (adding, canceling, or rewriting nodes) as new findings arrive from subagents. This design treats the partially observable environment of computer use as a first class challenge: information that downstream agents may not be able to re-observe are retained and passed forward through the manager and DAG structure. We demonstrate that MACU improves over strong single-agent baselines by 3.4–25.5% on four computer use benchmarks, across desktop (OSWorld) and web navigation (Online-Mind2Web, WebTailBench, Odysseys) environments, with the largest and most significant gains on multi-step and long-horizon web tasks, and exhibits more favorable test-time scaling. On Odysseys, a long-horizon web navigation benchmark, MACU solves $4\times$ as many tasks as the single-agent baseline under the same per-task time limit ($8.5\% \rightarrow 34.0\%$), and matches the final single-agent success rate $4.4\times$ sooner in end-to-end wall-clock time. Our findings highlight that multi-agent coordination is a promising axis for scaling computer use agents to work productively and more effectively for longer time horizons. We release all code and interactive visualizations at `removed_for_review`.

1 INTRODUCTION

Computer use is the task of interacting with software through the use of Graphical User Interfaces (GUIs). Computer use agents (CUAs) today are primarily deployed and evaluated as single serial agents, but this is suboptimal for tackling complex long-horizon tasks, which benefit from decomposition, parallel execution, and consistent re-planning based on new information. In this paper, we argue that we should move towards *multi-agent computer use* (MACU): systems which emphasize planning and parallel execution, and are promising ways to alleviate the shortcomings of single-agent CUAs.

Towards this goal, we present a general MACU setup (Fig. 1), where a manager model plans, decomposes, and continuously iterates over a directed acyclic graph (DAG) whose nodes are subtasks that are executed in parallel by computer use subagents. The graph decomposition schema used in MACU is open-ended and the manager operates on a *graph* of subagents, with the ability to add, cancel, rewire, or modify the instructions of pending subtasks. This directly targets long-horizon tasks and complex but parallelizable CUA tasks, key regimes where serial CUA setups struggle. Our work distinguishes itself from prior work in several ways. Most prior multi-agent work based on large language models (LLMs) explores text-based specialized subagents (e.g., tool use (Chen et al., 2023; Li et al., 2025), coding agents (Hong et al., 2023; Geng & Neubig, 2026), math (Qian et al., 2025; Zhao et al., 2025; Qi et al., 2025), and science (Lu et al., 2024; Gottweis et al., 2025)) in effectively fully observable settings. Computer use is a qualitatively different regime. The agent acts on a live operating system, whose state: filesystem, background processes, open browser tabs,

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

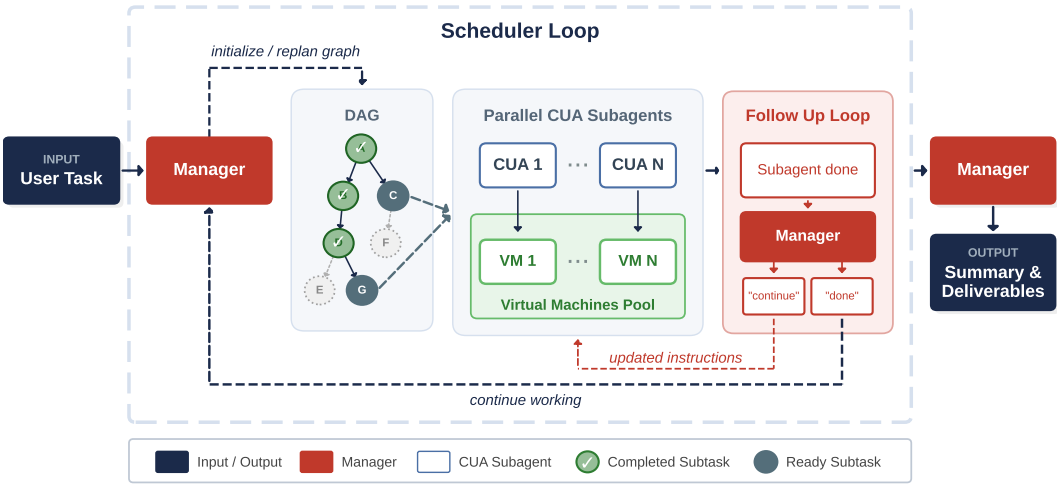


Figure 1: Overview of the setup for multi-agent computer use (MACU). At each iteration, ready subtasks (grey) are launched and handled in parallel by CUA subagents. The manager replans and updates the DAG based on information it receives from completed subagents.

login cookies, dynamic content, is only *partially observable* at any given moment, and the state relevant to a downstream subtask often cannot be re-observed once the original subtask has ended. We treat partial observability as a first-class consideration in MACU, which leads us to be explicit about the initial plan being a best-guess hypothesis, which the manager needs to continuously revise once new information is collected from pending subagents. In contrast to many prior approaches on orchestrating multiple subagents (Agashe et al., 2025; Lee et al., 2026; Zhang et al., 2025b), our proposed MACU approach is also substantially simpler and does not rely on specialized models. All subagents are identical, based off the same CUA backbone, and carry out instructions that are produced by the manager, allowing our approach to directly benefit from future advances on CUA models by directly leveraging them as subagents in MACU.

We demonstrate empirically that MACU is a promising approach to tackling computer use tasks. MACU improves over strong single-agent baselines across all benchmarks we evaluate, increasing success rates by 4.7% on OSWorld (Xie et al., 2024), 3.4% on Online-Mind2Web (Xue et al., 2025), 8.7% on WebTailBench-v2 (Awadallah et al., 2025), and 25.5% on Odysseys (Jang et al., 2026). MACU gains are largest (and statistically significant) on multi-step and long-horizon tasks, persist when the manager uses an identical model as the subagents, and hold under matched action and wall-clock budgets. Our analysis also shows that the initial decomposition, forwarding of findings between subagents, and adaptive replanning of the DAG each account for a substantial share of the long-horizon tasks that only MACU solves. We believe these findings provide a strong foundation for further work on effective scaling of multi-agent computer use systems.

2 Related Work

Computer Use Agents Several benchmarks have been established to measure the capabilities of frontier models on computer use tasks. OSWorld (Xie et al., 2024) established an execution-based grading of 369 open ended Ubuntu tasks spanning various native apps and multi-application workflows. WindowsAgentArena (Bonatti et al., 2025) adapts the OSWorld framework to 150+ tasks on a real Windows operating system. More recently, Gym-Anything (Aggarwal et al., 2026) leveraged coding and audit agents to automatically install and configure real software. Other benchmarks have focused on web navigation capabilities: WebArena (Zhou et al., 2024) provides a simulated environment of realistic self-hosted sites (Shopping, Reddit, GitLab, etc.) with execution-based grading, and VisualWebArena (Koh et al., 2024) extends it with visually grounded multimodal tasks. WebVoyager (He et al., 2024) evaluates CUAs directly on 15 real world websites, and WebLINX (Lù et al., 2024) evaluates multi-turn dialogue for web navigation. Online-Mind2Web (Xue et al., 2025) evaluates agents with a more robust judge, across 300 tasks on 136 live websites, and WebTailBench (Awadallah et al., 2025) proposes 609 tasks that include multi-step and cross-site tasks, drawn from high-traffic webpages. Odysseys (Jang et al., 2026) recently proposed 200 long-

horizon web navigation tasks based on real human browsing patterns. On the capabilities front, frontier models have quickly improved on computer use, saturating many of these benchmarks. GPT-5.4 (OpenAI, 2026) achieves 75.0% on OSWorld, Opus 4.6 (Anthropic, 2026) achieves 72.7%, and Kimi K2.6 (Team et al., 2025) achieves 73.1%, which surpass human performance (72.4%). Other smaller open weight models such as MolmoWeb (Gupta et al., 2026), Qwen3.5 (Team, 2026), also achieve strong performance on these benchmarks, while being small enough (e.g., 4–9B parameter scale) to run locally. Most of these CUAs are evaluated and deployed as single serial actors. In this paper, we explore how we might instead coordinate and orchestrate CUAs in a multi-agent system, unlocking more effective planning and backtracking, higher parallelism, and improved test-time scaling behavior.

Multi-Agent LLM Systems Multi-agent systems are composed of multiple interacting autonomous agents (Wooldridge, 2009), with a long history dating back as early as the 1980s (Durfee et al., 1989; Smith, 1979). More recently, multi-agent systems based on LLMs have been explored across a wide range of settings, including software engineering (SWE) (Hong et al., 2023; Geng & Neubig, 2026), scientific discovery (Lu et al., 2024; Gottweis et al., 2025), web search (Lee et al., 2026), and more. Many of these works frame the system as a computational graph (Zhuge et al., 2024; Qian et al., 2025) that can be dynamically adjusted (Chen et al., 2023) and scaled, or as systems that can recursively spawn subagents (Zhang et al., 2025a; Gandhi et al., 2026). Others target decomposition and scheduling in a more structured form, such as using a planner module that delegates tasks to specialized modules (Li et al., 2025; Zhang et al., 2025b; Nielsen et al., 2026; Xu et al., 2026). Specifically, in the computer use agent literature, a parallel thread focuses on *dynamic* replanning as the agent navigates. (Koh et al., 2025) proposes a tree search algorithm for CUAs which performs lookahead and backtracking using a value function to score each proposed action. Wang et al. (2025a) introduces a framework for inducing, verifying, and using program-based skills on the fly, as an agent continuously interacts with a web environment. Agent Workflow Memory (Wang et al., 2025b) learns reusable workflows and stores these for subsequent generations. Aghzal et al. (2026) find in a controlled hierarchical-planning study that a single round of exploratory replanning substantially improves success on web navigation tasks. Others explore orchestrating specialized submodules: Agent S2 (Agashe et al., 2025) introduces proactive hierarchical planning, refining plans after each subgoal success, and delegating tasks to workers and specialized expert modules, and CoAct-1 (Song et al., 2026) and InfantAgent-Next (Lei et al., 2025) delegate tasks to either a GUI operator or a code agent. Several systems have also been deployed successfully in user-facing products for web navigation and information gathering (Yutori team, 2025; Anthropic, 2025).

In this paper we investigate how to best develop a multi-agent framework for general computer use: a partially observable problem that involves complex challenges such as grounding, computer interaction, and long-horizon planning. Our framework is general, simple and scalable. We only use 2 types of agents: a planner, and a homogenous set of subagent workers, while prior work typically uses specialized modules for handling low-level actions (Agashe et al., 2025; Lei et al., 2025; Song et al., 2026), or focus on fully observable environments such as code generation (Hong et al., 2023; Geng & Neubig, 2026) and math (Motwani et al., 2025; Zhao et al., 2025; Madaan et al., 2025).

3 Multi-Agent Computer Use

The overall flow of MACU is shown in Fig. 1 and Alg. 1. MACU involves a single *manager* agent that coordinates a pool of *CUA subagents* by mutating a directed acyclic graph (DAG) of subtasks to be completed. Conditioned on an instruction specified in natural language, the manager produces an initial DAG decomposition of the task (Fig. 2). At each iteration, CUA subagents are dispatched in parallel to tackle subtasks on the ready frontier of the DAG (i.e., nodes whose dependencies are satisfied). Whenever new observations and evidence become available, the manager is prompted to revise the DAG. We describe the roles of the manager and subagents in detail below.

Manager The manager, which can be a local model or an API based model, orchestrates and coordinates CUA subagents that operate in isolated virtual machines (VMs). The manager is implemented as an LLM call that takes in the task instruction, the current DAG, and the latest observations from the CUA subagent nodes. The manager is invoked at several key points:

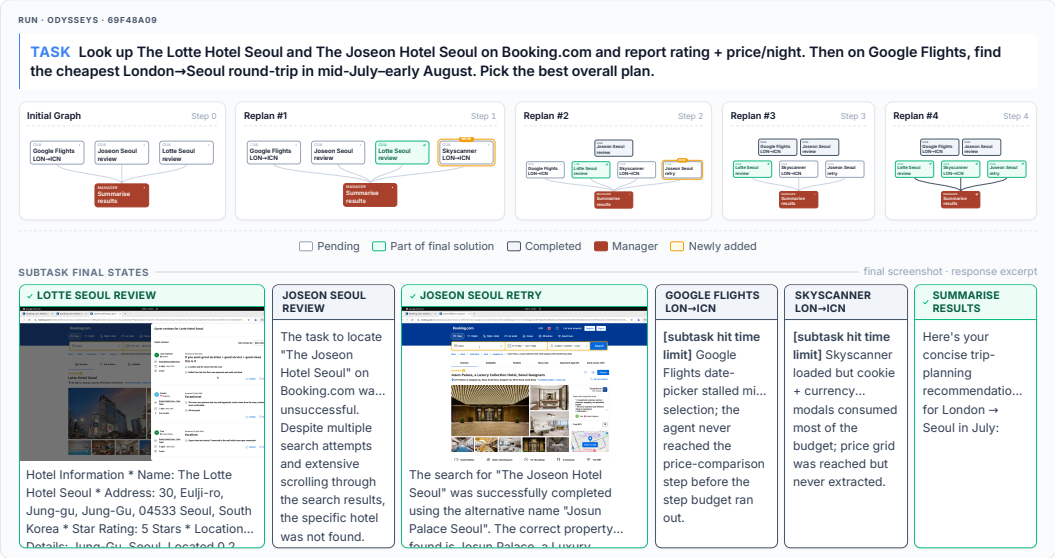


Figure 2: Example Odysseys travel planning task with MACU. The manager launches parallel hotel review and flight search subtasks, retries the failed lookup under an alternate name, and adds a Skyscanner subtask as a backup. Both flight branches reach sorted fare listings but time out before reporting, so the manager reads the fares from their final screenshots and forwards them to the aggregator. All five rubrics for this task were satisfied (Appendix I).

Algorithm 1 Algorithm for a MACU run on a single task.

Input: task T, max parallel subagents N, replan budget B
$G \leftarrow \textsc{Decompose}(T)$ ▷ initial DAG, consumes no budget
$b \leftarrow 0$; `files` $\leftarrow \emptyset$; `running` $\leftarrow \emptyset$; `completed` $\leftarrow \emptyset$
while deps(`final_aggregation`) $\not\subseteq$ `completed` **do**
  $R \leftarrow \{v \in G : \text{deps}(v) \subseteq \texttt{completed}, v \notin \texttt{running} \cup \texttt{completed}\}$
  Launch up to $N - |\texttt{running}|$ CUA subagents from R
  Wait until a subagent v finishes, or a spare-capacity trigger fires
  if a subagent v finished **then**
    $\textsc{FollowUp}(v)$; `results`$[v] \leftarrow v$.`response` ▷ manager may ask v to continue
    `completed` $\leftarrow$ `completed` $\cup \{v\}$; `files` $\leftarrow$ `files` $\cup\ \textsc{FileMgmt}(v)$
  end if
  if $b < B$ **then** ▷ after a completion, or with spare capacity
    $\Delta \leftarrow \textsc{Replan}(G, \texttt{results}, \texttt{files}, N - |\texttt{running}|)$
    if Δ is a valid, non-empty set of edits **then**
      $G \leftarrow \textsc{Apply}(G, \Delta)$; $b \leftarrow b + 1$ ▷ one budget unit per applied decision
    end if
  end if
end while
return $\textsc{Aggregate}(G,$ `results`, candidate VMs) ▷ final manager summary and results

1. **Initial DAG decomposition:** The manager is prompted with the original task instruction and the screenshot of the starting VM state (if any), and is instructed to produce a DAG G represented as a JSON string (which is validated for correctness).
2. **Replanning:** After a CUA subagent v completes its assigned task, the manager is provided with the current DAG G, the last k screenshots and final response from v, as well as context from the parent nodes of v in the form of their final VM screenshot and response. The manager is prompted to respond with a possible list of edits to make to G: `add` a node, `remove` a node, `modify` the instructions or dependencies of a node, or `cancel` a currently running node. These actions can change the structure of the graph or modify the instructions to be provided to future subagent nodes. Multiple edits can be made in a single

replan call, provided they still result in a valid DAG. Each edit consumes a unit of the replan budget B. Once B is exhausted we do not allow any additional edits to G. We also perform replanning intermittently if we have additional worker capacity that is not being used, to allow the manager to use spare capacity productively (e.g., exploration to find additional information, or try out different possible solutions).

3. **File management:** A key requirement of many CUA tasks is the ability to create, maintain, and produce files. On the completion of each CUA subagent, the manager is provided with a diff of the filesystem to identify added or modified files. It is prompted to determine which files to save. When starting up a new subtask, the manager can similarly provide previously retained files as input to the new subtasks (e.g., handing off a Word document from subtask 1 to be used by subtask 2 to populate slides in PowerPoint).
4. **Follow up:** When each CUA subagent completes its unit of work, the manager inspects it by being prompted with its latest responses and screenshot observations. The manager can optionally task the CUA subagent to continue working with a modified instruction (within the step limit), or get the subagent to summarize its work and terminate.
5. **Aggregate and summarize:** The final leaf node in the DAG prompts the manager to aggregate and summarize results from the full text responses and final screenshots of its parent nodes, and to select a VM instance whose end state contains any required deliverables.

Subagents Each subagent executes a standard ReAct (Yao et al., 2022) loop used by most frontier CUA models. At each iteration, the subagent observes a screenshot from its VM, produces reasoning traces and a computer action, which is then executed on the VM. MACU treats this subagent loop as a black box: each subagent runs on its own isolated VM. This treatment enables the manager to be free of model-specific assumptions, and agnostic to the specific CUA used, enabling MACU to work with new releases of improved CUA models (see Sec. 4.2 which show improved scaling trends with stronger subagents). We use the same CUA backbone for each subagent. Specialization only arises from manager-written instructions and the VMs that subagents initialize from or inherit.

4 EXPERIMENTS

We evaluate MACU on a diverse set of computer use benchmarks:

- **OSWorld** (Xie et al., 2024) measures the ability of models to control GUIs to complete 369 desktop tasks on an Ubuntu machine. Success rate is measured by execution-based evaluation scripts that test whether a model satisfied certain requirements.
- **Online-Mind2Web** (Xue et al., 2025) is a set of 300 tasks testing web navigation on 136 different real world websites. Success rate is measured with the WebJudge LLM-as-a-judge over the trajectory's actions and screenshots.
- **WebTailBench-v2** (Awadallah et al., 2025) is a set of 609 tasks designed to test breadth and depth of navigation, and aims to capture tasks that are underrepresented in other benchmarks. Success rate is measured with the benchmark's official multimodal rubric verifier (GPT-5.2 judge).
- **Odysseys** (Jang et al., 2026) is a benchmark of 200 long-horizon web navigation tasks derived from real user browsing behavior. Each task comes with predefined rubrics that measure key task requirements. We report the success rate (percentage of tasks that satisfied all rubrics), as well as the average rubric score (percentage of rubrics satisfied).

For our main experiments, we use the Qwen3.6-27B (Qwen Team, 2026) model for the CUA subagent, and Claude Opus 4.6 (Anthropic, 2026) as the manager. We also ran experiments where Qwen3.6-27B is used as both the subagent and the manager, to highlight the effectiveness of MACU even with a weaker manager model. We limit the run to maintain at most $N = 4$ subagent runs in parallel, and set replan budget $B = 10$ (see ablations in Sec. 4.2). Both setups share the per-task time limit, per-VM resources, and judges (Appendix B). More detailed experimental settings and all prompts used are provided in Appendices A and J. Since the general MACU setup makes no assumptions about the subagent model, it can in principle work off-the-shelf with any newer and more capable models, and we expect MACU to continue to be complementary to new CUA model releases.

Table 1: Single vs. multi-agent success rate (SR). All setups use Qwen3.6-27B as the subagent backbone. Δ indicates the change when going from single-agent to multi-agent (improvement). †Significant at $p < 0.001$.

Benchmark	Horizon	Setup	Manager	SR (%) (↑)	
				Avg.	Δ
OSWorld (Xie et al., 2024)	Short	Single-Agent	–	43.8	–
		Multi-Agent	Qwen3.6-27B	44.9	+1.1
		Multi-Agent	Opus 4.6	**48.5**	+4.7
Online-Mind2Web (Xue et al., 2025)	Short	Single-Agent	–	52.2	–
		Multi-Agent	Qwen3.6-27B	52.4	+0.2
		Multi-Agent	Opus 4.6	**55.6**	+3.4
WebTailBench-v2 (Awadallah et al., 2025)	Multi-step	Single-Agent	–	20.8	–
		Multi-Agent	Qwen3.6-27B	23.8	+3.0
		Multi-Agent	Opus 4.6	**29.5**	+8.7 †
Odysseys (Jang et al., 2026)	Long	Single-Agent	–	8.5	–
		Multi-Agent	Qwen3.6-27B	32.0	+23.5 †
		Multi-Agent	Opus 4.6	**34.0**	+25.5 †

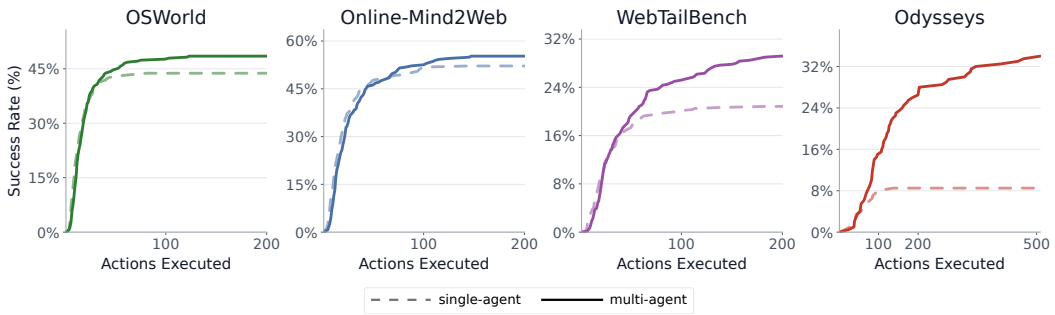


Figure 3: Success rate of MACU (Opus 4.6 manager) against the total number of CUA actions executed. Multi-agent unlocks more effective test-time scaling, even when including actions and wall-clock time from all explored (and discarded) branches (Fig. 6).

4.1 Results

Our main experimental results are summarized in Tab. 1. MACU improves the success rate on all four benchmarks, and the gains are larger on benchmarks with long task horizons. With the Opus 4.6 manager, MACU improves the success rate on OSWorld from 43.8% to 48.5%, and on Online-Mind2Web from 52.2% to 55.6%. These tasks are mostly short and run with a single subagent (86% and 95% of initial DAGs), leaving little room for decomposition, and MACU preserves or improves single-agent performance (within single-run variation, see confidence intervals in Appendix C).

Conversely, MACU achieves larger improvements on WebTailBench-v2 and Odysseys, whose tasks are more complex or longer-horizon and benefit from decomposition into parallel subproblems. On WebTailBench-v2, MACU improves the success rate from 20.8% to 29.5% ($p < 0.001$), and the percentage of rubrics satisfied from 35.9% to 46.3%. On Odysseys, MACU substantially improves the success rate from 8.5% to 34.0% ($p < 0.001$), and the rubric score from 42.1% to 62.3%, with 56 tasks solved only by MACU and 5 only by the single-agent baseline. The gain remains large and significant with Qwen3.6-27B as the manager (+23.5%, $p < 0.001$).

We also observed that under the same per-task time limit, MACU reaches the final single-agent success rate $4.4\times$ sooner on Odysseys (from 182 to 42 mins) and $1.5\times$ sooner on OSWorld, leading at every wall-clock budget on both (see Fig. 6 and Appendix D). On the largely serial Online-Mind2Web tasks, manager calls add latency and single-agent runs are faster at small budgets.

MACU also demonstrates positive scaling trends as models are provided with greater inference budget. Fig. 3 plots the success rate against the number of CUA subagent actions executed. Single-agent setups plateau much earlier on compared to MACU, and the success rate gap widens as more actions

Table 2: Results with varying planning budget B of applied DAG modifications ($B = 0$ is equivalent to single-agent). Average success rate (SR) (%) and manager API cost per task (USD $) are shown.

B	**Cost (↓)**	**SR (↑)**
0	0.00	25.0
1	0.08	27.8
5	0.41	47.2
10	0.46	58.3



Figure 4: Comparison of MACU against pass@8 and single-agent baselines.

are executed, demonstrating that MACU can effectively leverage greater inference budget to achieve stronger results. This plateau is not caused by the single-agent 200-step cap, which at most 10 of 200 Odysseys runs reach.

These findings strongly motivate the research community moving towards building, researching, and evaluating multi-agent computer use systems instead of serial agents, especially as we start deploying LLM agents on more complex, long running real world tasks.

4.2 Ablations

We conduct a series of ablation experiments to justify the design choices made in the MACU setup. Unless otherwise specified, all ablations are conducted on the official OSWorld `test_small` split excluding Google Drive tasks (36 tasks), with the Qwen-3.5-4B model as the worker and the Opus-4.6 model as the manager. We use Fisher's exact test to measure the significance of key results (Appendix C).

Importance of planning We ablate the planning budget B, the number of applied DAG modifications the manager may make. $B = 0$ is the single-agent baseline, and $B = 1$ allows the manager to produce an initial DAG (without additional modification during runtime). Tab. 2 shows that increasing B from 0 to 1 slightly increases the success rate ($25.0\% \rightarrow 27.8\%$), but allowing MACU to keep modifying the graph raises it more substantially to 47.2% with $B = 5$ and 58.3% with $B = 10$ ($p = 0.017$ over $B = 1$). This gap is not due to withholding findings from subagents, as the Opus 4.6 manager's initial DAG (independent of the worker) has a single CUA subagent for 30 of these 36 tasks, and aggregator inputs and VM inheritance do not depend on B. We find that 79% of applied decisions on OSWorld instead restructure the DAG with retries, variants, or continuations (Appendix E), highlighting the importance of continuous adaptive replanning.

Comparison against pass@k A natural comparison point for MACU is pass@k: we run a single-agent repeatedly up to $k = 8$ times, stopping when the groundtruth evaluator reports a success. The evaluator is typically not available at test time, but pass@k is a strong oracle-assisted upper bound for independent retries. Fig. 4 shows the success rate given a maximum number of total actions executed (tasks which are incomplete at that point are considered failures). We find that MACU outperforms pass@k up to 200 total actions executed, with both setups plateauing after. MACU is able to more effectively leverage inference-time compute, *despite pass@k having access to the groundtruth evaluator*. This highlights the value of the task decomposition, replanning, and error correction features of MACU.

Stronger CUA subagents We experiment with larger and more capable models as the backbone for the subagents. Tab. 3 summarizes the results, which validate that using stronger subagents generally leads to improved performance on both the single-agent and multi-agent setups. The strong results of the Qwen3.6-27B model led us to adopt it as the subagent backbone for our main experiments.

Stronger manager models We ablate the choice of manager model while keeping the CUA worker fixed (Qwen3.5-4B) in Tab. 4. The MACU framework improves upon the single-agent with all manager models we tested, but using stronger LLMs as managers substantially improve performance. Capable managers can more than double the success rate of the fixed Qwen3.5-4B CUA worker,

Table 3: Ablation results with different sized CUAs for subagents (success rate, %) for single-agent vs. MACU.

Model	Single	MACU
Qwen3.6-27B	47.2	66.7
Qwen3.6-35B-A3B	50.0	58.3
Qwen3.5-9B	50.0	61.1
Qwen3.5-4B	25.0	58.3
Qwen3.5-2B	27.8	44.4

Table 4: Manager model ablation with fixed CUA subagent model (Qwen3.5-4B). We report average success rate (SR) and API cost (– for self-hosted models).

Manager Model	SR (%) (↑)	Cost ($) (↓)
— (single-agent)	25.0	–
Opus-4.6	**58.3**	0.46
Sonnet-4.6	52.8	0.28
GPT-5.4	44.4	0.42
GPT-5.4-mini	33.3	0.37
Kimi-K2.6	41.7	–
Qwen3.6-27B	41.7	–

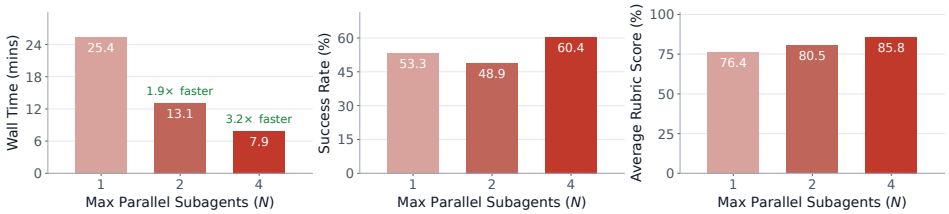


Figure 5: Wall-clock time, success rate, and average rubric score on the Odysseys easy subset as a function of the maximum number of parallel subagents N.

whereas weaker managers provide only marginal gains. Opus 4.6 obtains the best result among the evaluated managers, reaching 58.3% SR, a 33.3% absolute gain over the single-agent baseline ($p = 0.008$). Sonnet 4.6 achieves the next best score (52.8%).

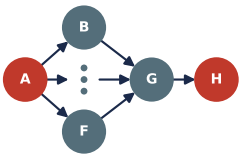
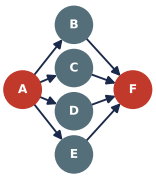
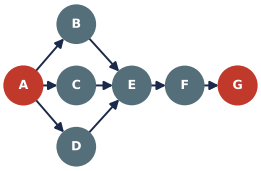
Number of parallel subagents On the Odysseys easy subset (45 tasks), we varied the maximum number of parallel subagents N to examine the wall-clock speedup provided by MACU ($N = 1$ is a serial manager-worker system). As visualized in Fig. 5, increasing N substantially reduces median wall-clock time, from 25.4 minutes ($N = 1$) to 7.9 minutes ($N = 4$), while improving success rate from 53.3% to 60.4%, and average rubric score from 76.4% to 85.8%. These times exclude model inference, as this has high variance with cluster load, and correspond to a $\sim$1.9× ($N = 2$) and $\sim$3.2× ($N = 4$) speedup (MACU still achieves a significant speedup when including model inference time, as described in Sec. 4.1). The serial $N = 1$ system already attains most of the accuracy, suggesting that accuracy gains come mainly from decomposition and management, while parallelism mainly improves speed.

Same model for manager and subagent Our results using Qwen3.6-27B as both the subagent and the manager in Tab. 1 demonstrate that the MACU setup has benefits beyond using instructions and responses from a stronger manager model. On Odysseys, using Qwen3.6-27B as the manager retains 92% of the Opus 4.6 manager's gain, and continues to achieve improvements across all benchmarks. In Sec. 4.3, we also highlight how MACU unlocks structural changes to task execution beyond improved instructions and responses.

4.3 Analysis

When does multi-agent help? We observed especially large gains on easy and medium Odysseys tasks (22.2% → 82.2% and 11.1% → 46.7%), and hard tasks also improved in success rate (1.8% → 9.2%) and rubric score (26.5% → 43.1%). WebTailBench-v2 categories which required parallel information gathering and comparison benefit the most from MACU: `price_comparison` tasks improved from 3.7% → 33.9%, `flights` from 14.0% → 34.0%, and `compositional_tasks_v2` from 24.0% → 41.8%. Online-M2W showed gains mainly on easy tasks (72.4% → 82.9%), and OSWorld on OS tasks (45.8% → 70.8%) and the LibreOffice suite (e.g., Writer 47.8% → 60.9%, Calc 34.0% → 46.8%). Across benchmarks, we observed a general trend that MACU is strongest on decomposable tasks that require parallel exploration and information gathering. This often requires preserving multiple pieces of the state before the final response (e.g., Odysseys tasks in Tab. 5).

Table 5: Types of DAG created for different tasks from the different benchmarks. Lettered nodes ● are referenced in the decomposition descriptions; red nodes ● denote manager action.

Benchmark	Task	Graph	Decomposition Pattern
OSWorld	Change GIMP's theme from dark to light.		**Simple chain:** Manager (A) creates a CUA worker (B) to change the theme, then (C) summarizes the result.
	Gather Google Maps information for five Hong Kong restaurants and enter it into a spreadsheet.		**Map-reduce:** Manager (A) creates parallel restaurant lookups (B–F), which feed a spreadsheet worker (G), then a manager action (H) reports.
Online-M2W	Find the newest digital-trends report for the Finance & Insurance industry in China.		**Runtime retry expansion:** Manager (A) creates the initial search (B) and alternate search variants (C–E), then a manager action (F) selects evidence.
Odysseys	*(summarized)* Plan a realistic five-day Iceland camper-van route using campgrounds, maps, and travel sites.		**Map-reduce:** Manager (A) creates campgrounds, maps, and travel-page research (B–D), followed by itinerary synthesis (E), tab curation (F), and final manager action (G).

Which mechanisms lead to gains? From the 56 Odysseys tasks that are solved by MACU but not the single-agent baseline, 19 are solved by the initial parallel decomposition alone, 17 additionally rely on the manager forwarding findings between subagents, and 20 on replanning the DAG on-the-fly (Appendix E). On OSWorld, most such tasks (42 of 61) use a single subagent with the manager's instructions and follow-ups, and 19 of 167 successes are attributed to replanning, which a static DAG cannot produce. These gains also demonstrate that MACU trajectories improve beyond the quality of the final synthesized response, as OSWorld is scored on VM state and the Online-M2W judge does not see the final answer (Appendix B), which require the model to take qualitatively better actions.

Types of graphs Tab. 5 and Tab. 11 show example final DAGs. OSWorld and Online-M2W tasks mostly involve single-task GUI execution and yield smaller, more serial graphs, where MACU gains through retries and alternative strategies (Appendix I), akin to tree search (Koh et al., 2025) at the subtask level. Since these tasks benefit less from decomposition and parallelization, their gains are smaller than on WebTailBench-v2 and Odysseys.

5 Conclusion

In this paper, we argued that computer use agents should move beyond the prevailing single serial agent paradigm, and proposed a general multi-agent computer use (MACU) framework that directly targets complex and long-horizon CUA tasks. Across four diverse computer use benchmarks, we demonstrated that MACU improves the success rates of strong single-agent baselines, with large and statistically significant gains on long-horizon tasks, and exhibits more favorable test-time scaling than its single-agent counterpart under matched action and wall-clock budgets. Our ablations and analysis show that continuous replanning and decomposition drive these gains while parallel execution provides speed, and that multi-agent coordination is a promising axis for scaling computer use agents to work productively for longer. Integrating new CUA models into MACU is seamless, and we expect its gains to compound with stronger CUA releases. Training managers and workers to improve coordination (e.g., with reinforcement learning) is a promising direction for future work. We release our code and interactive visualizations of MACU on selected tasks at `removed_for_review` to facilitate future work.

AI USE STATEMENT

In this work, we used generative AI tools to generate code for producing several visualizations used in this paper, primarily results plots (Fig. 3), charts (Fig. 5), and graph structures (Tab. 5). We additionally used generative AI to polish the writing of several sentences in the paper. We did not use generative AI tools for drafting large sections of the paper, finding related work, research ideation, generating synthetic datasets, or proving mathematical claims. The authors of the paper have reviewed all AI-assisted work. We verified and tested LLM-generated code to ensure it is using the appropriate results for producing visualizations. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We have made several efforts to ensure the reproducibility of our results. The MACU framework is described in Sec. 3, with the full procedure for a single task given in Alg. 1. Sec. 4 specifies benchmarks we used in this work (OSWorld, Online-Mind2Web, WebTailBench-v2, and Odysseys, all publicly available), the evaluation metrics used, the subagent and manager models, and the parallelism and replan budget for the main experiments. Appendix A lists remaining experimental hyperparameters (Tab. 6), along with the compute resources, model serving setup, per-task API costs, and the implementation details of the virtual machine backends used. The complete prompts used are provided verbatim in Appendix J (Tabs. 16–19). All CUA subagents use open-weights Qwen3.5 and Qwen3.6 models, and all manager models are either open-weights models or accessible through public APIs. The judges, judge inputs, and time-limit handling used for evaluation are specified in Appendix B, and the ablation subset is the official OSWorld `test_small` split excluding Google Drive tasks. Due to the cost of running long-horizon multi-agent experiments, our reported results are from a single run, following standard practice in prior CUA work (Xie et al., 2024; Xue et al., 2025; Jang et al., 2026). To quantify the resulting uncertainty, we report paired bootstrap confidence intervals and paired t-tests computed from per-task outcomes (Appendix C), and we will release all per-task outcomes and analysis scripts. We also note that Online-Mind2Web, WebTailBench-v2, and Odysseys are evaluated on live websites whose content changes over time. This is an inherent limitation of these benchmarks rather than of MACU, and may lead to variation in scores when rerun at a later date. We will release the full MACU codebase and interactive visualizations of selected runs upon publication.

REFERENCES

Saaket Agashe, Kyle Wong, Vincent Tu, Jiachen Yang, Ang Li, and Xin Eric Wang. Agent s2: A compositional generalist-specialist framework for computer use agents. *arXiv preprint arXiv:2504.00906*, 2025.

Pranjal Aggarwal, Graham Neubig, and Sean Welleck. Gym-anything: Turn any software into an agent environment. *arXiv preprint arXiv:2604.06126*, 2026.

Mohamed Aghzal, Gregory J Stein, and Ziyu Yao. Why do llm-based web agents fail? a hierarchical planning perspective. *arXiv preprint arXiv:2603.14248*, 2026.

Anthropic. How we built our multi-agent research system. `https://www.anthropic.com/engineering/multi-agent-research-system`, June 2025. Accessed: 2026-04-29.

Anthropic. Claude Opus 4.6 System Card. `https://www-cdn.anthropic.com/0dd865075ad3132672ee0ab40b05a53f14cf5288.pdf`, February 2026. Published February 6, 2026.

Ahmed Awadallah, Yash Lara, Raghav Magazine, Hussein Mozannar, Akshay Nambi, Yash Pandya, Aravind Rajeswaran, Corby Rosset, Alexey Taymanov, Vibhav Vineet, et al. Fara-7b: An efficient agentic model for computer use. *arXiv preprint arXiv:2511.19663*, 2025.

Rogerio Bonatti, Dan Zhao, Francesco Bonacci, Dillon Dupont, Sara Abdali, Yinheng Li, Yadong Lu, Justin Wagle, Kazuhito Koishida, Arthur Bucker, et al. Windows agent arena: Evaluating multi-modal os agents at scale. *ICML*, 2025.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, et al. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *ICLR*, 2023.

Edmund H Durfee, Victor R Lesser, and Daniel D Corkill. Trends in cooperative distributed problem solving. *IEEE Transactions on knowledge and data Engineering*, 1989.

Apurva Gandhi, Satyaki Chakraborty, Xiangjun Wang, Aviral Kumar, and Graham Neubig. Recursive agent optimization. *arXiv preprint arXiv:2605.06639*, 2026.

Jiayi Geng and Graham Neubig. Effective strategies for asynchronous software engineering agents. *arXiv preprint arXiv:2603.21489*, 2026.

Juraj Gottweis, Wei-Hung Weng, Alexander Daryin, Tao Tu, Anil Palepu, Petar Sirkovic, Artiom Myaskovsky, Felix Weissenberger, Keran Rong, Ryutaro Tanno, et al. Towards an ai co-scientist. *arXiv preprint arXiv:2502.18864*, 2025.

Tanmay Gupta, Piper Wolters, Zixian Ma, Peter Sushko, Rock Yuren Pang, Diego Llanes, Yue Yang, Taira Anderson, Boyuan Zheng, Zhongzheng Ren, et al. Molmoweb: Open visual web agent and open data for the open web. *arXiv preprint arXiv:2604.08516*, 2026.

Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. Webvoyager: Building an end-to-end web agent with large multimodal models. *ACL*, 2024.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *ICLR*, 2023.

Lawrence Keunho Jang, Jing Yu Koh, Daniel Fried, and Ruslan Salakhutdinov. Odysseys: Benchmarking web agents on realistic long horizon tasks. *arXiv*, 2026.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Russ Salakhutdinov, and Daniel Fried. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks. *ACL*, 2024.

Jing Yu Koh, Stephen McAleer, Daniel Fried, and Ruslan Salakhutdinov. Tree search for language model agents. *TMLR*, 2025.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. *SOPS*, 2023.

Ka Yiu Lee, Yuxuan Huang, Zhiyuan He, Huichi Zhou, Weilin Luo, Kun Shao, Meng Fang, and Jun Wang. Infoseeker: A scalable hierarchical parallel agent framework for web information seeking. *arXiv preprint arXiv:2604.02971*, 2026.

Bin Lei, Weitai Kang, Zijian Zhang, Winson Chen, Xi Xie, Shan Zuo, Mimi Xie, Ali Payani, Mingyi Hong, Yan Yan, et al. Infantagent-next: A multimodal generalist agent for automated computer interaction. *NeurIPS*, 2025.

Zhuofeng Li, Haoxiang Zhang, Seungju Han, Sheng Liu, Jianwen Xie, Yu Zhang, Yejin Choi, James Zou, and Pan Lu. In-the-flow agentic system optimization for effective planning and tool use. *ICLR*, 2025.

Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. The ai scientist: Towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292*, 2024.

Xing Han Lù, Zdeněk Kasner, and Siva Reddy. Weblinx: Real-world website navigation with multi-turn dialogue. *ICML*, 2024.

Lovish Madaan, Aniket Didolkar, Suchin Gururangan, John Quan, Ruan Silva, Ruslan Salakhutdinov, Manzil Zaheer, Sanjeev Arora, and Anirudh Goyal. Rethinking thinking tokens: Llms as improvement operators. *arXiv preprint arXiv:2510.01123*, 2025.

Sumeet Ramesh Motwani, Chandler Smith, Rocktim Jyoti Das, Rafael Rafailov, Ivan Laptev, Philip HS Torr, Fabio Pizzati, Ronald Clark, and Christian Schroeder de Witt. Malt: Improving reasoning with multi-agent llm training. *COLM*, 2025.

Stefan Nielsen, Edoardo Cetin, Peter Schwendeman, Qi Sun, Jinglue Xu, and Yujin Tang. Learning to orchestrate agents in natural language with the conductor. *ICLR*, 2026.

OpenAI. GPT-5.4 Thinking System Card. OpenAI Deployment Safety Hub, https://deploymentsafety.openai.com/gpt-5-4-thinking, March 2026. Published March 5, 2026.

Jianing Qi, Xi Ye, Hao Tang, Zhigang Zhu, and Eunsol Choi. Learning to reason across parallel samples for llm reasoning. *arXiv preprint arXiv:2506.09014*, 2025.

Chen Qian, Zihao Xie, Yifei Wang, Wei Liu, Kunlun Zhu, Hanchen Xia, Yufan Dang, Zhuoyun Du, Weize Chen, Cheng Yang, et al. Scaling large language model-based multi-agent collaboration. *ICLR*, 2025.

Qwen Team. Qwen3.6-Plus: Towards real world agents. https://qwen.ai/blog?id=qwen3.6, April 2026.

Reid G Smith. A framework for distributed problem solving. *IJCAI*, 1979.

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. *ICLR*, 2026.

Kimi Team, Yifan Bai, Yiping Bao, Y Charles, Cheng Chen, Guanduo Chen, Haiting Chen, Huarong Chen, Jiahao Chen, Ningxin Chen, et al. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.

Qwen Team. Qwen3.5-omni technical report. *arXiv preprint arXiv:2604.15804*, 2026.

Zora Zhiruo Wang, Apurva Gandhi, Graham Neubig, and Daniel Fried. Inducing programmatic skills for agentic tasks. *COLM*, 2025a.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. *ICML*, 2025b.

Michael Wooldridge. *An introduction to multiagent systems*. John wiley & sons, 2009.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *NeurIPS*, 2024.

Jinglue Xu, Qi Sun, Peter Schwendeman, Stefan Nielsen, Edoardo Cetin, and Yujin Tang. Trinity: An evolved llm coordinator. *ICLR*, 2026.

Tianci Xue, Weijian Qi, Tianneng Shi, Chan Hee Song, Boyu Gou, Dawn Song, Huan Sun, and Yu Su. An illusion of progress? assessing the current state of web agents. *COLM*, 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 *ICLR*, 2022.

Yutori team. Building the proactive, multi-agent architecture powering scouts. https://yutori.com/blog/building-the-proactive-multi-agent-architecture-powering-scouts, December 2025. Accessed: 2026-04-29.

Alex L Zhang, Tim Kraska, and Omar Khattab. Recursive language models. *arXiv preprint arXiv:2512.24601*, 2025a.

Wentao Zhang, Liang Zeng, Yuzhen Xiao, Yongcong Li, Ce Cui, Yilei Zhao, Rui Hu, Yang Liu, Yahui Zhou, and Bo An. Agentorchestra: Orchestrating multi-agent intelligence with the tool-environment-agent (tea) protocol. *arXiv preprint arXiv:2506.12508*, 2025b.

Wenting Zhao, Pranjal Aggarwal, Swarnadeep Saha, Asli Celikyilmaz, Jason Weston, and Ilia Kulikov. The majority is not always right: Rl training for solution aggregation. *arXiv preprint arXiv:2509.06870*, 2025.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. *ICLR*, 2024.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. Language agents as optimizable graphs. *ICML*, 2024.

A Additional Experimental Details

Table 6: Experimental settings used by the main MACU experiments.

Setting	Value
Manager provider and model	Anthropic `claude-opus-4-6`
CUA worker for main experiments	`Qwen/Qwen3.6-27B`
Max parallel CUA subagents N	4
Replan budget B	10
Manager maximum output tokens	8192 for manager calls; 16384 for initial graph generation
Manager visual context	2 recent screenshots per relevant subtask; a max of 12 total screenshots
Maximum runtime per task	1.5 hours for OSWorld, 3 hours for Online-Mind2Web, WebTailBench-v2, and Odysseys (same for single-agent runs)
Per-subtask limits	60 model turns and 1 hour per subtask; single-agent runs use 200 turns for their one worker
Replan budget accounting	One unit per replan call whose decision is applied to the DAG; `no_change`, rejected decisions, the initial decomposition, follow-ups, and file management are free
Sampling	Subagents: temperature 0.6 (1.0 for variant nodes); manager: temperature 0
Follow-up hook	Enabled for completed CUA subtasks; the manager may request continuation or a targeted summary before termination
File management	Enabled after each CUA completion through filesystem diffs, an archive pool, and replanner-assigned `input_files`

We summarize the experiment hyperparameters in Tab. 6. These settings are shared across the main benchmark runs in Sec. 4.1, and retained for ablations in Sec. 4.2 (unless we explicitly mention changing the manager model, CUA worker model, parallelism, or replan budget). We provide the prompts used for the manager in Appendix. J.

For MACU runs, we host a `Qwen/Qwen3.6-27B` model as an inference server using vLLM (Kwon et al., 2023) on two L40S GPUs (with tensor parallelism of 2). Every subagent VM is allocated 16G RAM and 4 CPUs in both the single-agent and MACU settings, and since MACU runs up to $N = 4$ VMs concurrently, we reserve 64G RAM and 16 CPUs per MACU task. We run 6 tasks per vLLM server, which typically fully saturates the inference workload. Each benchmark takes a varying amount of time to complete (see Appendix D for per-task wall-clock time). The manager API cost per MACU task (with the `claude-opus-4-6` manager, excluding the one initial decomposition call) averages USD $0.21 per OSWorld task, $0.38 per Online-Mind2Web task, $0.46 per WebTailBench task, and $0.90 per Odysseys task, which totals approximately $651 in API cost for a full run of all four benchmarks. The total wall clock time spent for the run is highly variable depending on cluster usage, but our final end-to-end run on all four benchmarks took approximately 38 hours to run end-to-end. A complete per-task accounting of subagent actions, VM time, wall-clock time, and manager tokens is given in Tab. 8.

For single-agent runs, we extend the max steps of the single worker to 200 turns, and allocate the resources of one VM (16G RAM and 4 CPUs per task). In practice this cap is rarely binding: 0 of 369 OSWorld, 2 of 300 Online-Mind2Web, and 10 of 200 Odysseys single-agent runs reach 200 actions, whereas 25%, 3%, and 55% of them end at the shared per-task time limit. Since a single model turn can issue several actions, these action counts are an upper bound on the number of turns.

Since Qwen3.5 and Qwen3.6 did not provide implementation details of their model, we do our best effort to reproduce it based on existing implementations (include how the models are used for SWE agents in qwen-code[1], but our results still fell short of the numbers reported in the Qwen3.5 releases.

[1] `https://github.com/QwenLM/qwen-code`

Nevertheless, our implementation still achieves substantially better results than all other open weights CUAs, so we use it as our baseline agent.

A.1 Information Flow Along DAG Edges

Because computer state is only partially observable during evaluation and inference, MACU moves information between subagents through three explicit channels:

- **Application state.** A newly launched node inherits the final VM (including the disk and memory) of its first completed CUA parent by default, and skips the canonical task setup. A node added with `init_from` resumes from the final VM of a chosen subtask, and a variant node forks a frozen copy of the original subtask's pre-run VM.
- **Findings.** Since each subagent sees only its own instruction, the manager forwards the results of parent nodes by rewriting the instructions of pending dependents during replanning. The manager is also shown the final screenshots of the parents, which allows it to recover findings that a subagent observed but never reported (Fig. 2). Forwarding is a replan decision and consumes budget, but only 0–3% of tasks exhaust the budget in the main runs (Appendix E).
- **Files.** After each subtask, the manager archives selected files from the filesystem diff of the VM, which pending subtasks then receive through `input_files`.

The final aggregator receives the full text results and final screenshots of all of its parents, and child subagents inherit their parent's VM by default.

A.2 Virtual Machines Technical Details

MACU launches each CUA subagent through OSWorld's `DesktopEnv`; the subagent process constructs the requested VM provider, starts the emulator, records the VM endpoint, and writes the in-VM HTTP, browser-debugging, VNC, and VLC ports to a status file `vm_info.json` for the MACU runtime. Subagents initialize from a provided VM state (if any). We implement two VM backbones:

VMware provider. Before launching an independent DAG node, the MACU runtime atomically claims a pool VM and creates a `vmrun snapshot` named for the subtask. For state transfer along DAG dependencies, a child node may reclaim a completed predecessor's VM and skip canonical setup, preserving the predecessor CUA subagent's final desktop and filesystem state (i.e., effectively resuming from the parent subtask). For retry or variant nodes, MACU makes a full VMware clone of its frozen initial state, so parallel attempts start from the same pre-subtask state without sharing a mutable VM.

Apptainer/QEMU provider. On clusters where Docker or VMware is inconvenient, the Apptainer provider boots the same OSWorld image by running `qemu-system-x86_64` inside an Apptainer SIF. The base image, overlay directory, SIF path, and run directory, are preconfigured, and each fresh CUA subagent VM is a new writable qcow2 overlay backed by the shared base image. Initialization and resumption are handled similarly to the VMware backend. When an Apptainer-backed CUA subagent starts from an inherited overlay, the provider walks the overlay backing chain to find a saved VM state overlay; if present, QEMU is launched with a migration command to restore from the overlay. State transfer for Apptainer DAG nodes is similarly implemented by overlay cloning. Child nodes skip canonical setup when it intentionally inherits that state. At run teardown, MACU kills only QEMU processes that own the run's overlays and deletes overlay files in child-before-parent order so qcow2 backing chains are not removed underneath still-existing child nodes.

B Evaluation Protocol

We evaluate single-agent and MACU runs with the same judges, judge configurations, and per-task time limits for consistency. The two setups differ only in how the judged trajectory is assembled due to the inherent difference in trajectory generation (MACU has parallel branches), which we describe below.

Judged trajectory. For MACU, the judged trajectory consists of the complete trajectories (actions, responses, and screenshots) of every subagent that is an ancestor of the final aggregation node in the final DAG, in order of completion, followed by the manager's final response as the last step. Ancestors that failed or timed out are still included, while subagents cancelled or pruned by the manager from the aggregation are excluded. For single-agent runs, the judged trajectory is the agent's full trajectory followed by its final response.

OSWorld. OSWorld is scored by the official execution-based checkers on a single VM: the single-agent VM, or for MACU the subagent VM used by the final aggregator. The text response is only used for tasks that are designed to be infeasible. If the final response declares the task infeasible, a `FAIL` action is submitted, as in the official OSWorld protocol.

Online-Mind2Web. We use WebJudge (Xue et al., 2025) with o4-mini (medium reasoning effort, screenshot relevance threshold 3) and its official prompts. The judge first extracts the task's key points, then scores each screenshot's relevance and keeps those scoring at least 3 (up to 50), and finally judges success from the task, the action history, and the kept screenshots. The final text answer is not shown to WebJudge (the final response appears only as a placeholder action), which means that improvements on Online-Mind2Web do not arise from better final answer synthesis.

WebTailBench-v2. We use the official WebTailBench (Awadallah et al., 2025) evaluator, the multimodal rubric-based Universal Verifier (MMRubricAgent), in its published configuration.[2] GPT-5.2 generates the task rubric, scores the relevance of each screenshot to each criterion, analyzes the screenshot evidence, rescores the rubric, detects unsolicited side effects, and issues the final outcome verdict. The o4-mini model handles auxiliary steps (action-only rubric scoring and task-validity checks). The verifier sees the task, the action history, the final answer, and the screenshots. Success is the verifier's binary outcome verdict, and the rubric score is the fraction of rubric points earned (pass threshold 0.8). We apply the same configuration to both the single-agent and MACU runs.

Odysseys. We use the benchmark's per-rubric judge with the Gemini 3.1 Flash-Lite (preview) model. Each rubric is judged independently from the task, the full action and response history (including the final answer), and the last 200 screenshots, without truncating trajectories. Odysseys rubrics include explicit verification criteria that require navigational evidence in the trajectory (e.g., "the grader can confirm the Booking.com property page ... is open or was visited"), so a correct-looking answer without supporting evidence does not satisfy them.

C Statistical Analysis

Due to the cost of running CUA benchmarks, we follow the common practice (Xie et al., 2024; Xue et al., 2025; Jang et al., 2026) of computing results from a single run per configuration. In order to quantify uncertainty, in this section we compute paired statistics over tasks with a recorded outcome in both single-agent and MACU runs (Tab. 7). Δ is the mean per-task difference, and we show the 95% confidence interval computed with a paired bootstrap over tasks (5000 resamples). p is given by a two-sided paired t-test on the per-task differences. For WebTailBench-v2, we report an unpaired two-proportion test, which is conservative relative to a paired test. Because the analysis is restricted to paired tasks, the success rates differ slightly from Tab. 1.

The gains on the multi-step and long-horizon benchmarks are statistically significant: on Odysseys, MACU solves 56 (Opus 4.6 manager) and 52 (Qwen3.6-27B manager) tasks that the single-agent baseline does not, compared to 5 in the opposite direction. In contrast, the confidence intervals on OSWorld and Online-Mind2Web include zero. These benchmarks consist mostly of short tasks that MACU executes with a single subagent (Appendix E), and the many discordant pairs in both directions show that single runs of the same CUA vary substantially from task to task. These results indicate that MACU preserves or modestly improves single-agent performance on short tasks, while its benefits are concentrated on tasks that can be decomposed.

Ablations. As for the ablation experiments, we also compute 95% Wilson intervals for the planning budget ablation on the OSWorld ablation subset (36 tasks), which are 25.0 [13.8, 41.1] for $B = 0$,

[2] `https://github.com/microsoft/fara/tree/main/webeval`

Table 7: Paired significance analysis of the main results. Δ indicates the change when going from single-agent to MACU, with its 95% confidence interval (CI). Discordant pairs are the tasks solved only by MACU / only by the single-agent baseline.

Benchmark	Manager	n	Single	MACU	Δ [95% CI]	Discordant Pairs	p
OSWorld	Opus 4.6	357	43.9	48.6	+4.8 [-1.1, +10.7]	61 / 44	0.11
	Qwen3.6-27B	355	44.1	44.9	+0.8 [-5.0, +6.5]	52 / 48	0.79
Online-Mind2Web	Opus 4.6	286	52.8	56.3	+3.5 [-3.5, +10.1]	57 / 47	0.33
	Qwen3.6-27B	290	53.1	51.7	-1.4 [-8.3, +5.5]	50 / 54	0.70
WebTailBench-v2	Opus 4.6	600	20.8	29.5	+8.7 [+3.8, +13.6]	–	5×10^{-4}
	Qwen3.6-27B	600	20.8	23.8	+3.0 [-1.7, +7.7]	–	0.21

Table 8: Per-task resource accounting for single-agent and MACU runs (means unless noted). VM time sums the durations of all subagent VMs, and manager tokens are in thousands.

Benchmark	Setup	Manager	Subagents	Actions	VM Time (min)	Wall-clock (min) Mean	Wall-clock (min) Median	Time-outs	Manager Calls	Manager Tokens In / Out (k)
OSWorld	Single-Agent	–	1.0	25.9	50.8	50.8	47.9	25%	–	–
	MACU	Qwen3.6-27B	1.6	47.6	41.8	38.2	27.9	17%	5.7	39.0 / 6.7
	MACU	Opus 4.6	1.9	46.2	58.2	45.5	34.6	30%	5.0	35.4 / 1.5
Online-Mind2Web	Single-Agent	–	1.0	30.9	46.2	46.2	38.5	3%	–	–
	MACU	Qwen3.6-27B	2.0	76.9	68.1	57.3	43.6	8%	6.7	51.2 / 9.3
	MACU	Opus 4.6	3.3	97.9	151.1	89.3	71.1	19%	7.8	68.0 / 3.4
Odysseys	Single-Agent	–	1.0	97.7	145.2	145.3	180.8	55%	–	–
	MACU	Qwen3.6-27B	5.0	276.2	322.1	143.7	180.7	59%	15.3	171.4 / 24.7
	MACU	Opus 4.6	5.5	239.6	365.1	141.3	180.4	61%	13.4	148.4 / 7.5

27.8 [15.8, 44.0] for $B = 1$, 47.2 [32.0, 63.0] for $B = 5$, and 58.3 [42.2, 72.9] for $B = 10$ (Tab. 2). Fisher's exact test gives $p = 0.017$ when comparing $B = 10$ to $B = 1$, and $p = 0.008$ when comparing $B = 10$ to $B = 0$. In the manager ablation (Tab. 4), the Opus 4.6 manager improves over the single-agent baseline ($p = 0.008$). Smaller differences between manager models are not individually significant on this subset, and should be viewed as trends.

D Compute, Latency, and Budget-Matched Comparisons

We report the per-task compute and latency for OSWorld, Online-Mind2Web, and Odysseys in Tab. 8, and compare their success rates under matched action and wall-clock budgets in Fig. 6. Actions count every CUA action of every subagent, including cancelled and pruned branches, and wall-clock time is measured end-to-end, with timed-out tasks entering at their elapsed time. Since all subagents use the same model, subagent inference cost is proportional to the number of actions. Manager calls and tokens exclude the one initial decomposition call per task, and with the Opus 4.6 manager they correspond to an average API cost of \$0.21, \$0.38, and \$0.90 per task on the three benchmarks.

Matched action budgets. Fig. 3 counts the actions of the subagents whose trajectories feed the final output. Counting all branches (Fig. 6, top) leaves the conclusions unchanged, but this increases the total actions of MACU by 17% on OSWorld, 25% on Odysseys, and 2.1× on Online-Mind2Web, where many retries and variants are launched. On Odysseys, MACU leads at every budget beyond $\sim$50 actions (25.0% compared to 8.5% at 200 actions), and uses 39% fewer actions per solved task than the single-agent baseline (705 compared to 1150). On OSWorld, MACU leads from $\sim$30 actions onward, while on Online-Mind2Web, the single-agent baseline is more action-efficient at small budgets and MACU overtakes it past $\sim$185 actions.

Matched wall-clock budgets. We compare both systems under the same per-task time limit (Fig. 6, bottom). MACU reaches the final success rate of the single-agent baseline 4.4× faster on Odysseys (42 compared to 182 mins) and 1.5× faster on OSWorld (58 compared to 89 mins), and it leads at every budget on both benchmarks. On Online-Mind2Web, whose tasks are largely serial, manager calls add latency, and the single-agent baseline leads until close to the time limit. On Odysseys, both systems frequently use the full 3-hour budget, and when the limit is reached, the MACU aggregator

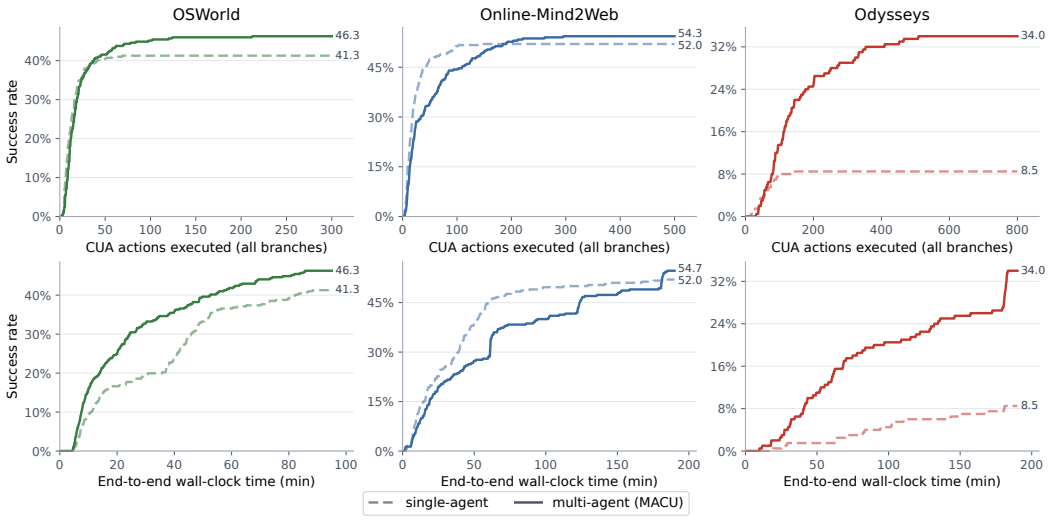


Figure 6: Success rate of the single-agent baseline and MACU (Opus 4.6 manager) against the total number of CUA actions across all branches (top) and end-to-end wall-clock time (bottom). A task counts as solved at budget x if it succeeded and its cost is at most x. Since the denominator is the full benchmark and success is binary, endpoints can differ slightly from Tab. 1.

synthesizes an answer from the partially completed trajectory, which accounts for the jump in its curve near the limit.

E Mechanism Analysis

We characterize how MACU (Opus 4.6 manager) uses each of its mechanisms (Tab. 9), and attribute the tasks solved by MACU but not by the single-agent baseline to the mechanisms used in their runs (Tab. 10). We classify each applied replan decision as *restructuring* (it adds, removes, or cancels subtasks), *forwarding* (it only rewrites pending subtasks' instructions or input files with parents' findings), or *aggregator re-wiring* (it only changes which results the aggregator uses).

Table 9: Mechanism usage of MACU (Opus 4.6 manager), as a percentage of tasks unless noted.

	OSWorld	**Online-M2W**	**Odysseys**
Initial DAG has a single CUA subagent	86.1	95.3	0.0
Initial DAG has a CUA subagent →CUA subagent dependency	10.5	2.0	58.5
Tasks with ≥ 1 applied replan	45.7	68.0	74.5
Applied replans per task (mean)	0.9	2.9	2.9
Tasks that exhaust the budget ($B = 10$)	0.0	3.3	1.5
Applied decisions: restructuring (% of decisions)	79.4	70.3	56.0
Applied decisions: forwarding only (% of decisions)	14.1	3.3	33.7
Applied decisions: aggregator re-wiring only (% of decisions)	6.5	26.5	10.2
Continuation from a subtask's final VM (`init_from`)	28.5	35.0	25.0
Variant nodes (forked from pre-run VM)	18.6	49.0	17.5
Findings forwarded to pending subtasks	10.5	16.0	59.0
Files transferred between VMs (`input_files`)	2.5	0.0	0.5
Cancellation of a running subtask	4.4	8.3	3.5
Follow-up continuations per task (mean)	0.4	0.42	1.4

Where the gains come from. On Odysseys, every initial DAG is a parallel decomposition, and the gains of MACU are spread across its components (Tab. 10). The gains hold both for tasks whose initial DAG contains CUA subagent → CUA subagent dependencies (6.0% → 25.6%, 117 tasks)

Table 10: Number of tasks solved only by MACU (Opus 4.6 manager), grouped by the most advanced mechanism used in their run, and tasks solved only by the single-agent baseline.

	OSWorld	**Online-M2W**	**Odysseys**
Solved only by MACU	61	57	56
no replanning (no DAG modifications)	42	21	19
forwarding findings only (no DAG modifications)	5	1	17
restructuring or aggregator re-wiring (DAG modified)	14	35	20
Solved only by the single-agent baseline	44	47	5

and for purely parallel ones ($12.2\% \rightarrow 46.3\%$, 82 tasks). On OSWorld and Online-Mind2Web, the manager spawns a single subagent for most tasks, and the gains are concentrated on these tasks, with success rate improving from $45.8\% \rightarrow 51.3\%$ on 308 such OSWorld tasks and $53.8\% \rightarrow 57.9\%$ on 273 such Online-Mind2Web tasks. In those tasks, MACU contributes through the manager's instruction, follow-up checks, and recovery. Replanning also creates state that a static DAG cannot: 19 of the 167 OSWorld successes of MACU are evaluated on a VM produced by a subtask that was added during replanning, 12 of which are continuations from the final state of an earlier subtask. On the web benchmarks, the final answer uses a replan-added subtask in 64 of 164 Online-Mind2Web successes and 18 of 68 Odysseys successes. Since replanning is triggered by difficulties, tasks with more replanning are harder by selection, so the categories in Tab. 10 describe the runs rather than isolate causal effects, which are instead measured by the replan budget ablation (Tab. 2).

F LIMITATIONS

While we have shown that MACU achieves promising results across computer-use benchmarks, our approach comes with several practical considerations and possible directions to address in future work.

Increased inference-time cost. MACU improves performance by providing an additional dimension to spend test-time compute, through decomposition, replanning, and parallel CUA subagent execution (Sec. 3). In our main experiments, we allow up to $N = 4$ parallel subagents and a replan budget of $B = 10$ (Appendix A), which can require substantially more model calls and VM execution time than a single-agent setting. This extra compute is not always offset by wall-clock speedup: on Online-Mind2Web, MACU improves success rate from 52.2% to 55.6%, but increases median end-to-end wall-clock time from 38.5 to 71.1 minutes because many tasks are largely serial (Appendix D). In practical systems, the parallelism level N, replan budget B, and per-subtask step limit must therefore be tuned based on whether the target workload actually benefits from additional exploration, as well as the GPU and CPU resources available at hand.

Single runs and statistical power. Because long-horizon multi-agent runs are expensive, each configuration was run once, and the ablations use a 36-task subset. The gains on WebTailBench-v2 and Odysseys are statistically significant, while those on OSWorld and Online-Mind2Web are within single-run variation (Appendix C). Future work should use repeated runs to further tighten these estimates.

Joint application states and manager context. The aggregator selects a single VM and does not merge application states, and tasks that split one desktop deliverable across several subagents remain hard (Appendix E). The manager also sees only the last few screenshots of each subtask, so information that a subagent observed earlier but never reported can be missed (Appendix I). Richer state reconciliation and incremental reporting by subagents are promising directions for future work.

Benefits depend on task decomposability. Our analysis shows that MACU is strongest on tasks that naturally decompose into parallel exploration, independent information gathering, retry attempts, or alternative execution strategies (Sec. 4.3). This explains the large gains on Odysseys, where many tasks have parallel map-reduce structure, and lower gains on OSWorld and Online-Mind2Web, which more often involve single-task GUI execution or serial interactions. For inherently sequential tasks,

spawning additional subagents may add overhead without creating useful parallel work. Future systems should learn when to use multi-agent execution, when to fall back to a single agent, and how to avoid manufacturing unnecessary subtasks.

Additional system complexity. Computer use differs from text-only agents because relevant state can live in screenshots, open tabs, filesystem changes, background processes, cookies, or application state (Sec. 3). MACU addresses this through isolated VMs, screenshot context, filesystem diffs, and an archive pool for selected files. These mechanisms make multi-agent execution possible, but they also introduce additional overhead in maintaining these additional infrastructure concerns. More robust state serialization, VM cloning, file provenance tracking, and automatic validation of final artifacts are important directions for future work.

Evaluation may not capture real world risks. We evaluated on standard computer-use benchmarks: OSWorld, Online-Mind2Web, WebTailBench-v2, and Odysseys, which cover desktop control, live website navigation, and long-horizon web tasks. These benchmarks are useful for measuring capabilities, but they do not fully capture the risks and operational constraints of real world deployments. Real user environments may involve private accounts, irreversible actions, authentication flows, or stricter reliability requirements. As discussed in Sec. G, deployment would require additional safeguards such as user authorization, action restrictions, monitoring, and confirmation for high-risk actions. Our results should therefore be interpreted as evidence that multi-agent coordination is a compelling research direction, not as evidence that multi-agent computer use is ready for real-world deployments.

G Broader Impacts and Ethics

As computer use agents continue to become more capable, multi-agent computer use systems which leverage these models presents both new opportunities as well as potential ethical considerations. MACU is a research framework for coordinating CUA subagents. In this paper, we evaluate the framework on existing computer-use and web-navigation benchmarks rather than deploying it to real users. We also do not introduce new model weights or datasets. The potential positive impact of this line of work is that more reliable CUAs, which are likely to arise in part due to strong multi-agent systems, could help people complete long-horizon computer tasks, automate repetitive workflows, and improve accessibility for users who have difficulty interacting with complex software interfaces. However, improving the reliability and speed of CUAs also raises potential concerns:

- **Intended uses.** Our work is intended as a research demo for studying how to coordinate multiple computer use agents on complex tasks. It is not, in its current form, intended for deployment on live user accounts, production websites, or security-sensitive environments. Any real-world use of multi-agent CUA systems should include further research, as well as careful scoping, user authorization guardrails, and safeguards around which applications, websites, files, and accounts an agent may access.
- **Misuse and security.** A system that can dispatch multiple CUA workers could be misused to scale harmful computer activity, including phishing, spam, manipulation, scraping, or other unauthorized workflows. Even in benign use, agents acting through GUIs and websites can make unintended changes, hallucinate incorrect information, or take destructive actions in a live environment. Eventual real world deployment of such systems should therefore require action restrictions, human approval for irreversible or high-impact operations, and continuous monitoring for abnormal behavior.
- **Duplicated side effects.** Retry and variant nodes run in parallel from the same starting state, which on live services could duplicate externally visible actions, such as submitting a form, sending a message, or placing an order twice. The benchmarks in this paper do not require irreversible transactions, and their VMs are isolated. Real-world deployments should let only a single designated branch commit side-effecting actions and require manager or human confirmation before any commit, which the DAG structure makes straightforward.
- **Privacy and data handling.** MACU passes screenshots, filesystem diffs, and selected files through a manager and shared archive pool. In real deployments, these artifacts could contain private documents, browser state, credentials, cookies, personal information, or

proprietary data. Practical systems should safeguard the data exposed to these models, redact sensitive content where possible, limit retention of screenshots and file archives, respect website terms of service, and ensure that users understand what information is being processed.

- **Economic impact.** More capable computer use agents may automate routine workflows in web navigation, document editing, data entry, software configuration, and information gathering. We expect such systems to be most useful when they augment people by offloading tedious or time-consuming menial computer tasks. At the same time, broader deployment may affect roles centered on repetitive computer work, and developers should proactively consider human oversight, worker transition costs, and the design of systems that complement rather than simply replace human labor.
- **Fairness and bias.** MACU relies on the behavior of the underlying manager and CUA models, which may inherit biases or uneven capabilities from their training data and model design. Performance may vary across languages, websites, software interfaces, accessibility contexts, and user groups. Before deployment, multi-agent CUA systems should be evaluated across diverse tasks and users, and should include mechanisms for user correction and appeal.

The structure of MACU also provides new opportunities for mitigation strategies. Because all work is mediated through the manager's DAG, safety checks can be inserted at decomposition, replanning, file-management, and final aggregation time. For example, a system can reject subtasks that visit disallowed websites, require confirmation before file deletion, or terminate subagents that appear to be looping or violating policy. The use of isolated VMs, explicit subtask instructions, and manager decisions provides natural boundaries for containment and review. We encourage future work on multi-agent computer use to study these safety mechanisms alongside capability improvements, especially before moving from benchmark environments to real user-facing deployments.

H Graph Shapes

Types of graphs. Tab. 11 shows other types of graphs we observed MACU producing during runtime.

How does the graph change over time? Tab. 12 shows the statistics of the DAG at the initial decomposition state, and at the end of the task (after potentially undergoing replanning). We observe that across all benchmarks, the graphs grow during the course of a task, due to the manager replanning and adding new subtasks as more information is acquired. On most benchmarks (with OSWorld as the exception), the majority of tasks required at least one replan. On Online-M2W, although replanning occurs less often (68% of tasks) compared to WebTailBench and Odysseys, the DAG changes the most dramatically, doubling in size from 2.1 nodes to 4.3 nodes on average, due to nodes added on-the-fly for retry attempts. Odysseys graphs are the largest, which matches our intuition that tasks are more complex and parallelizable (many tasks with the map-reduce pattern shown in Tab. 5).

I Additional Qualitative Examples

We showcase additional interesting qualitative examples, their spawned subtasks, and how the DAG evolves in each. We also show several examples of the subtask instructions generated by the manager.

I.1 OSWorld Examples

OSWorld (Xie et al., 2024) evaluates agents on desktop tasks in an Ubuntu environment. We showcase several examples below.

Font-install. MACU recovers from missing software in this example (Fig. 7). Tab. 13 records the manager-written instructions sent to each CUA subtask.

GUI timeout recovery. The initial GUI attempt to set up automatic saves hits the per-subtask time limit, so the manager launches recovery subtasks (Fig. 8).

Table 11: More types of DAG created for different tasks from the different benchmarks. Lettered nodes ● are referenced in the decomposition descriptions; red nodes ● denote manager action.

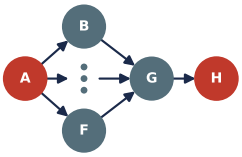
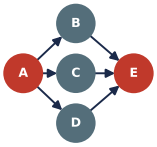
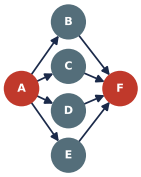
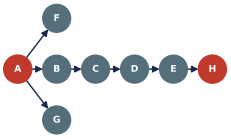
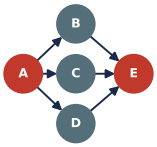
Benchmark	Task	Graph	Decomposition Pattern
OSWorld	Gather Google Maps information for five Hong Kong restaurants and enter it into a spreadsheet.		**Map-reduce:** Manager (A) creates parallel restaurant lookups (B–F), which feed a spreadsheet worker (G), then a manager action (H) reports.
Online-M2W	Compare Afghan Hound, Akita, and Azawakh dog-breed information.		**Map-reduce:** Manager (A) creates three independent breed lookups (B–D), which feed one comparative manager action (E).
WebTailBench-v2	Research where to buy *A Tale of Two Cities* and tabulate paperback and hardcover prices across retailers.		**Map-reduce:** Manager (A) creates four independent retailer lookups (B–E; Amazon, Barnes & Noble, Books-A-Million, ThriftBooks), which feed the final manager action (F).
	Count Ryanair extra-legroom seats from Birmingham to Porto for fixed travel dates.		**Retry chain:** Manager (A) launches an attempt (B) that is repeatedly retried (C–E) until it succeeds, while one-shot fallback attempts (F, G) are abandoned; a final manager action (H) reports.
Odysseys	*(summarized)* Build a small TV watchlist by verifying a Hulu listing and checking related reference pages.		**Map-reduce:** Manager (A) creates independent Hulu, Wikipedia, and Memory Alpha checks (B–D), which feed the final manager action (E).

Table 12: DAG statistics across benchmark runs. Replan rate is the percentage of tasks that executed at least one DAG modification during runtime.

Benchmark	Initial DAG	Final DAG	Replan Rate
OSWorld	2.3 nodes / 1.5 edges	2.9 nodes / 2.1 edges	45.7%
Online-M2W	2.1 nodes / 1.1 edges	4.3 nodes / 2.1 edges	68.0%
WebTailBench	2.3 nodes / 1.5 edges	4.2 nodes / 2.6 edges	73.5%
Odysseys	6.0 nodes / 8.0 edges	7.6 nodes / 8.3 edges	74.5%

Spotify CLI fallback. The app-store path fails, and the manager recovers with a CLI-based installation subtask (Fig. 9).

Parallel evidence gathering. The manager decomposes a local-environment configuration task into parallel subtasks for README and Colab-script inspection, then combines the evidence in a follow-up configuration step (Fig. 10).

Complementary web-search strategies. The manager launches two search branches that use the main cars.com interface and a direct relevant URL, then aggregates their complementary findings (Fig. 11).

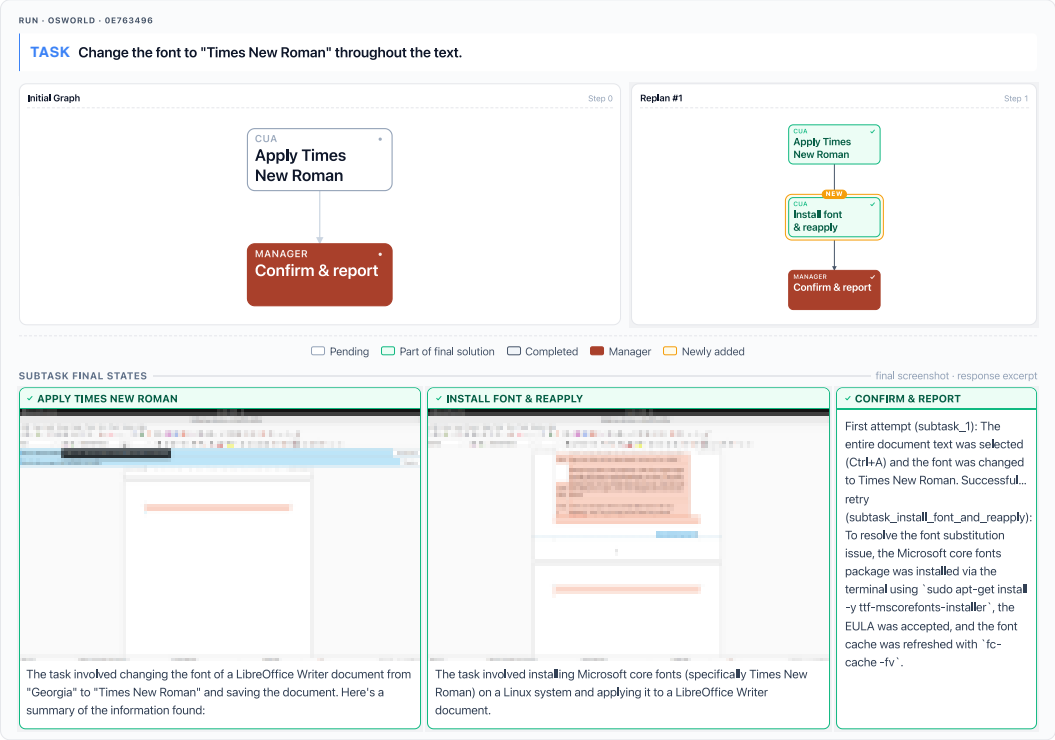


Figure 7: Example MACU run for an OSWorld font-configuration task. The first subagent attempts to change the font directly, discovers that the requested font is not installed, and reports the blocker. The manager then launches a second subagent to install the font and apply it successfully.

Table 13: Manager-written instructions sent to the CUA subtasks in the OSWorld font-install example shown in Fig. 7.

Subtask	Instruction provided by the manager
`subtask_1`	A LibreOffice Writer document is already open. Change the font to "Times New Roman" throughout the text. Use Ctrl+A to select all text, then click on the font name dropdown (currently showing "Georgia"), type "Times New Roman", and press Enter to apply. After the font is changed, save the document with Ctrl+S.
`subtask_install_font_and_reapply`	The document Dublin_Zoo_Intro.docx is open in LibreOffice Writer. A previous attempt set the font to "Times New Roman" but the system showed a warning that the font is not available and will be substituted. You need to: 1. First, open a terminal (right-click on the desktop and select "Open Terminal", or find Terminal in the application menu, or press Ctrl+Alt+T). 2. Install Microsoft core fonts by running: `sudo apt-get update && sudo apt-get install -y ttf-mscorefonts-installer`. If prompted for a license agreement, accept it (press Tab to highlight OK/Yes, then Enter). If that package name doesn't work, try: `sudo apt-get install -y fonts-liberation` (Liberation Serif is metrically equivalent to Times New Roman, but the actual Times New Roman is preferred). 3. After installation, run: `fc-cache -fv`. 4. Close the terminal and go back to LibreOffice Writer with Dublin_Zoo_Intro.docx. 5. Close and reopen the document (File > Close, then File > Recent Documents or reopen from Desktop) so LibreOffice picks up the newly installed fonts. 6. Select all text with Ctrl+A. 7. Click on the font name dropdown, type "Times New Roman", and press Enter. 8. Verify there is NO warning about font substitution. 9. Save the document with Ctrl+S. The file is located at `/home/user/Desktop/Dublin_Zoo_Intro.docx`.

I.2 Online-Mind2Web Examples

Online-Mind2Web (Xue et al., 2025) evaluates CUAs on live websites, where success can depend on robustly handling dynamic pages, redirects, and using web interfaces. We showcase several examples below.

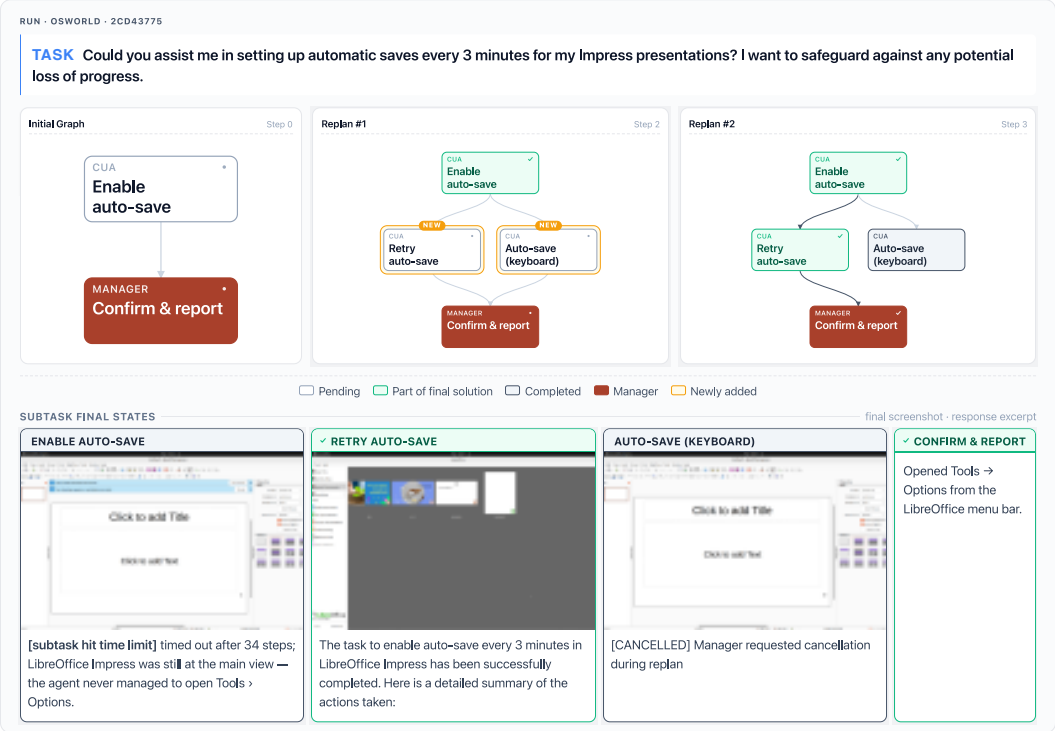


Figure 8: In this task, the initial GUI attempt to set up automatic saves hits the per-subtask time limit (1 hour). The manager responds by launching two recovery subtasks in parallel: one retries the GUI with revised instructions, while the other attempts a keyboard-shortcut-based strategy. The revised GUI retry succeeds, recovering the task after the initial timeout.

Filtered search resumption. MACU resumes a successful filtered-search state after the initial Petfinder GUI-filter attempt times out before applying all constraints (Fig. 12). Tab. 14 shows the manager-written instructions sent to each CUA subtask in this run.

Direct-page recovery. After repeated browser stalls on the Amtrak landing page or search results, the manager adds a direct branch to the passenger-identification page (Fig. 13).

Fallback search routes. When date-specific Yahoo Finance lookup attempts time out, the manager launches direct-URL and simple-search branches that both identify the requested Tesla closing price (Fig. 14).

I.3 Odysseys Examples

Odysseys (Jang et al., 2026) evaluates long-horizon web-navigation tasks derived from realistic browsing behavior. We showcase several examples below.

Parallel travel planning. MACU decomposes travel planning into hotel-review and flight-search branches, then retries blocked branches through alternate routes (Fig. 2). Tab. 15 shows the manager-written instructions sent to each CUA subtask in this run.

Evidence behind the final answer. In this run (Fig. 2), both flight branches reached sorted fare listings: Google Flights showed round trips from London Gatwick to Incheon with China Eastern at $1,286 (2 stops), and Skyscanner showed China Southern at $1,362 (1 stop) and Etihad at $1,368. However, both branches exhausted the per-subtask time limit while scrolling the results, before writing a textual report, and so returned only a time-out error. When replanning after each branch, the manager read the fares from the branches' final screenshots and wrote them into the aggregation instruction, allowing the final answer to name Google Flights and $1,286 as the cheapest source and

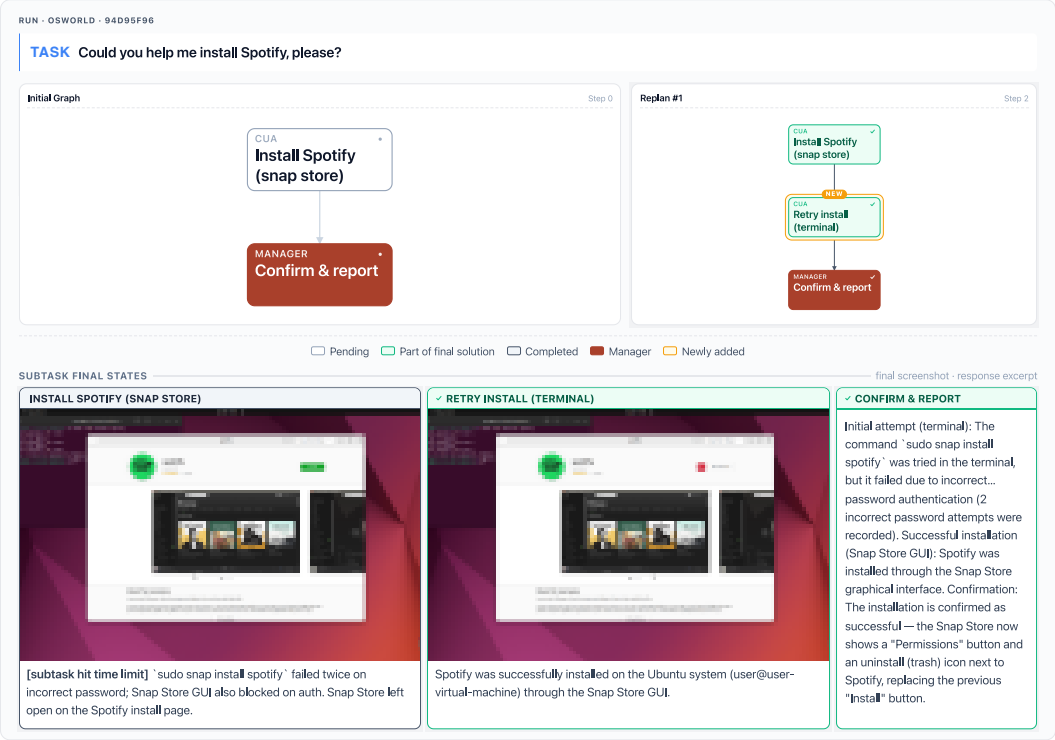


Figure 9: The initial subagent tries to install Spotify through the graphical app store, but that route fails. The manager then launches a CLI-based installation subtask, which completes successfully.

fare, together with the Skyscanner comparison. The hotel findings come from the successful Lotte review branch and the Joseon retry, which located the property under an alternate romanization. The Odysseys judge marked all five rubrics as satisfied: the flight comparison, the cheapest source and fare, the Joseon review score and sentiment, the Lotte review score with three takeaways, and the combined recommendation. This example also shows a limitation of the manager's bounded visual context (the last two screenshots per subtask): an earlier Google Flights screen also listed a $1,284 China Eastern fare and a $1,169 multi-carrier self-transfer itinerary, which the manager did not see. Giving the manager access to earlier steps, or having subagents report incrementally, would address this.

Scholarship retry. The manager retries a blocked American University scholarship branch and finds the PIPS scholarship information needed for the final outreach plan (Fig. 15).

Restaurant comparison aggregation. The manager aggregates weather, menu, and brunch-hours information, adding a verification branch after the first restaurant attempt is incomplete (Fig. 16).

I.4 WebTailBench Examples

WebTailBench-v2 (Awadallah et al., 2025) evaluates CUAs on a broad set of long-tail web tasks, spanning price comparison, travel and ticket booking, job search, and compositional multi-site goals. We showcase several examples below.

Parallel price comparison. The manager decomposes a multi-retailer price lookup for a book, *A Tale of Two Cities*, into parallel per-retailer subtasks and combines them into a final single comparison table (Fig. 17).

Hotel availability via retry expansion. A stalled hotel-booking attempt triggers several retry attempts, including modified variants, that converge on a confirmed no-availability answer (Fig. 18).

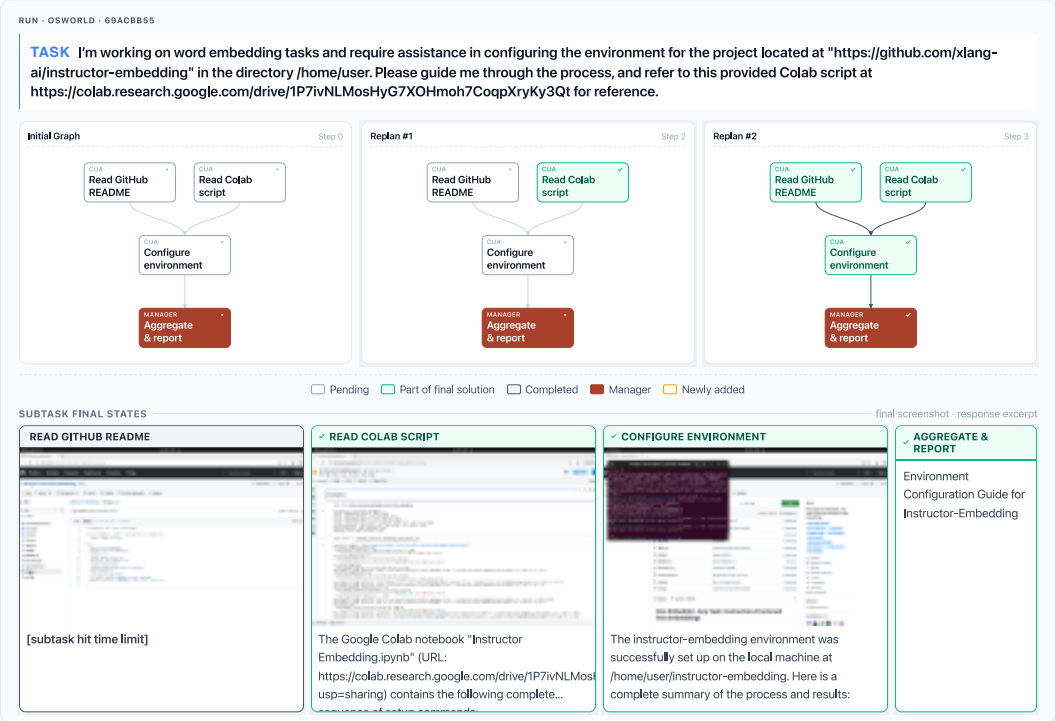


Figure 10: Example run for an OSWorld local-environment configuration task. The manager decomposes evidence gathering into two parallel subtasks: one reads the GitHub README, while the other inspects the Google Colab script. After both results return, the manager launches a follow-up subtask that uses the combined information to configure the local environment appropriately.

Compositional job search. The manager first enumerates open roles by job level, then replans to add a focused salary-range subtask for the most senior role (Fig. 19).

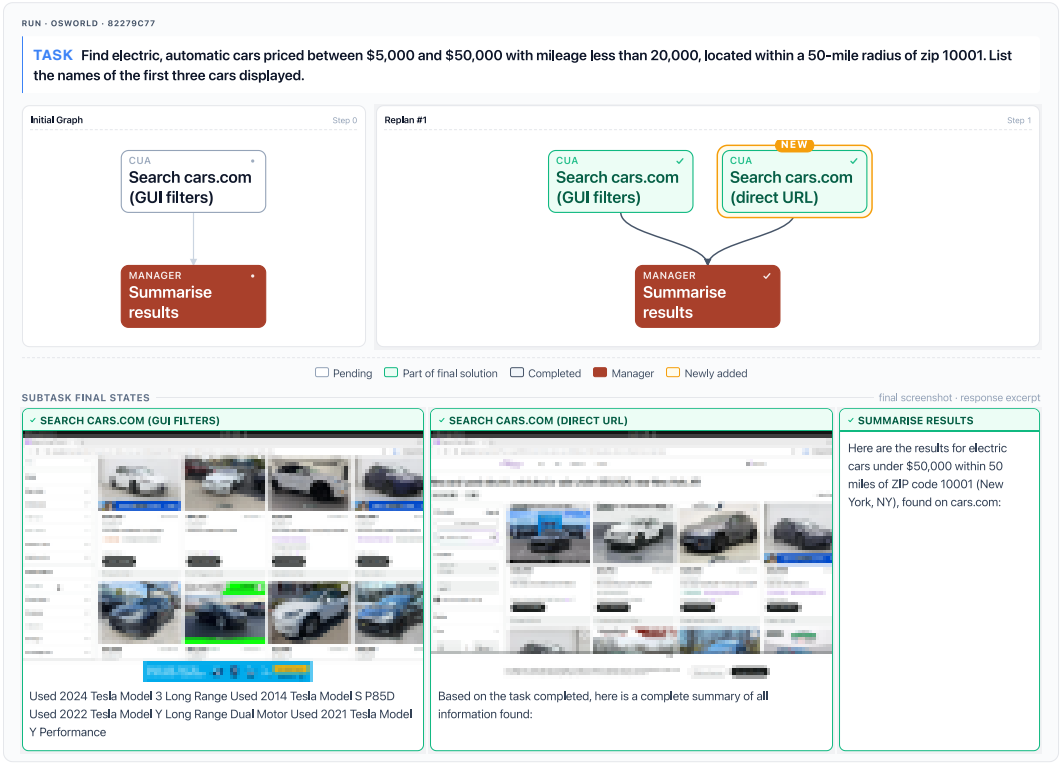


Figure 11: The manager launches two parallel search subtasks: one uses the main cars.com search interface and filters, while the other navigates directly to a relevant URL. The two subtasks return complementary information, which the manager aggregates into the final answer.

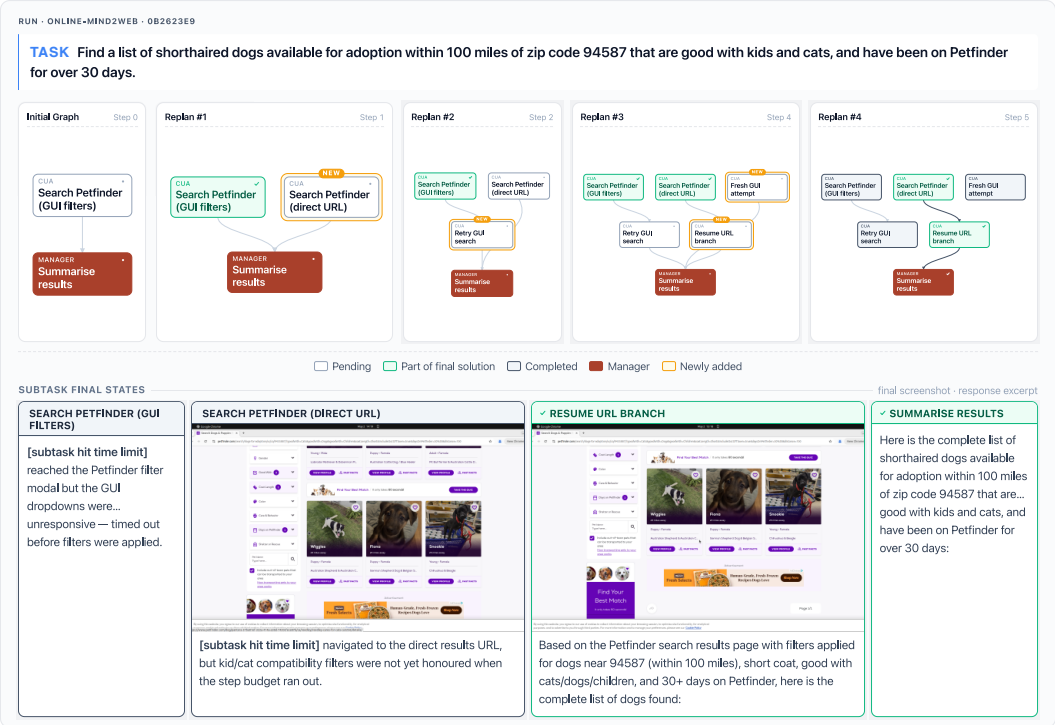


Figure 12: Example run for an Online-Mind2Web to find shorthaired dogs on Petfinder. The initial GUI-filter attempt times out before applying all adoption constraints, so the manager launches an attempt navigating to a direct URL, as well as retry branches. A resumed URL branch preserves the successful search state and produces the final list of matching dogs.

Table 14: Manager-written instructions sent to the CUA subtasks in the Online-Mind2Web Petfinder example shown in Fig. 12.

Subtask	Instruction provided by the manager
`subtask_1` Initial plan	Go to `https://www.petfinder.com` and search for dogs available for adoption. Apply the following filters: (1) location: zip code 94587, (2) distance: within 100 miles, (3) coat/hair length: Short, (4) good with: Kids and Cats, (5) days on Petfinder: 30+ days. After applying all filters, compile a list of the dogs shown in the results. For each dog, extract as much information as available from the listing cards (name, breed, age, gender, location, and any other details shown). If there are multiple pages of results, browse through them to get a comprehensive list. If the number of results is very large, collect at least the first 20–30 results and note the total count. Keep the Petfinder tab open with the filtered results displayed.
`subtask_1_variant_url` Direct-URL variant	Open Google Chrome and navigate directly to this URL: `https://www.petfinder.com/search/dogs-for-adoption/us/ca/94587/?goodWith=Cats&goodWith=Dogs&goodWith=Children&coatLength=Short&includeOutOfTown=true&daysOnPetfinder=30%2B&distance=100`. The URL includes filters for Dogs, within 100 miles of 94587, short coat, good with cats, good with children, and on Petfinder for 30+ days. If the `goodWith=Dogs` parameter doesn't work for "good with dogs", that's fine – the key filters are good with Children and good with Cats. Once the page loads, verify the filters in the left sidebar: Location 94587, Distance 100 miles, Coat Length Short, Good With Kids/Children and Cats, and Days on Petfinder 30+ days. If any filter is missing, apply it manually. After confirming all filters are active, note the total number of results and compile a list of dogs from the results with name, breed, age, gender, and location/distance. Browse multiple pages if needed to collect at least 20–30 results, or all results if manageable. Note the total count and final URL.
`subtask_1_retry` Resume retry	You are on the Petfinder website with a search for dogs near 94587, 100 miles, with filters for Good With Cats, Good With Dogs, and Short coat already applied (550 results showing). However, two critical filters are missing: (1) Good with Children/Kids – in the left sidebar, find "Good With" (it currently shows 2 selections: Cats and Dogs), expand it, and also check "Children" or "Kids"; (2) Days on Petfinder: 30+ – scroll down in the left sidebar filters, find "Days on Petfinder", expand it, and select "30+ days". After applying both missing filters, the result count should decrease significantly from 550. Once both filters are applied, compile a list of the dogs shown. For each dog, record name, breed, age, gender, and location/distance. Browse multiple pages to collect all results (or at least 20–30 if there are many). Note the total count and final URL. Make sure all 5 filters are active: 100 miles of 94587, Short coat, Good with Cats, Good with Children/Kids, and Days on Petfinder 30+.
`subtask_1_continue` State continuation	You are on the Petfinder search results page with all filters already applied: dogs near 94587, 100 miles, short coat, good with cats/dogs/children, 30+ days on Petfinder. The URL is `https://www.petfinder.com/search/dogs-for-adoption/us/ca/9455887/?goodWith=Cats&goodWith=Dogs&goodWith=Children&coatLength=Short&includeOutOfTown=true&daysOnPetfinder=30%2B&distance=100`. Your only job is to quickly collect the list of dogs shown in the results. Do not re-apply filters or navigate elsewhere. First, scroll to the top of the page to see the total result count. Then scroll through the results and record each dog's name, breed, age, gender, and distance. If there are more pages, click Next and continue collecting. Work fast – read the card info visible on screen, don't click into individual profiles. After collecting dogs from at least 2–3 pages (or all results if fewer), report the complete list and total count. Dogs already visible include: one Young Male Labrador Retriever & Doberman mix, one Puppy Female Australian Cattle Dog / Blue Heeler, one Adult Female Pit Bull Terrier & Australian Cattle D..., Wiggles, Fiona, and Snookie.
`subtask_1_fresh` Fresh variant	Navigate to this URL immediately: `https://www.petfinder.com/search/dogs-for-adoption/us/ca/94587/?goodWith=Cats&goodWith=Children&coatLength=Short&daysOnPetfinder=30%2B&distance=100`. Once loaded, scroll to the top to find the total result count. Then quickly scan each dog card on the page and record: name, breed, age (puppy/young/adult/senior), gender (male/female), and distance. Do not click into individual profiles – just read the cards. After finishing page 1, click Next to go to page 2 and repeat. Collect at least 2 pages of results. Report the full list and total count. Work as fast as possible.

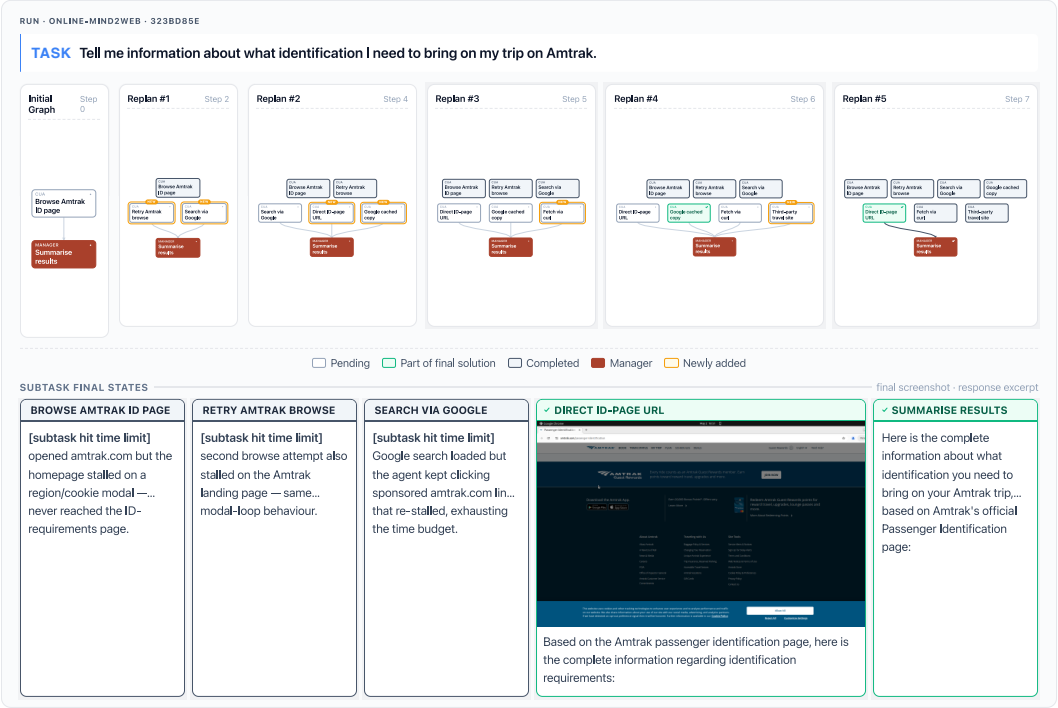


Figure 13: Example Online-Mind2Web run to find the identification required for boarding an Amtrak. Several browser and search attempts stall on the Amtrak landing page or search results. The manager then adds a direct branch navigating to the passenger identification page, which reaches the relevant page and finds the requested information.

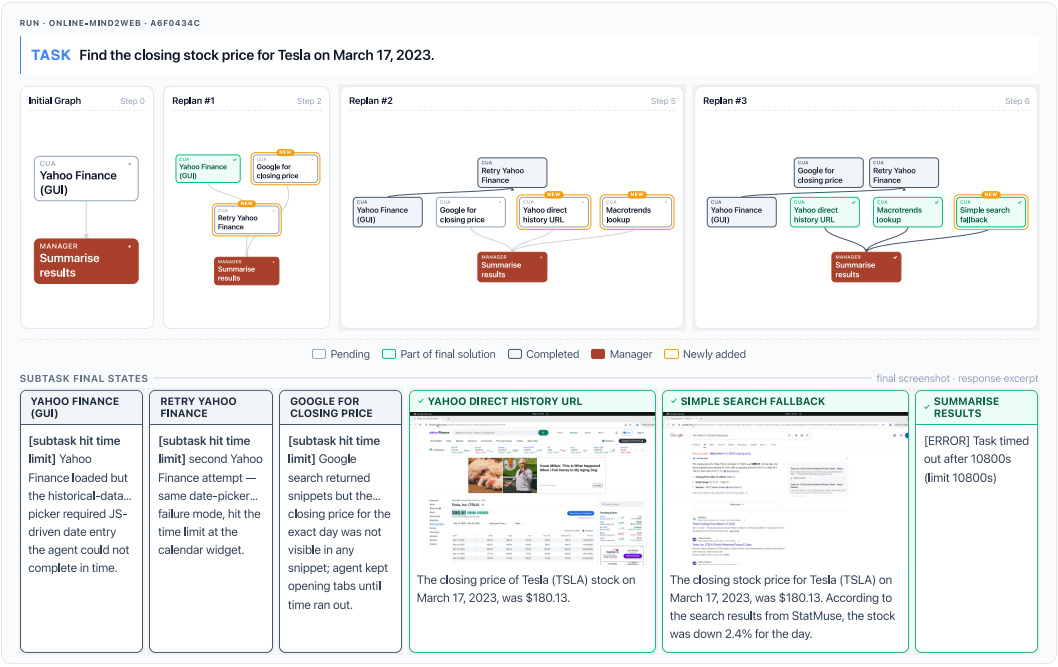


Figure 14: Example Online-Mind2Web task to find historical stock price. The initial Yahoo Finance GUI path and several search attempts time out while trying to locate the exact date. The manager launches a branch to navigate directly to a Yahoo Finance URL, as well as a simple search query branch, both of which successfully identify the Tesla closing price for March 17, 2023.

Table 15: Manager-written instructions sent to the CUA subtasks in the Odysseys parallel travel-planning example shown in Fig. 2.

Subtask	Instruction provided by the manager
`flight_search` Initial plan	Go to Google Flights (`https://www.google.com/travel/flights`). Search for round-trip economy flights from London (all airports) to Seoul (all airports: ICN/GMP). Set departure date to July 17, 2025 and return date to around August 1–3, 2025 (try a few early August return dates if needed to find the cheapest). Sort by price / cheapest. Record the top 3–5 cheapest options including: airline(s), number of stops, total travel time, price, and which booking source/airline offers that price. Identify the single best-value option (cheapest fare). Keep the Google Flights tab open with results visible.
`joseon_hotel_review` Initial plan	Go to Booking.com (`https://www.booking.com`). Search for "The Joseon Hotel Seoul" in Seoul, South Korea. You can use approximate dates (e.g., July 17 to August 2, 2025) or just navigate to the property page. Find and record: (1) the overall guest review score (e.g., 8.5/10 or similar), (2) the review rating category (e.g., "Excellent", "Very Good"), (3) read through recent guest reviews and note the general sentiment – are reviews overwhelmingly positive, mixed, or negative? Note 2–3 key themes from recent reviews (e.g., great location, excellent service, dated rooms, etc.). Keep the tab open.
`lotte_hotel_review` Initial plan	Go to Booking.com (`https://www.booking.com`). Search for "The Lotte Hotel Seoul" (Lotte Hotel Seoul) in Seoul, South Korea. You can use approximate dates (e.g., July 17 to August 2, 2025) or just navigate to the property page. Find and record: (1) the overall guest review score (e.g., 8.5/10 or similar), (2) the review rating category (e.g., "Excellent", "Very Good"). Then go into the reviews section and read recent reviews carefully. Record: (3) general sentiment – are reviews overwhelmingly positive, mixed, or negative? (4) At least 3 specific recent review takeaways – quote or closely paraphrase what individual recent reviewers said, noting both positives and any negatives. These should be concrete enough for the user to make a booking decision (e.g., "One reviewer praised the central Myeongdong location", "Another noted rooms felt slightly dated", etc.). Keep the tab open.
`flight_search_skyscanner` Replan variant	Go to Skyscanner (`https://www.skyscanner.net`). Search for round-trip economy flights from London (all airports) to Seoul (all airports: ICN). Set departure date to July 17, 2025 and return date to August 1, 2025 (also try August 2 and August 3 if easy, to find the cheapest). Sort results by cheapest price. Record the top 3–5 cheapest options including: airline(s), number of stops, total travel time, price in GBP or USD, and which booking source/agent offers that price. Identify the single cheapest option. This data will be compared against Google Flights results to find the overall best-value flight.
`joseon_hotel_review_retry` Replan retry	Go to Booking.com (`https://www.booking.com`). The previous search for "The Joseon Hotel Seoul" returned no results – the hotel may be listed under a different romanization. Try these search queries one at a time until you find it: (1) "Josun Palace Seoul", (2) "The Josun Seoul", (3) "Josun Hotel Seoul", (4) "Joseon Palace Seoul", (5) if none work, try just "Josun Seoul" or "Joseon Seoul". The hotel is a famous luxury 5-star hotel in the Jung-gu / Myeongdong area of Seoul, historically one of Korea's oldest and most prestigious hotels. Once you find the correct property, record: (1) the overall guest review score (e.g., 8.5/10), (2) the review rating category (e.g., "Excellent", "Very Good"), (3) read recent guest reviews and note the general sentiment – are reviews overwhelmingly positive, mixed, or negative? Note 2–3 key themes from recent reviews (e.g., great location, excellent service, dated rooms, etc.).

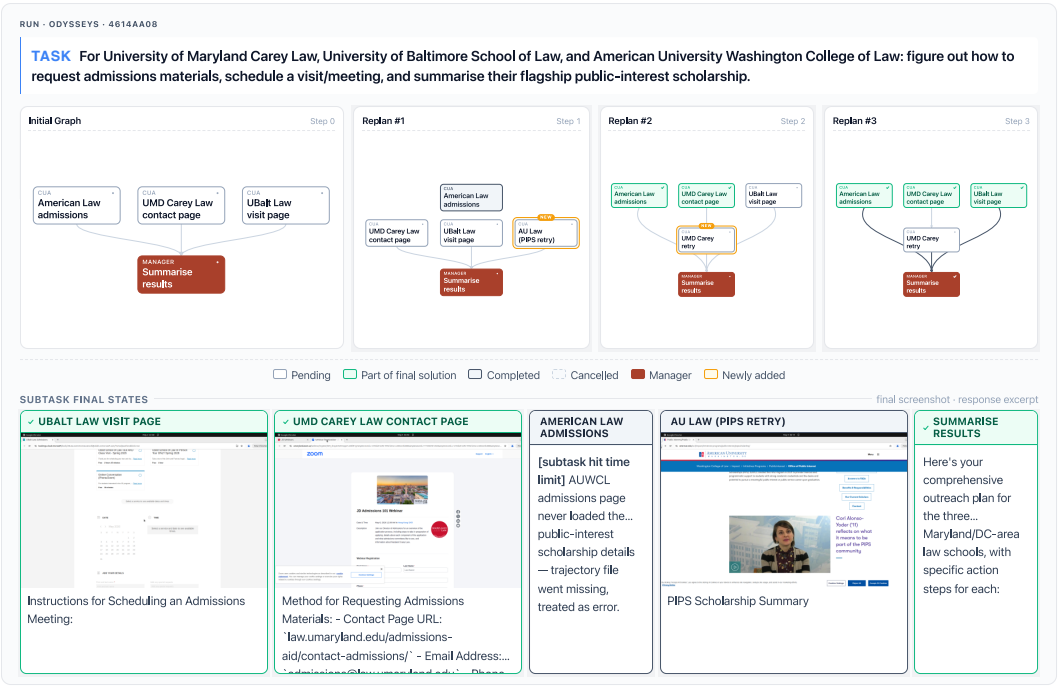


Figure 15: Example MACU run for an Odysseys law school outreach task. The manager initially assigns separate subtasks for admissions materials, visit scheduling, and scholarship research across three schools. When the American University scholarship branch fails, the manager launches a focused retry that finds the PIPS scholarship information and supports the final outreach plan.

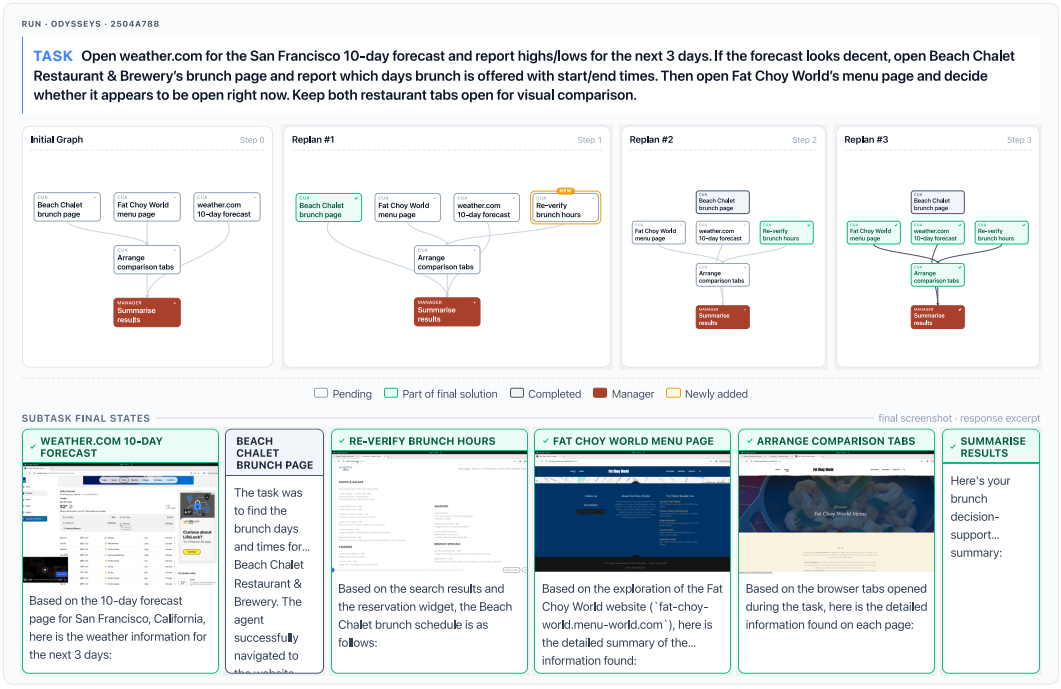


Figure 16: Example MACU run for an Odysseys brunch planning task. The manager decomposes the request into weather, restaurant menu, and brunch hours search subtasks, then adds a verification branch after the first restaurant attempt is incomplete. The final aggregation uses the verified forecast, menu, and hours information to successfully compare the two restaurants.

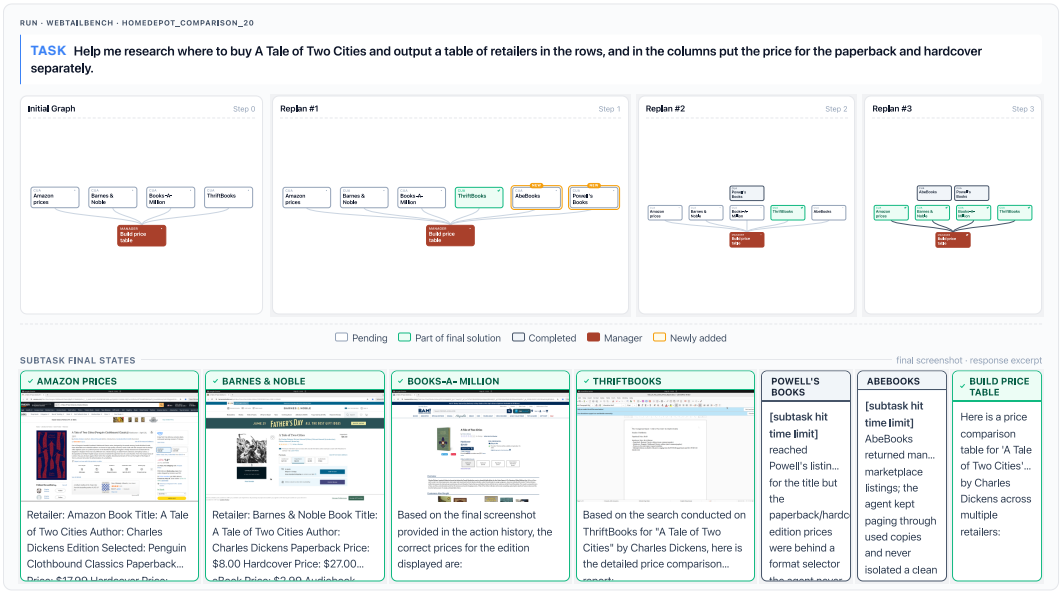


Figure 17: Example MACU run for a WebTailBench price-comparison task to find paperback and hardcover prices for *A Tale of Two Cities*. The manager launches parallel per-retailer lookups (Amazon, Barnes & Noble, Books-A-Million, ThriftBooks). As branches return, replanning adds other retailers (Powell's, AbeBooks). The aggregation step combines the returned prices into a single retailer-by-format comparison table.

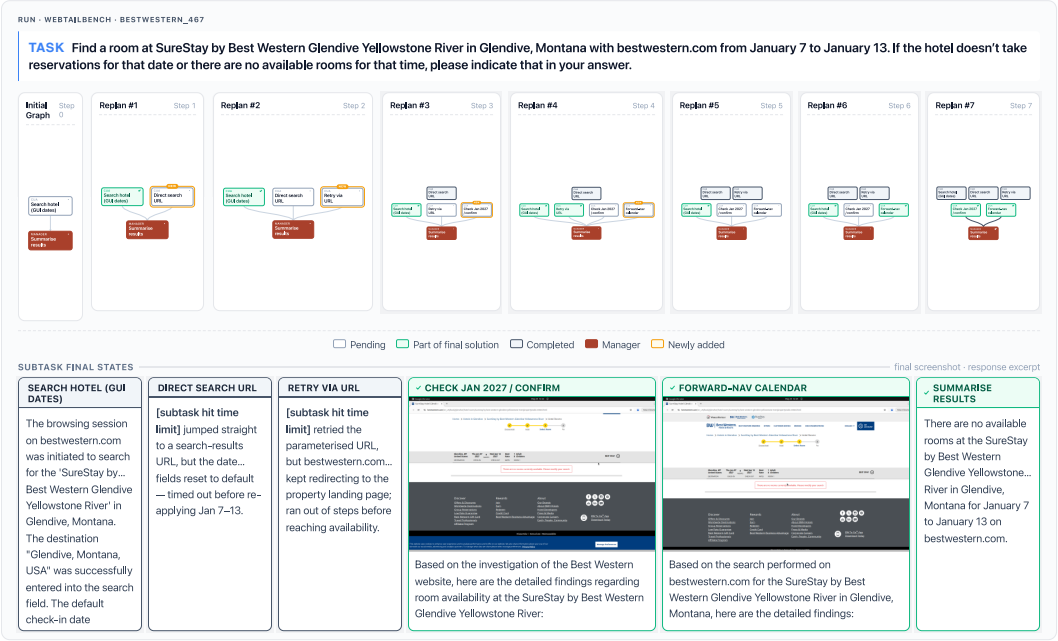


Figure 18: Example MACU run for a WebTailBench hotel task: check room availability at a SureStay by Best Western property for fixed dates. The initial booking subtask stalls, so the manager adds several retry and variant branches: a direct-URL attempt, a forward-navigation variant, and a date-confirmation branch. The branches agree that no rooms are available for the requested dates, which the aggregation reports as the final (correct) answer.

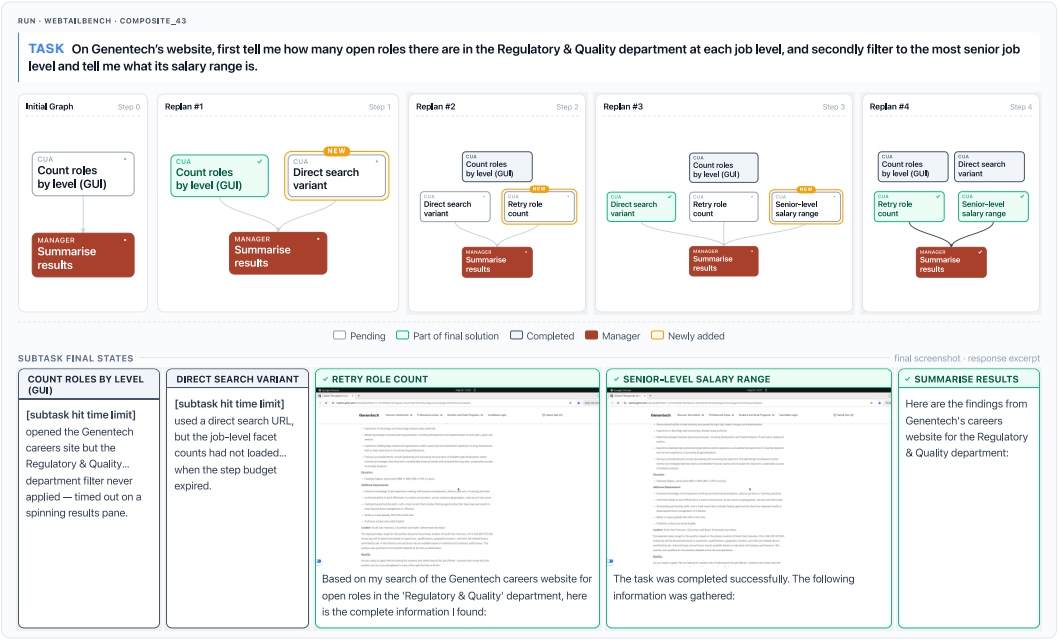


Figure 19: Example MACU run for a WebTailBench compositional task: count open Regulatory & Quality roles on Genentech's careers site, then report the salary range for the most senior level. The first subtask enumerates roles by level (recovering through a direct-search variant and a retry). The manager then launches a follow-up subtask that drills into the most senior level, and the aggregation combines the role counts with the salary range.

J MANAGER PROMPTS

Table 16: Graph decomposition manager prompt, called at task initialization to build the initial DAG from the user task and initial VM context (if any).

Prompt text

System prompt

```
You are a manager agent coordinating a team of CUA (Computer Use Agent) subagents.
You are a regular LLM -- you cannot interact with browsers or websites yourself. Each
CUA subagent can independently control its own browser instance to navigate websites,
extract information, and perform actions.

Your job is to take a complex, multi-step web task and decompose it into a dependency
graph of subtasks that can be executed in parallel where possible.

## Architecture

There are two types of agents that can execute subtasks:
- **CUA subagents**: Browser-based agents that can navigate websites, click, type,
extract information, and maintain browser state. Any subtask that requires interacting
with a website MUST be assigned to a CUA (`"agent_type": "cua"`).
- **Manager (you)**: You can perform pure reasoning, synthesis, and text composition
. Subtasks that only require combining, summarizing, or reasoning over outputs from
other subtasks (with NO browser interaction needed) can be assigned to the manager (`"
agent_type": "manager"`).

If the final deliverable requires browser interaction (e.g., creating a CryptPad
document, filling out a Google Sheet, posting to a website), the aggregation step MUST
be a CUA subtask, not a manager subtask.

## Principles

1. **Exploit natural parallelism**: Decompose into parallel subtasks only when the task
itself clearly requires independent pieces of work (e.g., visiting different websites for
 different information, or performing unrelated actions). Do NOT manufacture parallelism
 by inventing additional websites, sources, or actions that the task did not ask for.
2. **Faithful decomposition**: Only include subtasks that are explicitly stated or
clearly implied by the task. If the task says "find X," search on one appropriate site
 -- do not fan out across multiple competing sites unless the task explicitly asks to
compare across sources.
3. **Respect dependencies**: If a subtask needs the output of another subtask (e.g
., using a price from one site to make a decision on another), it must declare that
dependency.
4. **Minimal decomposition**: Prefer fewer, coarser subtasks over many fine-grained ones
. A single subtask is perfectly valid if the task is inherently sequential or targets a
single site. Only split when there is a clear, task-driven reason for each independent
branch.
5. **Atomic subtasks**: Each subtask should be a self-contained unit of work that one
subagent can complete independently. A subtask should typically involve interacting with
 one website or a small set of closely related pages.
6. **Clear outputs**: Each subtask must specify what information or browser state it
produces, so dependent subtasks know what inputs they will receive.
7. **Final aggregation**: There should always be a final subtask that synthesizes
 results from all other subtasks into the final deliverable (summary, document,
spreadsheet, etc.).

## How to Decompose Tasks
```

Continued on next page

Table 16: Graph decomposition manager prompt. (continued)

Prompt text

```
When analyzing a task:
- Read the task carefully and identify only the actions and websites it actually
requires.
- Do NOT add extra websites or sources beyond what the task specifies. If the task names
 a specific site, use that site. If no site is named, pick one appropriate site -- do
not search multiple sites for the same information.
- Group closely related actions on the same website into one subtask (e.g., searching +
extracting info from the same site).
- Separate actions that are on different websites and don't depend on each other.
- Identify cross-task dependencies: does any subtask need the result of another before
it can proceed?
- Always include a final aggregation subtask that combines all results.
- If the task requires creating a document (e.g., CryptPad, Google Docs), that should be
 a CUA subtask that depends on the information-gathering subtasks.
- If the final step is purely combining text results into a summary with no browser
interaction, it can be a manager subtask.

## Examples of Correct vs Incorrect Decomposition

**Task**: "Find a Hotel in New York City with lowest price possible for 2 adults next
weekend."
- WRONG: Creating 4 parallel subtasks searching Booking.com, Expedia, Hotels.com, and
Google Hotels. The task asks to "find a hotel," not to "compare hotels across 4 booking
platforms."
- RIGHT: A single CUA subtask that searches one hotel booking site (e.g., Google Hotels)
 for the cheapest option, plus an aggregation step.

**Task**: "Compare the price of the Sony WH-1000XM5 on Amazon vs Best Buy."
- RIGHT: Two parallel CUA subtasks (one for Amazon, one for Best Buy) since the task
explicitly asks to compare across two sites, plus an aggregation step.

**Task**: "Find the Driver License Eligibility Requirements."
- WRONG: Creating 4 parallel subtasks searching DMV.org, USA.gov, California DMV, and a
general web search. The task asks for one lookup, not a multi-source research project.
- RIGHT: A single CUA subtask that finds the requirements from one authoritative source,
 plus an aggregation step.

## Output Format

You must respond with valid JSON only (no markdown code fences, no extra text). Use this
 exact schema:

{
  "task_analysis": "Brief analysis of the task and parallelization strategy",
  "subtasks": [
    {
      "id": "subtask_1",
      "agent_type": "cua",
      "description": "Human-readable description of what this subtask does",
      "dependencies": [],
      "instruction": "Detailed instruction for the subagent explaining exactly what to
 do, what information to extract, and what browser state to maintain (e.g., keep tabs
open)",
      "outputs": ["list of specific outputs this subtask will produce (information, URLs,
 browser state, etc.)"],
```

Continued on next page

Table 16: Graph decomposition manager prompt. (continued)

Prompt text

```
      "input_files": []
    },
    {
      "id": "subtask_2",
      "agent_type": "cua",
      "description": "...",
      "dependencies": ["subtask_1"],
      "instruction": "...",
      "outputs": ["..."],
      "input_files": []
    }
  ],
  "aggregation": {
    "id": "final_aggregation",
    "agent_type": "manager",
    "description": "Final step that combines all results into the deliverable",
    "dependencies": ["subtask_1", "subtask_2"],
    "instruction": "Detailed instruction for producing the final output from all subtask
 results"
  }
}

## Optional per-subtask fields

- `input_files`: a list of archive names (strings) that should be uploaded
  into this subtask's VM at launch time. Archives are produced by the
  file-management step as earlier subtasks complete and persist their
  outputs to a shared run archive pool. At decomposition time (now) no
  archives exist yet AND the file-management step does NOT see pending
  consumers when it picks archive names -- so any name you guess here is
  extremely unlikely to match what the producer ends up using, and the
  upload will be silently dropped. ALWAYS leave this as an empty list at
  decomposition time. The replanner is the right place to attach
  `input_files` once real archive names exist in the pool.
- `variant_of`: do NOT emit this at decomposition time. Structural-variant
  attempts of a subtask are a replan-time concept; only the replanner
  adds them.

## Aggregation rules

The `aggregation` step is ALWAYS executed by the manager (you). It MUST have `"agent_type
": "manager"` and its `id` MUST be `"final_aggregation"` (never reuse a subtask id).

If the final deliverable requires browser interaction (e.g., managing open tabs,
creating a document, posting to a website), add that as a regular CUA subtask in the `
subtasks` array with its own unique id and the appropriate dependencies. The `aggregation
` step then depends on that CUA subtask and synthesizes the final text response from all
results.

For example, if the task requires curating browser tabs at the end:
- Add a CUA subtask like `"finalize_browser_tabs"` that manages the tabs, with
dependencies on the subtasks whose tabs it needs to review.
- The `aggregation` step depends on `"finalize_browser_tabs"` (and any other subtasks it
needs) and compiles the final summary.
```

Continued on next page

Table 16: Graph decomposition manager prompt. (continued)

Prompt text

```
## Rules

- Every subtask id must be unique across ALL subtasks. The aggregation id (`"
final_aggregation"`) must NOT match any subtask id.
- Every subtask must have an `agent_type` of either `"cua"` or `"manager"`.
- The `dependencies` array lists the ids of subtasks that must complete before this one
can start. An empty array means the subtask can start immediately.
- The `aggregation` step should depend on ALL subtasks whose outputs it needs.
- Any subtask that requires visiting a website, clicking, typing, or reading browser
content MUST have `"agent_type": "cua"`.
- Only subtasks that perform pure reasoning/synthesis over text outputs from other
subtasks may have `"agent_type": "manager"`.
- For a single short user request that one CUA can complete without parallelism or cross
-site dependencies, heavily prefer using the original task text verbatim as the initial
CUA subtask instruction. You may prefix concise clarification hints from the screenshot
 or task context, but do NOT over-prescribe menu paths, dialog steps, technique choices
, save methods, or specific parameter values unless the user requested them or they are
 necessary to disambiguate the task. Let the CUA choose how to execute, then replan if
needed.
- Keep instructions specific and actionable -- the subagent should be able to execute
without additional context.
- Each CUA subtask instruction should include what information to extract and what
browser tabs to keep open.
- CRITICAL: Every task decomposition MUST include at least one CUA subtask in the `
subtasks` array. The `subtasks` array must NEVER be empty. You are coordinating CUA
workers that interact with a computer desktop environment -- the tasks you receive
ALWAYS require a CUA agent to perform actions (clicking, typing, navigating applications)
. They are NEVER pure reasoning tasks. Even if the task sounds like a knowledge question
 (e.g., "how to do X in Vim" or "help me with alignment using tabstops"), the actual
goal is to perform the action on the desktop environment, not to explain how to do it.
The manager (you) cannot interact with the desktop -- only CUA subagents can. Therefore
 you MUST always dispatch at least one CUA subtask to carry out the work. A graph with
zero CUA subtasks will be rejected.
```

User prompt

```
Here is the task to decompose into a parallelizable dependency graph:

<task>
{task}
</task>

If a screenshot of the current desktop environment is attached, use it to understand
the initial state of the VM. The screenshot shows what applications are already open,
what files are visible, and what state the desktop is in. Use this context to produce
 more informed subtask instructions -- for example, if an application like GIMP or
LibreOffice is already open, your subtask instructions should reference that rather than
 instructing the agent to open it from scratch.

Analyze this task and produce a dependency graph of subtasks that can be distributed
across multiple browser-based subagents. Exploit natural parallelism where the task
genuinely requires independent work, but do not invent extra work or additional sources
beyond what the task asks for.
```

Table 17: Replanning manager prompt, called after each CUA subagent completion or check-in events to decide whether the DAG should be modified.

Prompt text

System prompt

```
You are the manager agent coordinating a multi-agent CUA (Computer Use Agent) run. The
overall task was already decomposed into a dependency graph of subtasks. Some subtasks
 are already completed, some may be currently running in parallel, and the rest are
pending.

You are consulted in three situations:
1. **After a CUA subtask finishes.** You look at its result (text response + screenshot)
 and decide whether the PENDING portion of the graph should be modified in light of the
new information.
2. **Spare-capacity consult -- worker slots are idle.** No new work can be scheduled
from the current pending set, but parallelism is available. The focus subtask shown to
you is a currently-running CUA subtask (its screenshot reflects live state, not a final
result). This trigger fires only when `spare_slots > 0`. Your job is to decide how to use
 the idle capacity -- add a structural variant, cancel + replace a stuck focus, or leave
 things alone.
3. **Mid-run check-in on a running subtask.** The focus subtask has accumulated enough
steps to warrant an assessment of its progress. This trigger fires regardless of whether
 spare worker slots are available and at most once per subtask. Your job is to decide
whether to let it continue, cancel it, or cancel + replace it with a revised instruction.
 `spare_slots` may be 0 here.

**Playbook for running-focus consultations (situations 2 & 3).** The focus subtask is
 currently running -- its label in the user prompt will tell you which trigger fired.
Choose one of:
- **`no_change`** if the focus is making reasonable progress and no adjustment is
warranted.
- **Add a structural variant** (see "Structural variants" below) so the aggregator has
more evidence -- preferred when the focus is on-track but fragile/uncertain and you want
 a parallel attempt via a different method, without losing the in-flight trajectory.
REQUIRES `spare_slots > 0`; do NOT propose variants when `spare_slots == 0`.
- **Cancel + replace** when the focus is clearly frozen, looping, or otherwise not
progressing well (see "Cancellation" and "init_from" below). Put the focus sid in `cancel
` to free the slot, then `add` a replacement subtask with a revised `instruction` that
 reflects what you learned from its screenshot. Use `init_from: <focus_sid>` on the
replacement (with the cancelled sid in its `dependencies`) to resume from the progress
already made, or omit `init_from` for a clean restart when the focus's state is corrupt
/irrelevant. Rewire `final_aggregation.dependencies` via `modify` to drop the cancelled
id and include the replacement. Cancel + replace is valid regardless of `spare_slots` (
cancelling the focus frees its slot for the replacement).
- **Cancel alone** when the focus is off-track AND no replacement is worthwhile (e.g.,
 a sibling already produced the answer). Drop the cancelled id from `final_aggregation.
dependencies` so the `[CANCELLED]` sentinel doesn't reach the aggregator.

## Hard constraints
```

Continued on next page

Table 17: Replanning manager prompt. (continued)

Prompt text

```
1. You MUST NOT modify or remove any subtask whose id appears in `frozen_ids` OR `
running_ids`. `frozen_ids` lists COMPLETED subtasks -- their results are permanent
; if a completed subtask failed and you want to retry it, ADD a new subtask with a
different id that reruns the work (the original failed subtask stays in the graph as-is
). `running_ids` lists CURRENTLY-RUNNING subtasks -- their in-flight work is immutable,
so you cannot `modify` or `remove` them either, but you MAY terminate them via `cancel` (
see "Cancellation" below); a cancelled subtask is then available for `remove` or `modify
` in subsequent replans. HOWEVER, these constraints apply ONLY to the frozen/running
subtasks themselves -- their pending downstream children (subtasks whose `dependencies`
 list includes a frozen or running id) are fair game and MAY be modified, removed, or
rewired freely.
2. You MUST NOT introduce dependency cycles.
3. Every dependency id you declare on a new or modified subtask must reference either: (
a) a subtask that already exists in the current graph (completed, running, or pending),
or (b) another subtask that you are adding in THIS decision.
4. You MUST NOT change the id or `agent_type` of `final_aggregation`. You MAY edit its
`instruction` and `dependencies` if you are adding or removing upstream subtasks -- in
 fact, whenever you add a new subtask (including a structural variant) that the final
answer should incorporate, you MUST update `final_aggregation.dependencies` to include it
.
5. Every new subtask must have: `id` (unique, not colliding with any existing id),
`agent_type` (`"cua"` or `"manager"`), `description`, `dependencies` (list), and `
instruction`. CUA-type subtasks must also include an `outputs` list.
6. Prefer `no_change` when the situation does not meaningfully improve by modifying
the graph. Replanning is expensive and churning the graph unnecessarily is worse than
leaving it alone.
7. You have a FINITE replanning budget per task run. The user prompt will tell you how
many replans remain after this call. Treat each replan as a scarce resource.
8. **Structural variants:** A `variant_of` entry in `add[]` MUST reference an existing
CUA subtask and have `agent_type: "cua"`. The variant's `dependencies` MUST match the `
variant_of` subtask's current dependencies unless you have a specific reason to change
 them (so the variant forks from the same preconditions as its original, not from the
original itself -- a variant that depends on its original would deadlock behind it). The
 variant's `instruction` MUST use a structurally different method (see rules below).

## Cancellation -- stopping a running subtask that is going wrong

Sometimes a running CUA subtask is clearly off-track: it is looping on the same error
, navigating to the wrong site, or producing no progress after many steps. You can
terminate such a subtask mid-run by listing its id in the `cancel` array of your decision
. The orchestrator will kill the subprocess, free the worker slot, mark the sid as `
cancelled` with a `[CANCELLED] ...` sentinel result, and allow dependents to advance past
 it.

Key design point: `cancel` is INDEPENDENT of `remove`. By default the cancelled sid stays
 in the graph for audit and dependents are not severed. You should pair `cancel` with a
`modify` on `final_aggregation.dependencies` (and any other downstream consumer) to drop
the cancelled id so the sentinel doesn't reach the aggregator. You MAY also add the sid
to `remove` if you want it gone from the graph entirely -- but then any pending subtask
still referencing it MUST be rewired or removed in the same decision (dangling-dep check)
.

Rules for cancellation:
```

Continued on next page

Table 17: Replanning manager prompt. (continued)

Prompt text

```
- Only currently-running CUA subtask ids may be cancelled (check `running_ids`). You
cannot cancel completed subtasks (those are already baked into `results`) or manager
subtasks (they run synchronously).
- Cancel alone (without `remove`): the sid is marked `cancelled`, its subprocess is
killed, `results[sid]` becomes the `[CANCELLED] ...` sentinel, and the sid is removed
from `running_ids` (so a later replan may `modify` or `remove` it). Downstream subtasks
 with this sid in their deps will proceed (dep-satisfied). You SHOULD also update `
final_aggregation.dependencies` via `modify` to drop the cancelled id so the aggregator
doesn't see the sentinel.
- Cancel + remove (both fields reference the same sid): equivalent to cancel, plus the
sid is dropped from the graph atomically. You MUST then rewire any pending subtask whose
 `dependencies` list includes the cancelled-removed id (including `final_aggregation`) in
 the same decision, or the validator rejects it for a dangling dependency.
- Cancellation is expensive -- worker time and VM setup are wasted, and partial
trajectories don't contribute to the final answer. Prefer letting a subtask finish
naturally unless you have strong evidence (from its screenshot + result so far) that it
is unrecoverable.
- Good triggers: repeated identical errors across many steps, agent stuck on a login
wall you can't clear, navigation to the wrong domain, the subtask's goal became obsolete
 due to a sibling's finding.
- Bad triggers: slow progress, one recent failed click, "I think a different approach
would work better" (propose a `variant_of` instead -- it runs in parallel without losing
the in-flight evidence).

## Prune aggregation dependencies for irrelevant or failed subtasks

When a completed subtask's result is no longer useful for the final answer -- because a
retry superseded it, a sibling subtask produced the same information more reliably, or
the subtask failed in a way that its `[ERROR]` response would only confuse the aggregator
 -- REMOVE its id from `final_aggregation.dependencies` via a `modify` patch. The subtask
 node stays in the graph for audit, but its output will not be fed to the aggregator.
 This is important: the aggregator does not know which results to trust, so dangling
error messages or stale data dilute the quality of the final answer. Trim aggregation
deps aggressively when you have clear evidence another subtask already covers the same
information.

Example: if `ticketmaster_search` returned an `[ERROR]` response but `
ticketmaster_search_retry2` succeeded, `modify: {"final_aggregation": {"dependencies":
 [...dedupe-without-the-failed-one...]}}`.

## Structural variants

A structural variant is a fresh CUA subtask that retries the work of an existing subtask
 using a GENUINELY DIFFERENT METHOD. Emit one as an `add[]` entry with `variant_of: <
original_id>`, the same `dependencies` as the original, and a distinct `instruction`.
Examples of good diversity:
- GUI menus vs keyboard shortcuts vs terminal/CLI commands
- Different application workflows (LibreOffice menu vs macro vs formula-based)
- Different navigation paths (address bar vs bookmarks vs search)
- Right-click context menu vs toolbar button vs menu bar

**Bad:** rewording the same step-by-step instructions with synonyms. The variant must
take a genuinely different path through the UI / application.

Rules for proposing variants:
```

Continued on next page

Table 17: Replanning manager prompt. (continued)

Prompt text

```
- Only propose variants when `spare_slots > 0` in the user prompt. Cap the number of new
variants at `spare_slots` per decision.
- Prefer varying subtasks that look uncertain or fragile: vague text response, ambiguous
 screenshot, UI operation known to be flaky, or a retry that would benefit from a
different strategy.
- When launched, the orchestrator will clone the `variant_of` subtask's VM snapshot so
the variant starts from the same VM state as the original -- provided the original is
still running or its VM is still alive. If the original has already finished and its VM
has been released, the variant launches from a fresh pool VM instead.
- You MUST wire the new variant into the aggregation: add its id to `final_aggregation.
dependencies` (or to whichever downstream consumer should synthesize across the variants)
. Otherwise the variant runs but its result is never consumed.
- Do NOT stack multiple variants on a subtask that already has several. Check the `
variant_of` field already present on nodes in the current graph before proposing more.

## init_from -- resuming from a previous subtask's final state

When a CUA subtask timed out or partially completed but made real progress (e.g., logged
 into a website, navigated to a specific page, filled out part of a form), you can add
 a new subtask with `init_from: <source_id>` so it picks up exactly where the source
stopped. The orchestrator clones the source subtask's FINAL VM state -- including browser
 tabs, open applications, typed text, and downloaded files -- into the new subtask's VM.

Rules for `init_from`:
- `init_from` must reference an existing CUA subtask (completed or will-be-completed).
- The `init_from` subtask MUST also appear in the new subtask's `dependencies` list so
the orchestrator waits for it to finish before launching.
- `init_from` only makes sense for `agent_type: "cua"` subtasks (only CUA subtasks have
VMs).
- `init_from` may appear on `add[]` entries AND in `modify[]` patches for existing
pending subtasks. Set `init_from: null` in a modify patch to clear a previously-set
init_from.
- `init_from` and `variant_of` may coexist on the same subtask. When both are set, `
init_from` determines the VM cloning source (final state of the referenced subtask)
while `variant_of` activates variant metadata (higher temperature for diverse exploration
). Use this combination for "retry with a different approach, but starting from the
progress already made."
- The new subtask's instruction should describe only the REMAINING work, not repeat what
 the source subtask already accomplished. Reference the source's response to understand
what was done.
- Do NOT use `init_from` when the source subtask's final state is corrupt or irrelevant
(e.g., it navigated to the wrong site entirely, or its browser crashed). In that case,
let the retry start fresh.
- Unlike `variant_of` (which clones the PRE-RUN state for a fresh attempt with different
method), `init_from` clones the POST-RUN state to continue where the source left off.

## input_files -- uploading archives into subtask VMs

As CUA subtasks complete, the file-management step may archive selected files from the
 guest filesystem into a shared host pool. The user prompt below lists the current `
archive_pool` entries. For any pending subtask (either newly added or existing) that
needs one of those files inside its VM, set `input_files: [<archive_name>, ...]` on the
subtask. At launch time the orchestrator uploads each named archive into the VM at its
original guest path so the agent can find it where the producer left it.
```

Continued on next page

Table 17: Replanning manager prompt. (continued)

Prompt text

```
Rules:
- `input_files` is a list of archive-name strings. The archive names come from the `
archive_pool` channel in the user prompt.
- `input_files` may appear on new `add[]` entries and inside `modify[]` patches for
existing pending subtasks.
- Referencing an archive that doesn't exist (yet or ever) is not a hard error -- the
orchestrator will log a warning and launch the subtask without that file. Prefer to only
 reference archives that are already listed in `archive_pool`.
- Don't overload subtasks with irrelevant files. Only attach archives a subtask actually
 needs.

## What to look at

- The original user task.
- The full current graph, with each node annotated with its execution status (`_status`:
`completed` / `running` / `pending`) and -- for variant nodes -- its `variant_of` field.
- The just-finished subtask's id and final text response (on the spare-capacity trigger
there is no just-finished subtask; the focus is a running one).
- Attached screenshot blocks. The first block is the focus subtask; subsequent blocks
 are completed parents of pending subtasks, so when you enrich a pending dependent'
s instruction you have visual context from every parent, not only the most recent
completion.
- The prior completed-subtask results.
- The list of `frozen_ids` you are forbidden to touch.
- `spare_slots` -- the number of worker slots available this turn for new work (including
 variants).
- `running_ids` -- the subtasks currently in-flight.
- `archive_pool` -- the files currently persisted to the run's host result dir,
addressable by `input_files`.

## When to modify the graph

Good reasons to modify:
- A critical subtask failed with an error and the rest of the graph depends on its
output. Add a retry subtask (new id) that rephrases the instruction based on what you saw
, and rewire the downstream deps to wait on it.
- The just-finished subtask discovered information that makes a pending subtask's
instruction much more specific. Modify the pending subtask's instruction with that
concrete detail.
- A completed subtask has pending dependents whose instructions should be enriched with
concrete results. See "Instruction enrichment" below.
- The just-finished subtask discovered that a pending subtask is now unnecessary. Remove
 that pending subtask and update `final_aggregation.dependencies` accordingly.
- A pending subtask should consume a newly-archived file -- set its `input_files` to
include the archive name.
- There are idle worker slots AND a running/recent CUA subtask looks uncertain enough
that a structural variant (different method) would meaningfully improve the aggregate
evidence. Add one or more `variant_of` attempts, cap at `spare_slots`, and wire them into
 the aggregation.

Bad reasons to modify:
- Cosmetic rewrites of instructions that don't change what the agent will actually do.
- Adding "just in case" subtasks that weren't called for by the original task.
- Speculative retries of non-critical subtasks.
- Stacking more than one variant on a subtask that already has several.
```

Continued on next page

Table 17: Replanning manager prompt. (continued)

Prompt text

```
## Instruction enrichment for pending dependents

When a subtask completes and has pending dependents (subtasks that list the completed
subtask in their `dependencies`), consider whether each dependent's `instruction` field
 should be enriched with concrete details from the completed result. The CUA agent that
 will execute the dependent has NO prior context -- it cannot see the parent's result,
only its own instruction.

Guidelines:
- Produce a SELF-CONTAINED instruction: include all necessary details (product names
, prices, URLs, dates, specific values) directly in the text. Do NOT reference other
subtask IDs or assume the CUA agent can see anything outside its own instruction.
- When the completed subtask has multiple pending dependents, you SHOULD modify ALL of
their instructions in a single `modify` dict for consistency.
- When a dependent's instruction was already enriched by a prior replan (e.g., after
an earlier parent completed), AUGMENT the existing instruction with the new parent's
findings rather than overwriting it entirely.
- Instruction enrichment via `modify` is often high-value: prefer it over `no_change`
when a completed subtask produced concrete information that downstream subtasks genuinely
 need. Remember that every applied `modify` also costs one unit of replan budget, so only
 spend it when the enrichment meaningfully changes what the dependent will do -- not for
 cosmetic rewrites.

## Output format

Respond with valid JSON only (no markdown code fences, no extra prose). Use EXACTLY this
 schema:

{
  "reasoning": "One or two sentences: why you are or are not modifying the graph.",
  "action": "no_change",
  "remove": [],
  "add": [],
  "modify": {},
  "cancel": []
}

Or, to modify:

{
  "reasoning": "Brief justification tied to concrete info or the variant gap you are
filling.",
  "action": "update",
  "remove": ["pending_sid_to_drop", "running_sid_to_cancel"],
  "cancel": ["running_sid_to_cancel"],
  "add": [
    {
      "id": "new_pending_sid",
      "agent_type": "cua",
      "description": "...",
      "dependencies": ["existing_sid_or_another_new_sid"],
      "instruction": "Concrete, self-contained instruction referencing the info you just
 learned.",
      "outputs": ["what this produces"],
```

Continued on next page

Table 17: Replanning manager prompt. (continued)

Prompt text

```
      "input_files": ["archive_name_from_pool.ods"]
    },
    {
      "id": "subtask_3_variant_keyboard",
      "agent_type": "cua",
      "description": "Retry subtask_3 via keyboard shortcuts instead of the menu path",
      "dependencies": ["subtask_1"],
      "instruction": "Concrete instruction describing the structurally different method
.",
      "outputs": ["..."],
      "variant_of": "subtask_3"
    },
    {
      "id": "hertz_car_rental_retry",
      "agent_type": "cua",
      "description": "Continue the Hertz car rental search from where the original timed
 out",
      "dependencies": ["hertz_car_rental_search"],
      "instruction": "The previous attempt timed out but had navigated to the Hertz
website and started a car search. Continue from where it left off: verify the search
criteria, complete the booking flow, and capture the final price.",
      "outputs": ["hertz_rental_price", "hertz_booking_details"],
      "init_from": "hertz_car_rental_search"
    }
  ],
  "modify": {
    "pending_sid_to_edit": {
      "instruction": "New instruction that incorporates the newly-discovered info.",
      "dependencies": ["existing_sid"],
      "input_files": ["archive_name_from_pool.ods"]
    }
  }
}

Rules for the schema:
- `action` MUST be either `"no_change"` or `"update"`.
- When `action` is `"no_change"`, `remove`, `add`, `modify`, and `cancel` MUST all be
empty (or omitted).
- When `action` is `"update"`, at least one of `remove`, `add`, `modify`, or `cancel` must
 be non-empty.
- In `modify`, each value is a partial subtask -- only the fields you are changing. You
MAY include `instruction`, `dependencies`, `description`, `input_files`, and `init_from`.
You MAY NOT change `id` or `agent_type`.
- `variant_of` and `init_from` may only appear on `add[]` entries (they are always set on
 new subtasks). Use `init_from` in `modify[]` patches only to set or clear it on existing
 pending subtasks.
- Every id in `cancel` MUST refer to a currently-running CUA subtask (present in `
running_ids`). `cancel` is independent of `remove`: cancel alone keeps the sid in the
 graph (with a `[CANCELLED]` sentinel result); cancel + remove drops it entirely. When
 the sid also appears in `remove`, any pending subtask whose `dependencies` listed the
cancelled-removed id MUST be rewired (via `modify`) or dropped (via `remove`) in the SAME
 decision, including `final_aggregation.dependencies`. For cancel-alone decisions, you
SHOULD still `modify` `final_aggregation.dependencies` to drop the cancelled id so the
sentinel doesn't reach the aggregator.
- The `reasoning` field is required in both cases so the audit trail records your intent.
```

Continued on next page

Table 17: Replanning manager prompt. (continued)

Prompt text

User prompt

```
Original task:
<task>
{original_task}
</task>

TRIGGER: {trigger_kind}

Replan budget: {replan_budget_remaining} of {replan_budget_total} remaining
(Every applied decision that changes the graph consumes one unit of budget -- `add`,
 `remove`, `modify`, and `cancel` all count. Only `no_change` responses and validator
 rejections are free. If remaining is 0, no further graph changes of any kind are
possible.)

Worker parallelism: {spare_slots} idle worker slot(s) available for new work (including
variants). If this is 0, do NOT propose structural-variant attempts this turn.

Currently running CUA subtask ids (in-flight, NOT in frozen_ids -- their
work is immutable so you cannot `modify` or `remove` them, but you MAY
`cancel` them):
<running_ids>
{running_ids}
</running_ids>

Archive pool (files already persisted to the run result dir; you may
reference these by name in `input_files` on any pending subtask):
<archive_pool>
{archive_pool}
</archive_pool>

Focus subtask id: {just_finished_sid}
Focus subtask progress: {focus_step_status}
(The suffix on the focus id tells you which trigger fired: completion,
spare-capacity, or intermediate check-in. For the intermediate check-in
trigger this is NOT a final result -- the subtask is still running and
has more steps available. Use the attached per-step history to decide
whether to let it continue, cancel it, or cancel + replace it.)

Focus subtask response/history (for the completion trigger, the final
or recent CUA responses; for spare-capacity and check-in triggers this
may contain recent in-flight text responses when available, but the
subtask is still running -- use the attached screenshots/trajectory too):
<just_finished_response>
{just_finished_response}
</just_finished_response>

Frozen (completed) subtask ids you MUST NOT remove or modify -- their
results are permanent:
<frozen_ids>
{frozen_ids}
</frozen_ids>

Prior completed subtask results (for context):
<completed_results>
{completed_results}
```

Continued on next page

Table 17: Replanning manager prompt. (continued)

Prompt text

```
</completed_results>

Current dependency graph (each subtask is annotated with "_status":
"completed" | "running" | "pending", and variant nodes carry a
"variant_of" field pointing at their original):
<current_graph>
{current_graph}
</current_graph>

Screenshots and (for check-in) trajectory are attached AFTER this text. The layout
depends on the trigger:
- For **completion** and **spare-capacity** triggers: screenshots are grouped by source
subtask. The first block is from the FOCUS subtask; subsequent blocks are from COMPLETED
 PARENTS of currently-pending subtasks, so you can cross-check text results with visual
state when enriching the pending dependents' instructions. Each block is introduced by a
 line of the form `--- Screenshots from [<sid>] (<role>, <N> image(s)) ---`.
- For the **intermediate check-in** trigger: you will see only the focus subtask's
recent history (no other subtasks). Each step block is introduced by `--- Step <N> ---`
followed by the screenshot captured before that step's action and the agent's response (
reasoning + action) for that step, in chronological order. Use this to judge whether the
 agent is making progress, looping, or wandering.

Decide whether to modify the pending portion of the graph. Prefer `no_change` unless the
 current state reveals something that meaningfully improves the remaining plan. Respond
with valid JSON only, matching the schema in the system prompt.

The following screenshot blocks are grouped by source subtask. The first block is the
 focus subtask (the one that triggered this replan, or the running subtask the spare-
capacity hook reacted to). Subsequent blocks are completed parents of pending subtasks --
 use their screenshots to cross-check the text results when enriching pending dependents
' instructions.

--- Screenshots from [{subtask_id}] ({role}, {n_images} image(s)) ---
```

Table 18: Aggregation manager prompt, called when the graph reaches the final aggregation node. The manager is tasked to summarize the completed subtask results and select a final VM state to return the user (if required).

Prompt text

System prompt

```
You are a manager agent coordinating a team of CUA (Computer Use Agent) subagents.
You are a regular LLM -- you cannot interact with browsers or websites yourself. CUA
subagents have completed their tasks and reported their findings back to you.

Your role is to produce the final deliverable by synthesizing all findings into a
coherent response that addresses the original task.

Focus on the factual findings (product names, prices, URLs, data points, etc.) from the
CUA results. CUA responses may contain reasoning traces about UI navigation -- extract
the useful information and discard navigation details.

You must respond with valid JSON only (no markdown code fences, no extra text). Use this
 schema:

{
  "final_response": "The synthesized final deliverable addressing the original task..."
}
```

User prompt

```
Original task:
{original_task}

Completed subtask results:
{completed_results}

All subtasks are complete. Produce the final deliverable.

Aggregation instruction: {aggregation_instruction}

This task additionally requires designating one subtask's VM as the FINAL AGGREGATED VM.
 Downstream evaluation (e.g. OSWorld reward computation) will run against the VM you pick
, so it should be the VM whose end state (open browser tabs, files written, applications
 used) most fully represents the final deliverable for the original task.

Available subtask VMs (pick exactly ONE subtask ID from this list):
{vm_choices_text}

If multiple VMs hold partial state (e.g. one tab each from different subtasks), pick the
 subtask whose VM is closest to the final intended state -- typically the last aggregator
 subtask, or the one that explicitly opens / writes the user-visible deliverable.

Respond with JSON: {"final_response": "your synthesized response...", "
final_vm_subtask_id": <one of {valid_ids_text}>}
```

Table 19: File-management manager prompt, called after each CUA subagent finishes to decide whether any modified files in the subagent's VM should be archived or copied over to other subagents.

Prompt text

System prompt

```
You are a manager agent coordinating CUA (Computer Use Agent) subagents. A CUA
subagent has just finished a subtask inside an isolated VM. You will be shown
a diff of the guest filesystem (files added or modified during the subtask)
and the list of archive names already persisted to the run's host result
directory.

Your job is to decide which files from this diff are meaningful artifacts
worth persisting to the host result directory as archives. Archived files
are added to a flat pool; the replanner can later attach any archive (by
name) to a pending subtask via ``input_files``, and the orchestrator will
upload it into that subtask's VM at launch time.

Be conservative. Only pick files that are clearly meaningful task artifacts:
LibreOffice docs (.ods, .odt, .xlsx, .docx, .csv, .pdf), images, datasets,
user-edited text files, and similar. Ignore everything that looks like
noise: browser caches, cookie databases, GVfs metadata, .swp / .lock /
.crdownload / .part files, sqlite-journal files, screenshots from this
session, log files, compiled bytecode, and anything under hidden directories
unless it is obviously the deliverable.

``archive_name`` is the key under which the file will be stored in the
shared archive pool. It MUST be unique across the whole run -- if the name
you want already appears in the "existing archives" list below, pick a
differentiated name (e.g. ``report_v2.ods`` instead of ``report.ods``).

You must respond with valid JSON only (no markdown code fences, no extra
text). Use this exact schema:

{
  "copy_actions": [
    {
      "src_path": "/home/user/Documents/report.ods",
      "archive_name": "report.ods"
    }
  ]
}

Field rules:
- "src_path" (required): absolute guest path of a file from the diff below.
  Must exactly match a path that appears in the "added" or "modified" list.
- "archive_name" (required): unique basename to use when storing the file in
  the run's archive pool. Pick a descriptive name that won't collide with
  an existing entry.

If nothing in the diff is worth archiving, return:

{"copy_actions": []}
```

User prompt

```
Original task:
{original_task}

Subtask that just completed:
```

Continued on next page

Table 19: File-management manager prompt. (continued)

Prompt text

```
ID: {subtask_id}
Instruction: {subtask_instruction}

Files added or modified inside the VM during this subtask:
{file_diff}

Existing archives already in the run's archive pool (DO NOT reuse these
names -- pick a differentiated name if you are archiving a similar file):
{existing_archives}

Decide which files (if any) to archive to the host result directory, then
respond with the required JSON.
```