HYBRID COMPUTER USE: A UNIFIED PROGRAMMATIC, SEMANTIC, AND VISUAL INTERFACE FOR GENERAL AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Computer-use agents increasingly tackle tasks ranging from everyday desktop activities to long-horizon professional workflows. Yet interfaces centered on screenshots and mouse-and-keyboard actions can impose a human-interaction prior, requiring repeated screen interpretation while leaving the computer's semantic and programmatic capabilities underexposed. We argue for exposing these capabilities and letting models choose and combine them. We introduce Hybrid Computer Use (HCU), a unified interface for general agents that combines three complementary channels: programmatic interaction through code and command-line tools, semantic interaction through accessibility (AX), and visual interaction through graphical user interfaces (GUI). Our HCU supports efficient AX interaction and guards against stale-state actions during channel switching. A supplementary HCU skill guides agents in selecting and coordinating channels to use the unified interface effectively. Our designed AX tools return compact observations, execute actions in batches, and report their effects as AX diffs, so the model needs fewer round trips. On OSWorld 2.0, HCU achieves the highest Partial and Binary scores among the compared systems. Analyzing agent trajectories, we find that all evaluated models improve with HCU yet allocate work across the channels differently and consistently outperform compared baselines in terms of both efficiency and task success, highlighting the value of unified computer-use interfaces that allow flexible channel selection and combination.

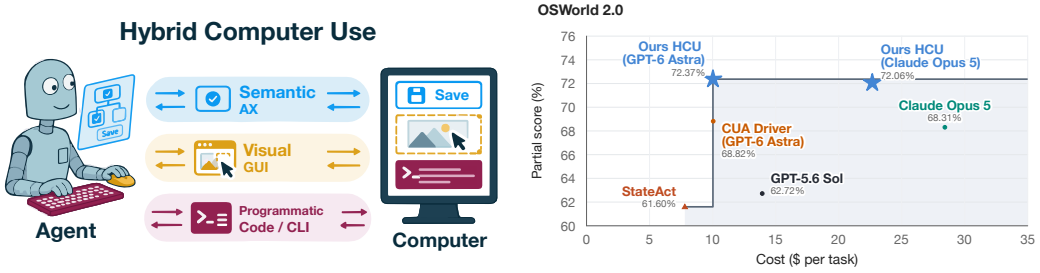


Figure 1: Left: HCU's programmatic, semantic, and visual channels. Right: OSWorld 2.0 results. Stars mark HCU and the staircase the displayed-point Pareto frontier.

1 INTRODUCTION

AI agents are increasingly able to carry out user requests by operating software applications on computers directly. Computer-use research has expanded from everyday desktop tasks in OSWorld 1.0 (Xie et al., 2024) to long-horizon workflows and specialized professional software in OSWorld 2.0 (Yuan et al., 2026). As tasks become more involved, agents must sustain progress across many operations, making interaction efficiency important alongside task performance.

Most existing computer-use interfaces mirror how humans operate software: inspect the screen, locate a UI control, and use the mouse or keyboard. While this approach offers broad coverage

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.

with human-like perception and action, it also imposes a strong human-interaction prior on the action space of a frontier Large Language Model (LLM), overlooking more efficient channels for interacting directly with the system. An agent with existing computer-use interface may spend substantial effort precisely locating a field by screen coordinates even when the application exposes its identity and value through accessibility data, or manipulate data through repeated click-and-drag actions when code could process it directly.

Modern computer systems offer a wide range of interaction tools that fall into three complementary channels: visual interaction through GUIs, structured semantic access through accessibility APIs, and direct programmatic access through code and command-line tools such as Bash. Accessibility (AX), originally developed for assistive technologies and now also used for automation, exposes application UI controls, their properties, and supported operations (The Qt Company, 2026; Microsoft, 2025a). Code and command-line tools provide direct access to files, data, and software, while GUI interaction supplies visual evidence and control where structured access is insufficient.

Prior work (Sun et al., 2026; Li et al., 2026) combines programmatic and visual interaction but makes limited use of the semantic channel provided by AX. UFO2 combines UI Automation with GUI and application-API execution (Zhang et al., 2025), while Tactile exposes semantic actions alongside OCR and visual fallbacks (Liu et al., 2026). CUA Driver, an open-source computer-use interface, supports all three channels but favors AX-based interaction in the background to preserve the user's active desktop (Cua, 2026b). We build on these efforts with an interface-centered objective: organize reference-addressed AX actions, in-call postconditions, and post-action feedback for efficient, sustained interaction, while exposing all three channels within a general agent for model-directed selection and combination.

We introduce Hybrid Computer Use (HCU), a unified interface for general agents that combines three complementary channels: programmatic, semantic, and visual interaction (Figure 1). HCU adds AX and GUI tools to the agent's own programmatic tools by Model Context Protocol (MCP) with cross-channel action guards to keep consistent observation of current computer stage for model and HCU skill to guide the model better understand and use the exposed channels. Our designed AX channel lets the model read a compact working view, act on several targets in one call, and see the resulting changes without a separate observation. All three channels act on the same desktop, so an action in one can invalidate the evidence another is holding, action guards therefore re-check targets and layouts against live state before each input. A supplementary HCU skill explains how to use these tools and when each channel tends to help, which is designed to help model understand the input and output formats of execution and reduce execution errors.

We evaluate HCU on OSWorld 2.0 and OSWorld 1.0. On OSWorld 2.0, HCU achieves the highest Partial and Binary scores among compared systems, advancing the performance–efficiency frontier (Figure 1, right). With GPT-6 Astra, it reaches 72.37% Partial, outperforming CUA Driver by 3.55 points. With Claude Opus 5, it improves Partial by 3.75 points and Binary by 3.76 points over the model's reported GUI-based result, reaching 35.19% Binary at 20% lower Cost. On OSWorld 1.0, HCU outperforms CUA Driver across four other backbone models, each in its own harness, averaging 6.48 more Score points, 22% fewer tool calls, and 30% lower Cost. Component ablations and tool-call analyses reveal distinct channel preferences across models, yet gains consistently concentrate on tasks using HCU's tools. Within tasks, we observe that channels often play complementary roles, with screenshots supporting observation and AX enabling execution. The skill helps agents use the interface more reliably, reducing the proportion of AX and GUI calls rejected for malformed arguments from 17.37% to 1.36%.

Our contributions are threefold:

1. We introduce HCU, a unified interface for general agents that combines programmatic, semantic, and visual interaction. HCU pairs model-directed channel selection with cross-channel action guards and AX tools for compact observation, batched actions with postconditions, and post-action feedback.
2. We show that HCU achieves the highest Partial and Binary scores on OSWorld 2.0 and advances the performance–efficiency frontier, and that it outperforms CUA Driver in both task performance and efficiency across five backbone models and four agent harnesses.

3. We analyze the ablations and the cross-model comparison at the level of tool calls, showing that models given the same channels divide them differently, and we distill design lessons for future computer-use interfaces.

2 Related Work

From GUI-based to hybrid computer use. Most computer-use agents observe screenshots and act through mouse-and-keyboard input. Work in this line improves grounding and planning, for example by routing GUI localization to specialist models and refining plans hierarchically (Agashe et al., 2025), and most results reported on OSWorld 1.0 and OSWorld 2.0 come from such screenshot-based loops (Xie et al., 2024; Yuan et al., 2026; OSWorld Team, 2026). Hybrid systems add programmatic access beside this visual loop. Some build dedicated multi-agent systems: CoAct-1 uses an orchestrator to delegate subtasks to a Programmer and a GUI Operator (Song et al., 2026), and StateAct prioritizes direct program-state access and delegates visual interaction to a GUI subagent (Yang et al., 2026). Others place GUI interaction inside a general agent: the GUI-as-Tool setup in ALE and the desktop plugin evaluated in WeaveBench expose screenshots and mouse-and-keyboard actions as tools alongside the agent's programmatic tools (Sun et al., 2026; Li et al., 2026). More broadly, SWE-agent shows how agent-computer interface design affects agent behavior and performance in software engineering (Yang et al., 2024). Like ALE and WeaveBench, HCU extends a general agent and leaves the division of work among channels to the model, while adding a semantic channel beside the programmatic and visual ones.

Semantic interaction with computers. Interfaces that expose semantic information differ in how they identify targets and execute actions. The original OSWorld 1.0 set-of-marks baseline uses AX information to identify targets, but maps element identifiers to coordinates for GUI input (Xie et al., 2024; OSWorld Contributors, 2024). UFO2 combines UI Automation and visual control detection with GUI and application-API execution (Zhang et al., 2025). Tactile exposes compact desktop candidates and semantic actions with OCR and visual fallbacks (Liu et al., 2026). CUA Driver, our baseline, supports element-addressed actions and coordinate input under a best-effort background policy (Cua, 2026b). In browsers, Playwright MCP provides accessibility snapshots and explicit target references (Microsoft, 2025c), while Cocoa-Agent combines DOM queries and element-addressed actions with screenshots and programmatic tools (CocoaBench Team, 2026). HCU uses explicit target references on desktop AX and adds batched AX actions with postconditions in the same call, cross-channel action guards, and AX diffs as post-action feedback.

Benchmarks and efficiency. OSWorld 1.0 is a widely used benchmark for computer-use agents in real desktop environments (Xie et al., 2024), and OSWorld 2.0 extends this evaluation to long-horizon real-world workflows (Yuan et al., 2026). Early evaluations centered on GUI interaction; recent benchmarks increasingly examine hybrid interaction that combines GUI and programmatic tools. RecreationWorld uses application recreation for hybrid-agent training and evaluation, combining interface exploration, coding, and verification (Bai et al., 2026). Following cost-aware agent evaluation (Kapoor et al., 2024), we report task performance together with resource use, using OSWorld 2.0 for the main experiment and OSWorld 1.0 for the ablations and cross-model comparison.

3 Hybrid Computer Use Interface

HCU combines three interaction channels in one agent's task context: programmatic, semantic, and visual (Figure 2, top). The programmatic channel uses the agent's own code, shell, and file tools. HCU adds the semantic and visual channels through two tools exposed via the Model Context Protocol (MCP) (Model Context Protocol, 2025): the AX channel reads application accessibility trees and executes reference-addressed actions, while the GUI channel takes screenshots and sends mouse and keyboard input (Appendix Table 4). All three channels act on the same computer. Action guards check AX targets and GUI layouts against live state before input. A supplementary skill explains how to use the interface.

The agent uses HCU within its existing observation–action loop. Given the task and available observations, the model chooses a channel to inspect state or act; the selected tool executes the request and returns its result to the same context, from which the model decides whether to continue, inspect

further, or change channels. Observation and execution need not use the same channel: a screenshot can supply visual context for a subsequent AX action. Channel selection remains model-directed. HCU adds no planner, subagents, or loop of its own, so the same interface attaches to different agent harnesses (Appendix A.5).

The following subsections describe the AX channel's observation, execution, and feedback design (Section 3.1), the action guards that support switching between channels (Section 3.2), and the skill that explains how to use the interface (Section 3.3).

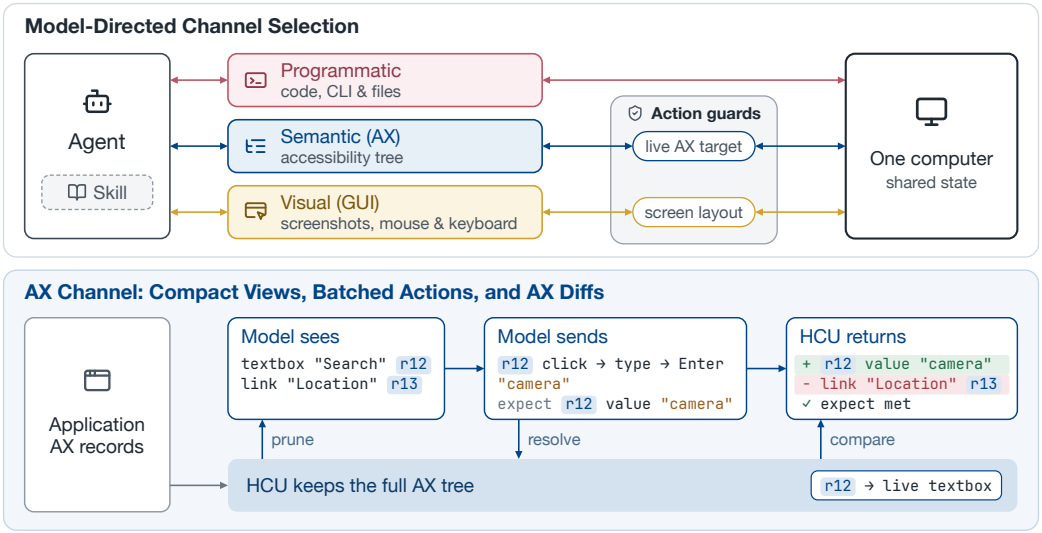


Figure 2: HCU overview. Top: channels, skill, and action guards. Bottom: a compact AX view, batched actions with a postcondition, and the returned AX diff.

3.1 Accessibility for Computer Control

AX exposes a structured representation that applications or UI frameworks supply for assistive technologies: a tree of elements with roles, names, values, states, and supported operations. It lets an agent identify and operate controls without locating every target in pixels, but using it to control a computer presents three challenges.

1. **Coverage depends on what applications expose.** Applications choose which properties and operations to expose through AX. Custom-drawn content may have no accessible elements, accessible objects need not map one-to-one to widgets, and virtualized items may be absent until rendered (The Qt Company, 2026; Microsoft, 2025b). AX therefore provides incomplete coverage of visible content and available interactions. CUA Driver addresses this by returning the tree and a screenshot together by default, so the model can use pixels when AX omits or misrepresents a target (Cua, 2026d). HCU retains the acquired AX records, marks unavailable information, and lets the model request a screenshot when semantic evidence is insufficient.
2. **Changing targets.** Controls can share names, move, disappear, or be recreated; AT-SPI explicitly marks defunct objects (GNOME Project, 2026b). A label or tree position therefore cannot establish identity across actions. CUA Driver instructs the agent to refresh element indices before acting and rejects ambiguous window targets (Cua, 2026a;c). HCU binds target references independently of the displayed view and resolves them against live elements before each action, including actions within a batch.
3. **Verifying action effects.** AT-SPI reports invocation, which does not establish that the intended state change occurred (GNOME Project, 2026a). CUA Driver verifies caller-defined postconditions through a separate operation (Cua, 2026a). HCU lets the agent compose actions and optional AX postconditions in one batch, such as selecting a checkbox and verifying its checked state before continuing, without a separate verification call.

CUA Driver prioritizes best-effort background input to preserve the user's active desktop, with foreground input available on request (Cua, 2026b). In contrast, HCU prioritizes completing the requested action, using native AX operations or focused input as needed. Its efficiency-oriented design also gives the agent more control over how work is composed: compact views reduce observation size, while batched actions, optional postconditions, and AX diffs let the agent combine execution, verification, and state feedback within a single call.

3.1.1 COMPOSABLE SEMANTIC INTERACTION

These properties motivate an AX interface that lets the agent compose actions while HCU validates targets, verifies requested postconditions, and returns evidence for continuation. Figure 2 (bottom) illustrates this interaction: a compact AX view supplies target references, a batch combines actions with optional postconditions, and post-action feedback supports the next decision.

Compact AX views and target references. HCU separates the AX records it retains from the compact AX view shown to the model. It removes structural wrappers and condenses repeated content in the view while retaining the acquired records for target identification and state comparison. Target references are bound to elements before this compression, so shortening the view does not change which element a reference identifies. The retained records also support AX diffs after an action, while `find` and scoped observations let the model retrieve targets omitted from the compact AX view. Appendix A.2 gives same-capture examples and measures the reduction in tokens.

Batched actions and postconditions. HCU lets the agent compose known actions and optional AX postconditions in a single call. The `act` mode executes the actions in order within one application window, checking each target with the AX target guard before input (Section 3.2). The agent can attach a postcondition to an action, such as requiring a checkbox to be checked before continuing, or to the batch to verify its final state. This combines batching, also used for GUI interaction (Li et al., 2026), with state verification akin to Playwright assertions (Microsoft, 2026). If an action fails, a postcondition is unmet, or window continuity is lost, HCU stops the batch and reports the stopping point and execution evidence. This lets the agent decide how to continue without blindly repeating earlier actions.

Post-action feedback. HCU returns an AX diff by default after an action call, showing changed values and target references relative to the last eligible observation delivered to the model. The agent can instead request a screenshot or execution evidence only, according to what it needs for its next decision. Returning this feedback with the action lets the agent continue without a separate observation when the information suffices (traces and samples in Appendix Figure 9 and D.3).

HCU reports only what the available evidence supports, so the agent is not led to treat input delivery as a verified effect. Like CUA Driver (Cua, 2026a), it distinguishes known non-delivery from an unknown effect after possible input. The agent uses this distinction and the returned continuation instructions to decide whether to retry, inspect state, or stop.

3.2 GUARDING ACTIONS ACROSS CHANNELS

The model observes the desktop through discrete tool outputs rather than a continuous view of its state. After an action, it may not know which earlier observations remain valid, especially when combining channels. An AX action may open a dialog over coordinates from an earlier screenshot, or a GUI action may replace a control addressed by an AX reference. Reusing that evidence can send input to an unintended target without the model noticing. HCU therefore leaves channel choice to the model while using action

Algorithm 1 Cross-channel action guards (simplified)

Input: a targeted AX batch or GUI coordinate batch A
last screenshot F, if any

for each $a \in A$ in order **do**
  if a is an AX action **then** ▷ AX target guard
    $t \leftarrow \textsc{LiveTarget}(a.\text{ref})$ ▷ $\perp$ if identity differs
    $ok \leftarrow t \neq \perp \wedge \textsc{CanExecute}(a, t)$
  else ▷ GUI layout guard
    $ok \leftarrow F \neq \perp \wedge \textsc{LiveLayout}() = F.\text{layout}$
  if not ok **then** ▷ no input for a or later actions
    return fresh evidence or a request to observe
  $\textsc{Execute}(a)$

guards to check the targets and layouts on which its actions depend immediately before input (Algorithm 1).

The AX target guard resolves each target reference immediately before execution and checks the element's identity and action-specific requirements. If the target is missing, ambiguous, or replaced, or an action-specific requirement is not met, HCU rejects the action without sending input. It does not redirect the action to a similarly named element. The GUI layout guard checks whether the layout underlying the model's chosen coordinates still matches the last screenshot. Before coordinate input, it compares the current desktop bounds, visible top-level windows, stacking order, and active window with those associated with that screenshot. If the layout has changed, HCU blocks the affected input and returns a fresh screenshot so the model can choose new coordinates.

Both guards inspect current state regardless of which channel changed it, including earlier actions in the same channel. Detected mismatches lead to explicit rejection before input, allowing the model to revise its next action (Appendix A.4; recorded rejection in Figure 11).

3.3 Skill for Effective Interface Use

HCU provides a supplementary skill that explains how to choose and compose the interface's capabilities. We distilled its instructions from recurring mistakes in development trajectories, including malformed arguments and misused target references, following the trajectory-informed refinement of usage instructions in SWE-agent (Yang et al., 2024). The skill addresses three aspects of interaction: (1) **Channel capabilities.** The skill explains what programmatic, semantic, and visual interaction can provide, including their observation scope and limitations, so the model can decide which channels to use and combine for the task. (2) **Efficient interaction.** It explains how to use target references, batch known actions with postconditions, and continue from returned feedback, requesting additional observations only when needed. (3) **Verification and recovery.** It distinguishes input delivery, local UI changes, and task outcomes; explains how to verify saved or applied results; and describes how to recover from stale targets, invocation errors, or insufficient evidence.

The skill makes these instructions concrete through rules for common invocation and recovery mistakes, while the tool schemas define argument shapes. It contains no task-specific solutions; Appendix F.2 presents its text. The skill ablation in Section 4.3 shows reduced invocation errors and improved interaction efficiency; Appendix B.6 describes instruction integration.

4 Experiments

4.1 Experimental Setup

Benchmarks. The main experiment uses the OSWorld 2.0 full set (0808 release; Yuan et al., 2026), and the ablations use OSWorld 1.0 (Xie et al., 2024); Appendix B.1 lists the task sets.

Models and comparisons. Each model always runs in the same harness: GPT-6 Astra and GPT-5.4 in Codex, Claude Opus 5 and Claude Opus 5.5 in Claude Code, Gemini 3.8 Flash in Gemini CLI, and Grok 4.7 in Grok Build. We therefore name configurations by interface and model. In HCU–CUA Driver comparisons, both interfaces use the same harness execution flow and programmatic tools within each model; auxiliary harness features are restricted to focus the comparison on the interface. The main experiment evaluates HCU (GPT-6 Astra) and HCU (Claude Opus 5) at each model's highest reasoning effort, with CUA Driver (GPT-6 Astra) as the matched baseline and reported results as context. The interface-component ablations use GPT-5.4 (xhigh); the cross-model comparison adds Claude Opus 5.5 (low), Gemini 3.8 Flash (high), and Grok 4.7 (xhigh). We gray out the official GPT-6 Astra results since they were evaluated internally on a subset of tasks. We further details all configurations in Appendix B.2.

Metrics and budgets. We report Partial (mean task score) and Binary (full-credit rate) on OSWorld 2.0 and Score on OSWorld 1.0, along with per-task resource use. The main experiment uses a 500-step limit and the ablations 100 Steps per task. We average results across rounds within each configuration. Appendix B details the rounds, protocols, and metric accounting.

4.2 Main Results

Table 1 compares HCU with results reported by the official OSWorld 2.0 website (OSWorld Team, 2026), StateAct (Yang et al., 2026), and model vendors (OpenAI, 2026; Meta, 2026a; Google DeepMind, 2026a; Kimi Team, 2026) (Appendix C.3). Among the full-set results in Table 1, HCU achieves the highest Partial (72.37% with GPT-6 Astra) and Binary (35.19% with Claude Opus 5). Only StateAct costs less than HCU with GPT-6 Astra, about $7.80 against $10.03 per task, at 61.60% Partial; the two form the performance–efficiency frontier (Figure 1, right).

Table 1: OSWorld 2.0 main results. Bold: best. * OpenAI's offline-subset result; unranked. § Pre-0808: StateAct uses 0624; Gemini/Kimi task counts unspecified.

System	Model	Partial (%) ↑	Binary (%) ↑	Out./1k ↓	Cost ($) ↓
Computer-use loop	GPT-6 Astra*	72.60	–	–	–
	Claude Opus 5	68.31	31.43	102.87	28.43
	Muse Spark 1.3	66.90	32.00	–	–
	GPT-5.6 Sol	62.72	27.34	36.14	13.94
	Gemini 3.8 Flash§	59.00	–	–	–
	Kimi K3§	58.30	–	–	–
StateAct	Claude Opus 4.8§	61.60	26.90	∼100.00	∼**7.80**
CUA Driver (Codex)	GPT-6 Astra	68.82	31.50	25.8	10.05
HCU (Codex)	GPT-6 Astra	**72.37**	32.40	**25.4**	10.03
HCU (Claude Code)	Claude Opus 5	72.06	**35.19**	118.29	22.67

With Claude Opus 5, HCU scores 3.75 points higher in Partial and 3.76 points higher in Binary than the official GUI-based result, at 20.26% lower Cost. With GPT-6 Astra, HCU outperforms the matched CUA Driver baseline by 3.55 points in Partial and 0.90 points in Binary, using 29.81% fewer tool calls at nearly identical Cost (Appendix Table 9).

4.3 Interface Components

We remove the GUI channel, the AX channel, or the skill one at a time to assess each component's contribution (Table 2).

Table 2: GPT-5.4 ablations, OSWorld 1.0. Darker red: larger losses vs. HCU. Rounds: Appendix D.1.

Configuration	Score (%) ↑	Steps ↓	Input (M) ↓	Time (s) ↓	Cost ($) ↓
HCU	65.59	28.60	0.941	351.3	2.52
w/o GUI channel	66.08	32.16	0.909	352.4	2.45
w/o AX channel	62.79	33.28	1.051	401.8	2.82
w/o skill	64.00	33.47	0.962	385.5	2.57
CUA Driver	62.30	40.74	2.181	500.8	5.65

HCU beats the baseline on every measure. HCU scores 3.29 points higher than the baseline while using 29.80% fewer Steps, 56.85% less cumulative input, 29.85% less Time, and 55.36% lower Cost. Every component contributes to this efficiency. Each one cuts Steps by 11–15%, and AX and the skill also cut Time by 12.58% and 8.88% and Cost by 10.64% and 1.79%, so the full interface needs the fewest Steps of all configurations. The following paragraphs trace the advantage to the AX channel, its design, and the skill.

AX contributes most. AX is the channel HCU adds to GUI-based computer use: it exposes the applications' control tree as named, reference-addressed targets and returns observed state changes after action batches. Added to the harness's programmatic tools and the GUI channel, it raises Score by 2.79 points and cuts Steps by 14.08%, cumulative input by 10.44%, Time by 12.58%, and Cost by 10.64%, the largest gain of any component in Score, input, Time, and Cost. The GUI channel is the other side of this comparison: without it, Score moves by +0.50 points, less than the run-to-run spread, while Steps rise 12.45% (Appendix D.1). AX thus carries the Score gain and most of the saving, and Section 4.4 shows how the two channels divide observation and execution.

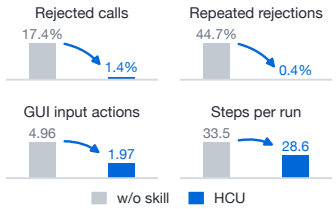


Figure 3: Skill effects.

Execution-centered AX design supports efficient interaction. HCU's AX channel integrates guarded execution, optional postconditions, and post-action state in one call (Section 3.1.1). With this channel alongside the agent's programmatic tools, HCU w/o GUI exceeds CUA Driver by 3.79 Score points while using 21.06% fewer Steps, 58.33% less cumulative input, 29.63% less Time, and 56.70% lower Cost. CUA Driver also offers element-addressed actions and separates delivery from effect (Cua, 2026a), but leaves caller-defined verification and renewed observation to separate operations. Both interfaces can sequence actions programmatically; HCU incorporates these execution and feedback semantics into the AX call itself. This comparison supports HCU's interface package as a whole, including its skill, and does not isolate the contribution of batching, guards, or feedback.

The skill improves calling reliability and interaction efficiency. The skill (Section 3.3) provides calling conventions and general rules for verification and recovery. With the skill, the share of AX and GUI calls rejected for malformed arguments falls from 17.37% to 1.36%, and the share of task-runs that repeat such rejections on the same tool while varying the arguments falls from 44.69% to 0.37% (Figure 3). Feedback selection also changes: 87% of AX action requests omit the `observe` field, selecting the default AX diff, compared with 0.2% without the skill. GUI input actions decrease from 4.96 to 1.97 per task. These behavioral changes accompany a 14.55% reduction in Steps, while Score increases by a comparatively modest 1.59 points (Appendix B.4).

4.4 Agent Behavior Across Models

We compare HCU with CUA Driver across four models (Table 3), examining whether the gains transfer and how each model obtains them. Here GPT, Claude, Gemini, and Grok denote GPT-5.4, Claude Opus 5.5, Gemini 3.8 Flash, and Grok 4.7, respectively.

HCU improves every model in Score, input, and Cost. Every model scores higher with HCU than with CUA Driver, by 2.94 to 12.90 points, and uses 18–57% less cumulative input at 8–55% lower Cost (Table 3). Steps and Time also decrease for GPT, Gemini, and Grok. For Claude, Steps remain nearly unchanged while Time increases by 6%; we examine the contrasting patterns of desktop use behind its Step count below.

Table 3: HCU versus CUA Driver on OSWorld 1.0. Resource changes are relative to CUA Driver; darker blue indicates larger improvements. Details: Appendix E.1.

	Score (%) ↑			Change with HCU (%) ↓			
Model	CUA Driver	HCU	Δ (pp)	Steps	Input	Time	Cost
GPT-5.4	62.30	65.59	+3.29	−30	−57	−30	−55
Claude Opus 5.5	68.68	71.62	+2.94	−1	−32	+6	−8
Gemini 3.8 Flash	54.67	67.57	+12.90	−17	−39	−29	−44
Grok 4.7	67.90	74.71	+6.81	−22	−18	−13	−13

Models use HCU's channels differently. All four models allocate a larger share of tool requests to HCU's AX and GUI tools than to CUA Driver (Figure 4a). Among HCU's AX and GUI requests, AX accounts for 80% for GPT, 68% for Grok, and 51% for Gemini. Claude instead uses GUI for

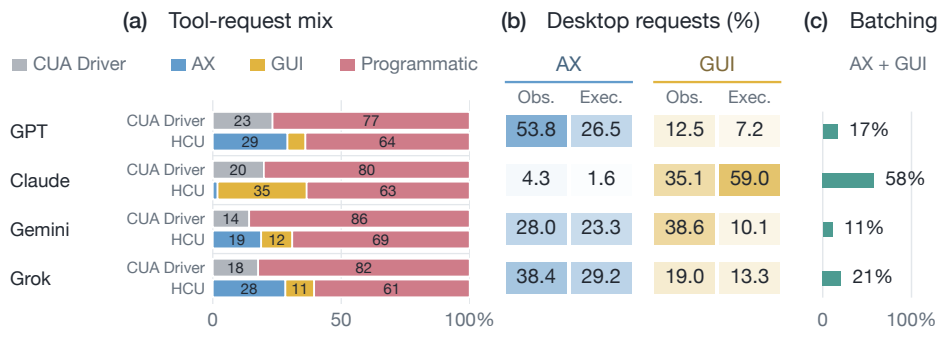


Figure 4: Interface use. (a) Mean per-task tool-request shares (Programmatic: harness tools; CUA Driver includes browser tools). (b) Within-HCU AX/GUI shares: observation (AX reads, GUI screenshots) versus execution (AX actions, GUI inputs). (c) AX/GUI acting requests with at least two actions (%).

94% of these requests (Figure 4b) and batches actions more often: 58% of its acting requests contain two or more actions, compared with 11–21% for the other models (Figure 4c).

Models combine observation and execution across channels. GPT, Gemini, and Grok combine visual observation with semantic action within the same task. On about 30% of the tasks where each model used GUI, all GUI requests were screenshots, while the model executed actions through AX (Appendix E.2). In one recorded task, a screenshot confirmed an entered value missing from AX feedback, after which the agent continued acting through AX (Appendix E.2). By leaving channel choice to the model, HCU allows observation and execution to use different channels, enabling the model to draw on their complementary capabilities as the task requires.

Score gains concentrate on tasks with AX or GUI use. Across all four models, nearly all of the overall Score gain over CUA Driver comes from tasks where HCU makes at least one AX or GUI request (Figure 5a). We group tasks by whether HCU makes at least one such request in each run and average the groups' weighted contributions across runs. The group without these requests contributes only -0.7 to +0.1 percentage points to the overall Score difference (Appendix E.3).

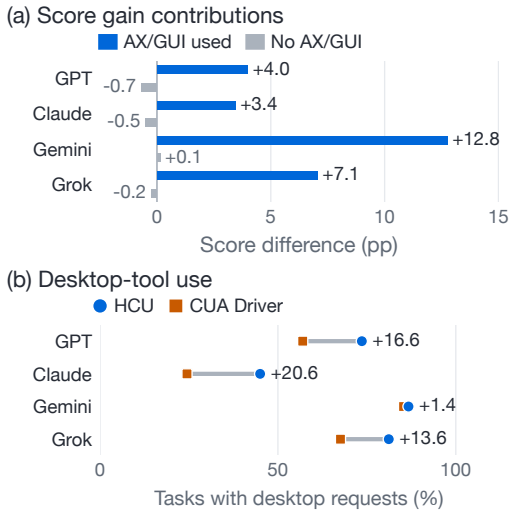


Figure 5: (a) Contributions to the HCU–CUA Driver Score difference, grouped by HCU AX/GUI use. (b) Tasks using each interface's desktop tools; labels show HCU–CUA Driver differences in percentage points.

Overall Steps mask differences in interaction efficiency. Here we observe behavior of desktop tools, *i.e.,* AX and GUI channels that control desktop state. On tasks where both configurations invoke their desktop tools, HCU uses 19–40% fewer Steps across the four models. HCU also invokes desktop tools on more tasks than CUA Driver (Figure 5b). For Claude, tasks where only HCU invokes desktop tools gain 7.6 Score points, accompanied by 53% more Steps. This group's weighted increase largely offsets the savings where both configurations use desktop tools, explaining Claude's nearly unchanged overall Steps despite its higher Score. Appendix E.3 provides the complete decomposition.

5 Discussion and Conclusion

HCU exposes programmatic, semantic, and visual interaction to a general agent and leaves their selection and combination to the model. It achieves the highest Partial and Binary scores on OSWorld 2.0, and across five backbone models and four agent harnesses it outperforms CUA Driver in both task performance and efficiency. These results suggest that interfaces built on how people operate software, inspecting the screen and using the mouse and keyboard, cost agents both performance and efficiency. Given AX, three of the four models send most of their desktop requests through it (Section 4.4). This echoes the bitter lesson (Sutton, 2019): a strong human-like interface that fixes how the computer is used may limit what increasingly capable models can do with it. The capabilities HCU exposes already exist and need no training, although what AX offers is bounded by what applications and platforms expose.

Finally, AX channel exposes element states that code can read and act on. This makes it possible to treat the desktop as a programmable environment and to bring desktop work into code execution and programmatic tool calling (Wang et al., 2024). HCU's batched actions with postconditions (Section 3.1.1) are a first step. We hope this work encourages interfaces that fully leverage what computers already offer to release the capabilities of models on computer use.

AI USE STATEMENT

We used an AI assistant for literature search, argument organization, English drafting, analysis scripts, LaTeX preparation, and figure layout; the authors reviewed all content and take responsibility for it. The left panel of Figure 1 is conceptual artwork generated with an image model and edited by the authors, not a recorded interaction. Algorithm 1 summarizes implementation checks as pseudocode. The other empirical figures use recorded data, tool requests, model-visible responses, and original application screenshots. No model is listed as an author.

ETHICS STATEMENT

Desktop actions can modify files, applications, and external services. Evaluation requires authorized environments and adherence to task restrictions. Public artifacts and traces require review for credentials, personal information, and confidential content. A dedicated desktop session does not remove authorization or destructive-action risks.

REPRODUCIBILITY STATEMENT

Appendix F reproduces HCU's tool definitions, server instructions, and skill. Appendix B gives the task sets, configurations, metrics, trace definitions, and instruction integration behind Section 4. Appendices D and E give the component and cross-model analyses; recorded examples appear beside the mechanisms and behavior they illustrate. Appendix A.2 provides the captured AX observations.

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. URL https://arxiv.org/abs/2504.00906v1. Version 1, April 1, 2025.

Shuai Bai, Jiayong Deng, Yikun Fu, Chang Gao, Xuhao Hu, Mianqiu Huang, Yizhen Jiang, Yuheng Jing, Dehui Kong, Keliang Li, Ning Li, Wanli Li, Dayiheng Liu, Dunjie Lu, Changwei Luo, Que Shen, Zheyuan Wang, Zijian Wang, Jie Wu, Gao Wu, Zhihui Xie, Rui Xie, Haiyang Xu, An Yang, Jiakang Yuan, Yanming Zhang, Jiajun Zhang, Xi Zhang, Zhenru Zhang, Zhuo Zhen, Mingkang Zhu, and Bowen Zhou. RecreationWorld: Scalable and verifiable environments for hybrid computer-use agents. arXiv preprint arXiv:2609.22000, 2026. URL https://arxiv.org/abs/2609.22000v1. Version 1, September 18, 2026.

CocoaBench Team. CocoaBench: Evaluating unified digital agents in the wild. arXiv preprint arXiv:2604.11201, 2026. URL https://arxiv.org/abs/2604.11201v2. Version 2, April 14, 2026; first submitted April 13, 2026.

Cua. Action results and postcondition verification. Cua Driver documentation, 2026a. URL https://github.com/trycua/cua/blob/5fcd67326dd0406bca823e5488cf39f47491c869/libs/cua-driver/docs/action-result-contract.md. Snapshot dated September 15, 2026; accessed September 24, 2026.

Cua. Best-effort background. Cua Driver documentation, 2026b. URL https://github.com/trycua/cua/blob/5fcd67326dd0406bca823e5488cf39f47491c869/docs/content/docs/concepts/the-no-foreground-contract.mdx. Snapshot dated September 15, 2026; accessed September 15, 2026.

Cua. Cua Driver MCP server instructions. Cua Driver source code, protocol.rs, 2026c. URL https://github.com/trycua/cua/blob/5fcd67326dd0406bca823e5488cf39f47491c869/libs/cua-driver/rust/crates/cua-driver-core/src/protocol.rs#L363-L396. Snapshot dated September 15, 2026; accessed September 26, 2026.

Cua. Capture and delivery modalities. Cua Driver documentation, 2026d. URL https://github.com/trycua/cua/blob/6d5e1db7243d094efb215a10aa2a413adcc72952/docs/content/docs/concepts/capture-and-delivery-modalities.mdx. Snapshot dated September 14, 2026; accessed September 25, 2026.

GNOME Project. Atspi.Action.do_action. AT-SPI 2 API documentation, 2026a. URL https://docs.gtk.org/atspi2/method.Action.do_action.html. Accessed September 21, 2026.

GNOME Project. Atspi.StateType. AT-SPI 2 API documentation, 2026b. URL https://docs.gtk.org/atspi2/enum.StateType.html. Accessed September 21, 2026.

Google DeepMind. Gemini 3.8 Flash: Model card, 2026a. URL https://deepmind.google/models/model-cards/gemini-3-8-flash/. Published September 2, 2026; accessed September 24, 2026.

Google DeepMind. Gemini 3.8 Flash: Model evaluation, 2026b. URL https://deepmind.google/models/evals-methodology/gemini-3-8-flash/. September 2026; accessed September 24, 2026.

Sayash Kapoor, Benedikt Stroebl, Zachary S. Siegel, Nitya Nadgir, and Arvind Narayanan. AI agents that matter. arXiv preprint arXiv:2407.01502, 2024. URL https://arxiv.org/abs/2407.01502v1. Version 1, July 1, 2024.

Kimi Team. Kimi K3: Open frontier intelligence. Technical report, 2026. URL https://github.com/MoonshotAI/Kimi-K3/blob/3cb39dfd32e51c3328e2e4b4af21341247d06c43/k3_tech_report.pdf. Revised August 6, 2026; accessed September 24, 2026.

Wanli Li, Bowen Zhou, Yunyao Yu, Zhou Xu, Yifan Yang, Dongsheng Li, and Caihua Shan. WeaveBench: A long-horizon, real-world benchmark for computer-use agents with hybrid interfaces. arXiv preprint arXiv:2606.09426, 2026. URL https://arxiv.org/abs/2606.09426v3. Version 3, July 6, 2026; first submitted June 8, 2026.

Yong Liu, Zhenyi Zhong, and Zhanpeng Shi. Tactile: Giving computer-using agents hands and feet. arXiv preprint arXiv:2607.14443, 2026. URL https://arxiv.org/abs/2607.14443v1. Version 1, July 16, 2026.

Meta. Muse Spark 1.3, 2026a. URL https://dev.meta.ai/models/muse-spark/. Accessed September 24, 2026.

Meta. Muse Spark 1.3: Evaluation methodology, 2026b. URL https://research.meta.ai/static/muse-spark-1-3-multimodal-evaluation-methodology. Accessed September 24, 2026.

Microsoft. Implementing the UI Automation Invoke control pattern. Microsoft Learn, .NET Framework documentation, 2021. URL https://learn.microsoft.com/en-us/dotnet/framework/ui-automation/implementing-the-ui-automation-invoke-control-pattern. Document date September 15, 2021; accessed September 21, 2026.

Microsoft. IUIAutomationElement::GetRuntimeId method. Microsoft Learn, Win32 documentation, 2024. URL https://learn.microsoft.com/en-us/windows/win32/api/uiautomationclient/nf-uiautomationclient-iuiautomationelement-getruntimeid. Document date February 22, 2024; accessed September 21, 2026.

Microsoft. UI Automation control patterns overview. Microsoft Learn, Win32 documentation, 2025a. URL https://learn.microsoft.com/en-us/windows/win32/winauto/uiauto-controlpatternsoverview. Document date July 14, 2025; accessed September 19, 2026.

Microsoft. Working with virtualized items. Microsoft Learn, Win32 documentation, 2025b. URL https://learn.microsoft.com/en-us/windows/win32/winauto/uiauto-workingwithvirtualizeditems. Document date July 14, 2025; accessed September 21, 2026.

Microsoft. Playwright MCP. Software and documentation, 2025c. URL https://github.com/microsoft/playwright-mcp/blob/850520321bbc2a2b6e814dde43f4980a452ad8b1/README.md. Snapshot dated December 11, 2025; accessed September 15, 2026.

Microsoft. Assertions. Playwright documentation, 2026. URL https://playwright.dev/docs/test-assertions. Accessed September 21, 2026.

Model Context Protocol. Architecture. Model Context Protocol specification, 2025. URL https://modelcontextprotocol.io/specification/2025-06-18/architecture. Specification version 2025-06-18; accessed September 15, 2026.

OpenAI. GPT-6 Astra: A new generation of intelligence, 2026. URL https://openai.com/zh-Hans-CN/index/gpt-6-astra/. Accessed September 22, 2026.

OSWorld Contributors. OSWorld reference PromptAgent and Set-of-Mark action parser. Source code, xlang-ai/OSWorld repository, 2024. URL https://github.com/xlang-ai/OSWorld/blob/f8e604082f0bc35d37f887a99b69ee1ca6fa9fde/mm_agents/agent.py. Snapshot dated May 30, 2024; descriptive source-code title; accessed September 15, 2026.

OSWorld Team. OSWorld 2.0: Official leaderboard and performance–efficiency data, 2026. URL https://osworld-v2.xlang.ai/. Leaderboard and interactive-chart data retrieved September 22, 2026; releases 2026.08.08 and 2026.06.24.

Linxin Song, Yutong Dai, Viraj Prabhu, Jieyu Zhang, Taiwei Shi, Li Li, Junnan Li, Silvio Savarese, Zeyuan Chen, Jieyu Zhao, Ran Xu, and Caiming Xiong. CoAct-1: Computer-using multi-agent system with coding actions. arXiv preprint arXiv:2508.03923, 2026. URL https://arxiv.org/html/2508.03923v3. Version 3, February 20, 2026; first submitted August 5, 2025.

Yiyou Sun, Xinyang Han, Weichen Zhang, et al. Agents' last exam. arXiv preprint arXiv:2606.05405, 2026. URL https://arxiv.org/abs/2606.05405v2. Version 2, June 11, 2026; first submitted June 3, 2026.

Rich Sutton. The bitter lesson. Author's essay, March 2019. URL http://www.incompleteideas.net/IncIdeas/BitterLesson.html. March 13, 2019; accessed September 15, 2026.

The Qt Company. Accessibility. Qt 6 documentation, 2026. URL https://doc.qt.io/qt-6/accessible.html. Accessed September 19, 2026.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. arXiv preprint arXiv:2402.01030, 2024. URL https://arxiv.org/abs/2402.01030v4. Version 4, June 7, 2024.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. arXiv preprint arXiv:2404.07972, 2024. URL https://arxiv.org/abs/2404.07972v2. Version 2, May 30, 2024.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652. Curran Associates, Inc., 2024. doi: 10.52202/079017-1601. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

Yan Yang, Xiangru Jian, Ziyang Luo, Zirui Zhao, Yutong Dai, Ziji Shi, Hanshu Yan, Jun Hao Liew, Silvio Savarese, and Junnan Li. StateAct: Program state, before pixels, for long-horizon computer-use agents. arXiv preprint arXiv:2607.22798, 2026. URL https://arxiv.org/html/2607.22798v1. Version 1, July 24, 2026.

Mengqi Yuan, Zilong Zhou, Xinzhuang Xiong, Weiming Wu, Jiayang Sun, Jiamin Song, Kaiqian Cui, Bowen Wang, Haoyuan Wu, Yitong Li, Dunjie Lu, Haikong Lu, Qi Zhen, Xinyuan Wang, Jiaqi Deng, Yuhao Yang, Cheng Chen, Boyuan Zheng, Alex Su, Xiao Yu, Hao Zou, Saaket Agashe, Xing Han Lu, Manpreet Kaur, Zhengyang Qi, Vincent Sunn Chen, Frederic Sala, Dayiheng Liu, Junyang Lin, Zhou Yu, Yu Su, Siva Reddy, Xin Eric Wang, Peng Qi, Tianbao Xie, and Tao Yu. OSWorld 2.0: Benchmarking computer use agents on long-horizon real-world tasks. arXiv preprint arXiv:2606.29537, 2026. URL https://arxiv.org/abs/2606.29537v2. Version 2, July 13, 2026; first submitted June 28, 2026.

Chaoyun Zhang, He Huang, Chiming Ni, Jian Mu, Si Qin, Shilin He, Lu Wang, Fangkai Yang, Pu Zhao, Chao Du, Liqun Li, Yu Kang, Zhao Jiang, Suzhen Zheng, Rujia Wang, Jiaxu Qian, Minghua Ma, Jian-Guang Lou, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. UFO2: The desktop AgentOS. arXiv preprint arXiv:2504.14603, 2025. URL https://arxiv.org/abs/2504.14603v2. Version 2, April 25, 2025.

Appendix A details HCU's design and implementation. Appendix B defines the experimental setup and analysis protocol. Appendix C expands the OSWorld 2.0 main results, Appendix D analyzes the interface components, and Appendix E examines agent behavior across models. Appendix F reproduces the tool definitions, current skill, and recorded feedback messages.

A HCU Design and Implementation

This appendix follows an interaction from the public interface through observation, execution, and state management to its platform implementation. Recorded examples illustrate these mechanisms; Appendix B.6 explains instruction integration.

A.1 Interface and System Boundary

Three callable channels, one task context. HCU makes programmatic, semantic, and visual interaction available within an agent's existing tool-calling process. Programmatic interaction uses the agent's code, shell, and file tools. Semantic and visual interaction are exposed through the independently callable `AX` and `GUI` desktop tools. AX offers both observation and reference-addressed actions; GUI offers both screenshots and mouse-and-keyboard input. A screenshot can inform a subsequent AX action, and an AX response can inform a subsequent programmatic operation; the model decides which combinations to use.

Figure 2 gives an overview of the channels. Figure 6 shows the service boundary, Appendix A.4 explains cross-channel state, and Appendix A.5 describes agent integration.

Table 4 summarizes the desktop operations; tool and parameter names are literal. Programmatic tools remain part of the agent, while HCU supplies the AX and GUI operations and their shared state.

Table 4: Desktop operations and selected optional fields.

Operation	Main request fields	Returned evidence and use
AX `observe`	`observation_target`; application or window identity; `scope`, `root`	Discover applications or inspect a selected surface, with state and references for later calls.
AX `find`	Application or window identity, `query`, `scope`; or a returned `cursor`	Retrieve matching acquired targets and local context without reading an entire window view.
AX `act`	Ordered `actions`; optional action/batch `postcondition`; `observe`	Execute reference-addressed actions, report execution evidence, and provide selected post-action observation.
AX `assert_ax`	`assertion`; permitted identity and `scope`	Check a specified AX-visible predicate. A false predicate is a result, not a tool error.
GUI screenshot	`actions` containing a single `screenshot`	Return desktop pixels and establish the layout used to check later coordinate input.
GUI input	Ordered `actions`: pointer, drag, scroll, text, or keys	Report input execution and visual feedback; coordinate actions are checked against the screenshot-bound layout.

Session ownership and runtime boundary. One `computer-use mcp` process owns one desktop session (Figure 6). The Go service publishes schemas, validates calls, and retains AX snapshots, reference registries, find cursors, eligible diff snapshots, and the GUI screenshot-layout binding. Platform providers implement acquisition and native input behind a typed private protocol; the shared service manages the public contract and session state. The agent retains model inference, conversation history, and its programmatic tools. The evaluation runner prepares isolated task desktops and evaluates their outcomes; it is not part of HCU's public interaction interface. Table 6 maps service responsibilities to source files.

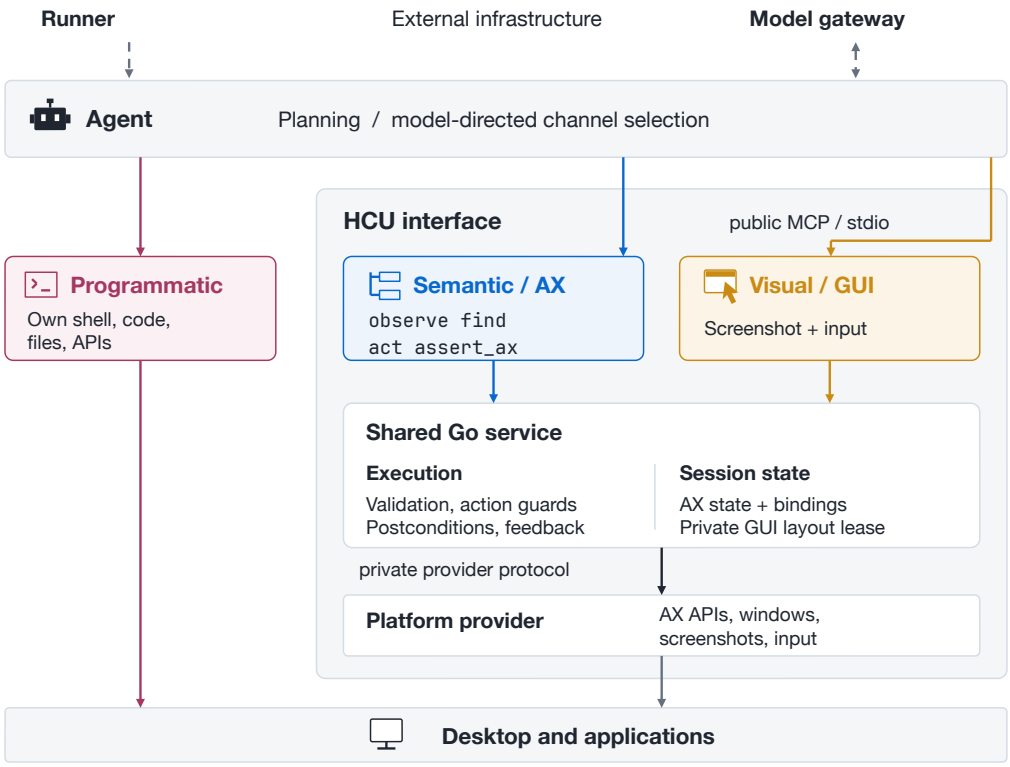


Figure 6: HCU service boundaries. AX and GUI share a Go service; programmatic tools and the evaluation runner remain external. Platform backends: Figure 12.

A.2 AX Observation and Target Identity

Compact observations with addressable targets. Accessibility exposes an application's selected semantic representation, not a transcript of the screen. Its provider may omit, duplicate, or misrepresent controls, and large trees may contain repeated labels and containers that add little to the next decision. HCU therefore separates the acquired structured representation from the compact AX view derived from it. The internal typed tree, called an AX AST here, retains the acquired hierarchy, properties, and identity evidence. The interface binds short target references before deriving the view shown to the model.

This separation serves two related purposes. Projection can remove redundant structure without making the displayed line number or tree position the identity of an element. The retained records can also support targeted lookup and structured comparison after an action. The model sees labels, roles, relevant values and states, references, and explicit omissions where applicable. If that view is insufficient, `find` and scoped observations retrieve additional evidence. The retained records hold what the provider exposed. Appendix A.2 compares unpruned and compact AX views from the same recorded captures.

References and observation scope. The service maintains references for applications (`aN`), logical top-level windows (`wN`), and elements (`rN`). Read requests use one permitted identity source, such as an application specification, `app_ref`, or `window_ref`. Element actions consume returned references; they do not accept invented IDs or arbitrary name selectors in place of a reference. A targeted action derives its application and window from the reference, rather than accepting a competing request-level identity. A targetless keyboard batch instead requires `window_ref`.

The default window scope includes application chrome. Content scope selects a uniquely identifiable content surface, while rooted observations narrow a read to a known subtree or focus context. A reference stays valid while the service can validate its ownership and current target; replacement, restart, or a new task session ends it. Platform identifiers also have lifetime limits: UI Automation runtime IDs, for example, may be reused over time (Microsoft, 2024).

Acquisition, binding, and projection. The provider collects bounded records containing roles, names, values, states, supported actions, geometry, relations, and identity evidence. The service establishes application and window ownership, builds the typed tree, and binds eligible targets. It then derives the compact AX view by projection. Provider filtering, traversal limits, collapsed menus, and virtualized content determine what the records contain before projection begins.

Projection first removes empty or redundant structural wrappers and normalizes text. It merges adjacent unreferenced text leaves and removes duplicate labels only when supported by provider relations; equal strings alone are insufficient. It preserves context needed to distinguish same-name controls, summarizes repetitive collections, and applies a structural budget with omission markers. This ordering separates semantic simplification from length control. A budget may omit an addressable subtree from the displayed view without deleting its acquired records or reference binding. A new `find` query acquires current application state and searches its retained records rather than the rendered text. Cursor continuation pages through that query's saved matches and snapshot without acquiring new state.

Recorded observation examples. These examples supplement Section 3.1. We select three states from 31 diagnostic captures across 12 applications, prepared in isolated Linux environments on September 20, 2026. Chrome illustrates container pruning, Audacity label deduplication, and Obsidian representation design. The states use ordinary demonstration files and were captured without running a model or task evaluator.

A.2.1 Same-Capture AX Views

The released HCU binary delivers a compact AX view from one provider capture. An offline renderer derives an unpruned AX view from the same records and reference bindings; this is a rendering of HCU's acquired records, not the original provider AX tree. Replay reproduces the delivered compact AX view. This isolates projection from collector and serialization differences.

Table 5: AX-view token counts (`o200k_base`, `tiktoken 0.13.0`). Reduction is 1 − Compact/Unpruned; "All text" adds headers, not images. Refs: retained/issued references. No output-budget truncation is applied.

Application state	Unpruned	Compact	All text	Reduction	Refs
Chrome permissions	1,817	1,016	1,043	44.1%	53/53
Audacity compressor	439	377	426	14.1%	11/11
Obsidian research note	1,374	1,201	1,237	12.6%	48/48

Figure 7 illustrates local transformations; Table 5 measures all passes. Projection keeps every reference but can drop other content: Chrome loses some named unaddressable controls. Audacity's provider values can differ from the displayed units.

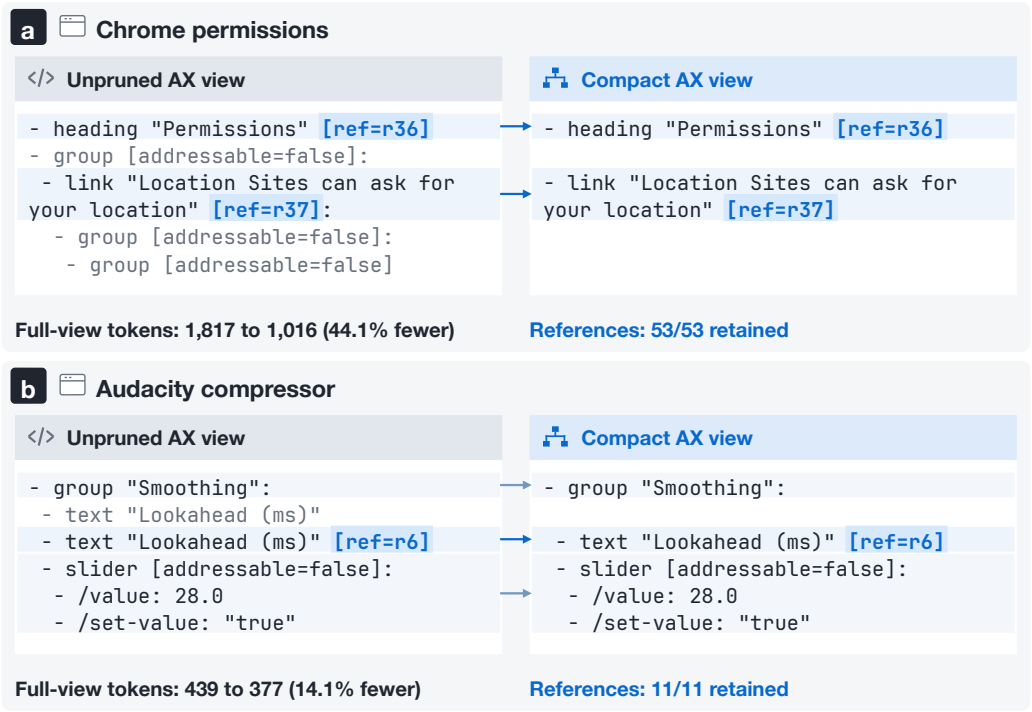


Figure 7: Same-capture AX excerpts. Counts cover complete view bodies, excluding headers and images (Table 5). Projection retains all issued references but drops some unaddressable labels; provider values may differ from displayed units.

A.2.2 Stock XML and the Compact AX View

The Obsidian example adds a separate comparison with the stock `/accessibility` XML returned in the official OSWorld 2.0 software image. An adjacent read, with no recorded intervening input, locates the same titled window. Its original-byte window fragment contains 30,526 tokens, whereas HCU returns 1,237 text tokens, including headers (Figure 8). The fragment excludes namespace declarations inherited from the desktop root. The two collectors ran at different times.

The XML-to-HCU difference therefore reflects the whole representation; within HCU's own capture, projection alone reduces the text by 12.6%.

Measurement and provenance. The capture calls request `observe=full`, which returns text and a window image; the token counts cover the text.

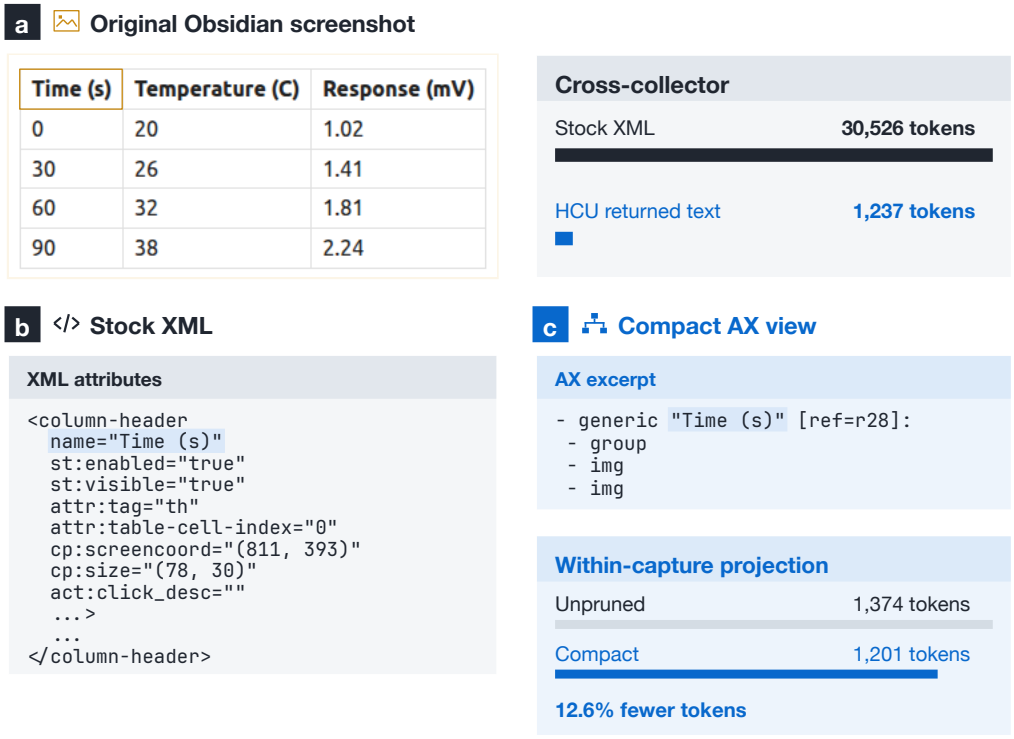


Figure 8: Obsidian screenshot, stock OSWorld 2.0 XML, and compact AX excerpts. Adjacent captures differ in collector and serialization; only the within-HCU comparison isolates projection. Note contents are demonstration data.

A.3 Action Execution and Post-Action Feedback

Post-action feedback supports direct continuation when it contains the next target and targeted lookup when more information is needed. In a Claude Opus 5 interaction, opening File returns the Reload reference `r111`, which the next request uses without another read. In a GPT-5.6 Sol interaction, a five-action batch requests three postconditions on checkbox states; the response reports them as passed and returns focused context.

Figure 9 shows the complementary use of targeted lookup. The search-field reference supports three known input actions, whose return exposes the entered value. A subsequent `find` supplies the result reference used to open the appointment page.

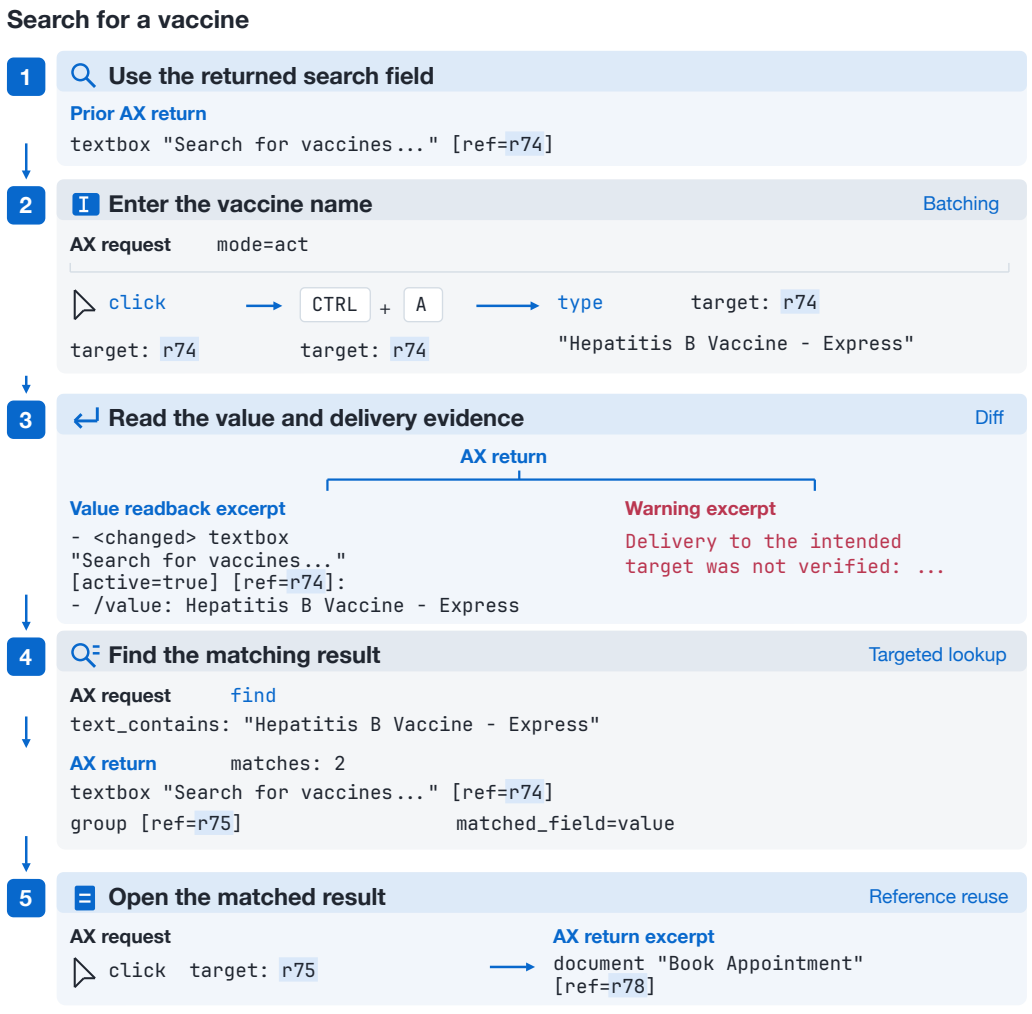


Figure 9: Recorded AX interaction with GPT-5.6 Sol, diagnostic OSWorld 2.0 run (Appendix B.5). AX records are not screenshots; stage numbers do not count calls or Steps. The ellipsis omits the delivery warning's input-queuing explanation. No postcondition was requested; `find` supplies the next target. The final view establishes navigation, not booking completion.

Batched actions and local postconditions. The model can submit known steps together as an ordered AX action list within one application and logical window. It must split the interaction when later actions depend on targets or decisions not yet available. Per-action postconditions can check an expected local state before the next step, and a batch-level postcondition can check the final requested predicate before returning the selected observation. Examples include an exposed

checked state or field value. Postconditions cover AX-visible state; a saved file, a completed remote operation, or task success needs other evidence.

Both AX and GUI can batch actions. HCU's semantic batch combines reference-based addressing with live resolution, requested postconditions, and post-action feedback. It is sequential, not atomic: stopping a batch does not roll back earlier actions. The response reports the stopping point and execution evidence, so the model can distinguish unattempted actions from actions whose effects need inspection before continuing.

The following schematic AX arguments illustrate a reference-addressed write with a local postcondition. They assume that a preceding observation returned `r1` for an editable field.

```
{
  "mode": "act",
  "actions": [{
    "type": "set_value",
    "target": {"ref": "r1"},
    "value": "example",
    "postcondition": {
      "type": "value", "target": {"ref": "r1"},
      "value": "example", "match": "equals"
    }
  }],
  "observe": "diff"
}
```

Feedback for continuation. AX action requests accept `observe=none`, `diff`, or `screenshot`, with `diff` as the default. Read-mode observations have a different set of permitted policies, including `tree` and `full`, which `act` does not accept. Under the `diff` default, the response contains changed subtrees, a bounded compact AX view, compact context, or an explicit observation-unavailable result. The next request can use newly returned references.

Execution evidence is independent of the observation policy: with `none`, requested postconditions are still checked. HCU reports known non-delivery and unknown effects separately, so the model can inspect state, retry an unexecuted operation, or use another channel when semantic evidence is insufficient.

Sequential execution and evidence. The service validates the batch before dispatch; targeted actions resolve their references against live state. The backend may invoke a native accessibility operation or synthesize input for the target element. Action errors, failed specified postconditions, or loss of logical-window continuity stop the batch; completed operations are not rolled back. Internal refreshes update execution state without returning to the model. On macOS, eligible targetless keypress sequences without postconditions execute within the provider, checking window continuity before each step and acquiring an AX snapshot at the sequence boundary rather than between keys. A non-error uncertainty flag alone does not stop the batch.

The response distinguishes whether input is known not to have been delivered (`not_executed`) from an effect that cannot be established after possible input (`effect_unknown`). Either flag can appear in an error or a non-error response. Assertions report the specified AX predicate, with `assertion_passed=false` when it does not hold. HCU does not count invocation success as an effect, because accessibility APIs may acknowledge invocation before it is serviced (Microsoft, 2021). Negative predicates and reports that a target disappeared likewise depend on observation coverage.

AX diffs and diff-snapshot eligibility. Post-action diffing compares structured snapshots, not strings produced by the compact renderer. The comparison uses an eligible previously delivered snapshot for the matching logical window, view, scope, and root. An internal refresh that the model never saw does not become the diff snapshot. Changes are then rendered with relevant state, target references, and bounded lifecycle context.

When no eligible snapshot exists or coverage cannot support a reliable delta, the interface returns current context, a bounded tree, or explicit unavailability. Only a complete, eligible snapshot that reached the model becomes the next diff snapshot, so display compression and internal sampling cannot change what counts as observed by the model.

A.4 Cross-Channel State and Failure Handling

Live target resolution and channel switching. Before an AX action, the service resolves the target against current provider state and checks identity and action-specific requirements, refreshing geometry as needed. A missing, ambiguous, or superseded target is rejected rather than treated as a similarly named replacement. Provider-level resolution may use constrained identity and semantic evidence, but an expired reference is never rebound to a new element.

GUI input that may have changed state invalidates the AX working snapshot, hidden reference cache, and find cursors. Registered references and the eligible diff snapshot remain, so the interface discards the stale working snapshot while keeping identity and previously delivered evidence for later checks.

GUI coordinate input uses a different check. The Go service retains the screenshot-layout binding, and the platform provider compares it with the live desktop bounds, visible top-level windows, stacking, and platform-specific foreground-window evidence before coordinate input. On macOS, that evidence is the frontmost on-screen layer-zero window, rather than an independently queried active window. A mismatch blocks the affected input and returns a fresh screenshot. Because the check inspects the current layout rather than the cause of a change, it also covers window changes made by shell commands, which trigger no HCU invalidation. The check does not cover rearrangement within a window or control focus, and GUI text and key actions follow current focus.

When recovery must stop. In the refactored implementation, an uncertain focus or pointer preparation failure can require stopping desktop automation, even when the requested input was not emitted. A queued operating-system request may still change focus or layout later. The batch stops without a recovery observation, a postcondition, or later actions when preparation may still be pending. Such an explicit stop response is not permission to refresh and replay: a new screenshot or a restarted MCP process does not establish that the pending request has settled. This differs from a stale-reference rejection, for which a new observation may supply a valid target.

Recorded checks and continuation. In an OSWorld 1.0 Writer task with HCU, a GUI screenshot precedes an AX action that changes the window context. The next coordinate batch is rejected with `STALE GUI FRAME`: no actions are dispatched, no input is emitted, and a fresh screenshot returns. The retry with adjusted coordinates dispatches seven actions and reports `effect_unknown=true` (Figure 11).

A.5 Platform Backends and Agent Integration

Shared semantics and native mechanisms. HCU provides native backends for Linux, macOS, and Windows behind the same public AX and GUI tools (Figure 12). The shared Go service owns AX representation, target references, batch orchestration, postconditions, and post-action feedback; it coordinates action guards with platform providers, which perform native checks, acquisition, and input. Linux uses a persistent embedded Python provider for AT-SPI, X11 window management, screenshots, and input. The macOS provider uses Swift with AXUIElement, CoreGraphics, and ScreenCaptureKit; the Windows provider uses C# with UIA3 and Win32. Each provider is private to its MCP process. The binary selects its backend at build time, without adding a platform-selection parameter to the public tools.

The native backends have been tested and validated on real desktop tasks. This functional validation is separate from the benchmark experiments reported in this paper, which all use Linux desktops; we do not report cross-platform benchmark comparisons. macOS requires a graphical login and Accessibility and Screen Recording consent. Windows requires an unlocked interactive session and compatible target integrity level, and does not bypass UAC or secure desktops.

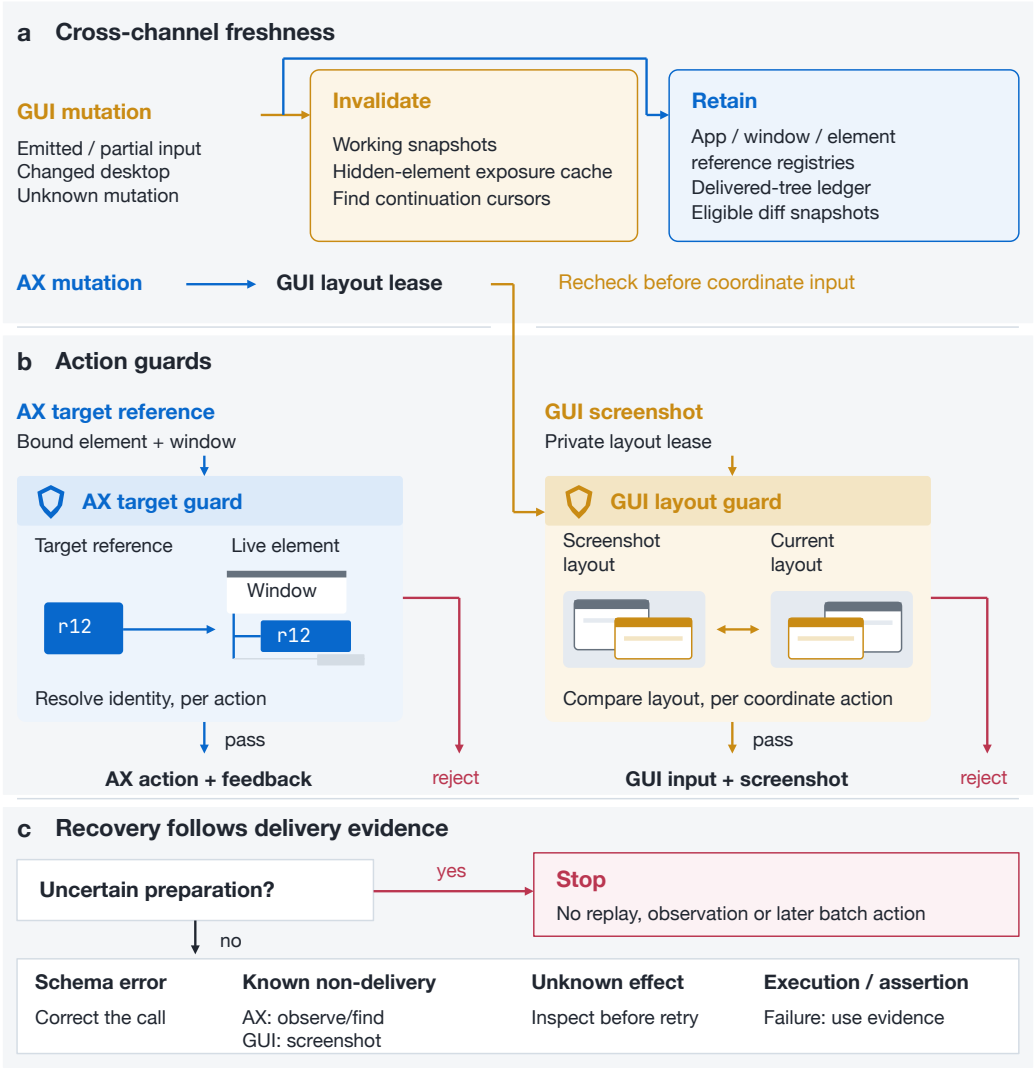


Figure 10: Cross-channel state and action guards (schematic). GUI mutation invalidates working observations, retaining identities and eligible diff snapshots. The GUI guard compares layouts, not pixels. Guards precede relevant inputs; earlier batch actions may have run. Uncertain preparation requires stopping.

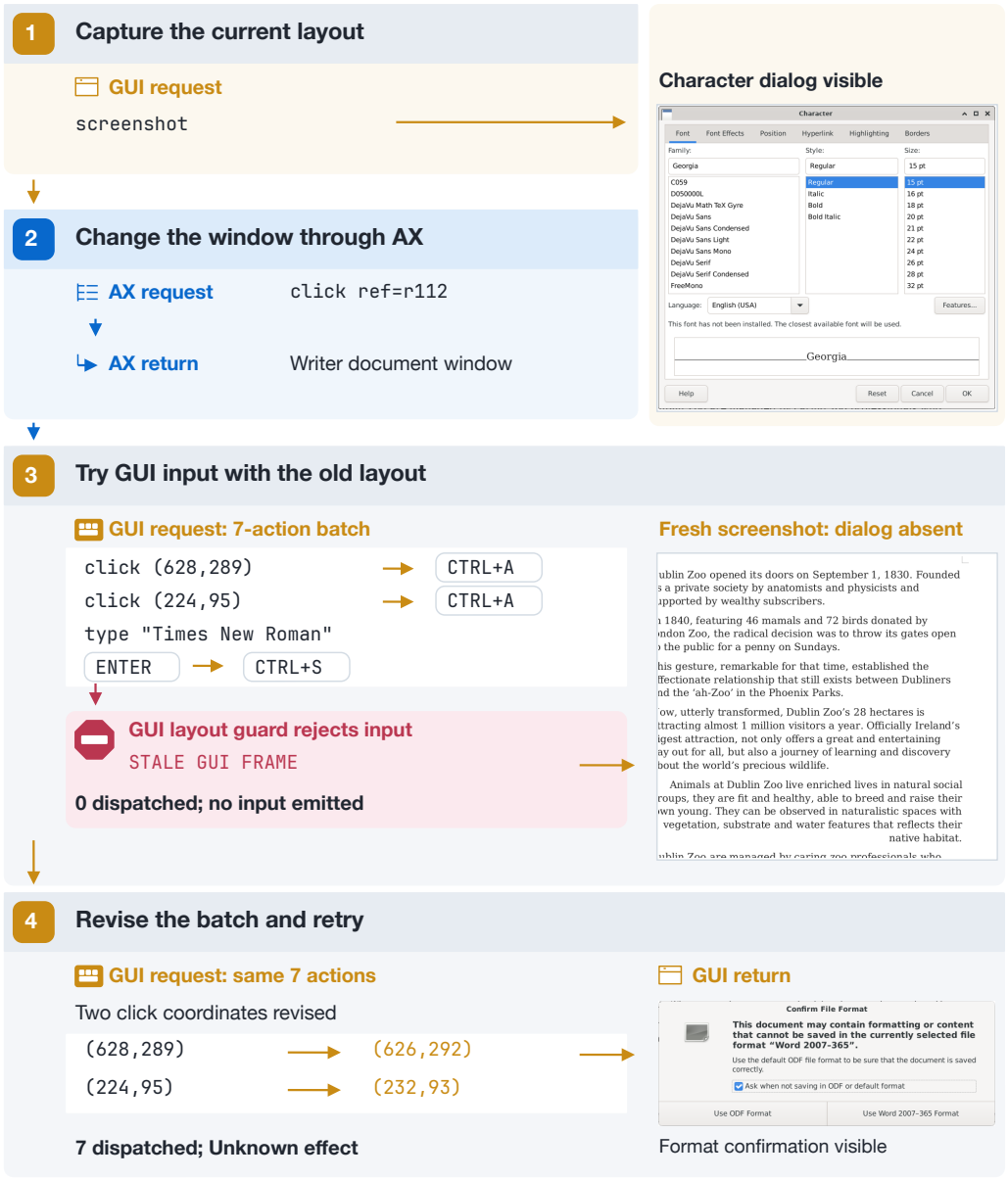


Figure 11: Recorded Writer layout-guard interaction, HCU (GPT-5.4), OSWorld 1.0. The GUI batch is rejected before input and returns a fresh screenshot; the first two crops share region and scale. Retrying with revised pixel coordinates dispatches seven actions with unknown effect. The dialog does not verify saving.

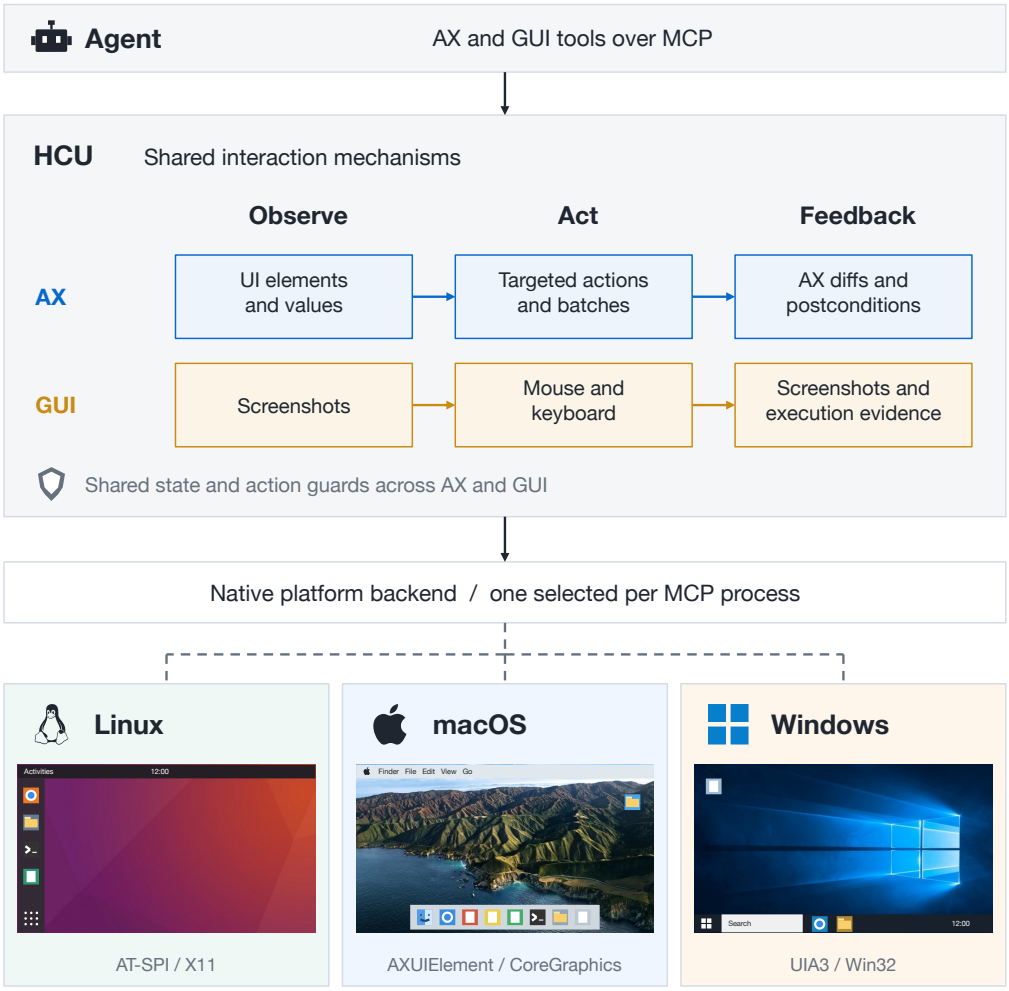


Figure 12: Shared HCU interface and native backends. Rows show capabilities, not a fixed workflow; AX can also return screenshots, and its postconditions are optional. Dashed branches are alternative backends, one per MCP process. Desktop illustrations (Ubuntu for Linux) are not recorded task screenshots or tested-version claims; the agent's programmatic tools remain external.

Table 6: Source files by responsibility. Paths are relative to `computer-use/`.

Responsibility	Source files
Public contract and transport	`tool_contract.go`, `mcp_transport.go`, `main.go`
AX representation and projection	`ax_snapshot_v1.go`, `ax_renderer.go`, `ax_distillation.go`, `ax_structural_budget.go`
References and targeted lookup	`ax_ref_registry.go`, `ax_find.go`, `native/linux/provider/element_resolver.py`
Batch execution and assertions	`ax_batch.go`, `ax_batch_plan.go`, `ax_batch_ordered_input.go`, `ax_assert.go`
Feedback and recovery	`ax_snapshot_store.go`, `ax_semantic_diff.go`, `ax_action_result.go`, `ax_failure_recovery.go`
Visual interaction and provider protocol	`gui_service.go`, `provider_wire_v2.go`
Native checks, acquisition, and input	`native/linux/provider/`, `native/macos/Sources/ComputerUseProviderKit/`, `native/windows/ComputerUse.Windows.Runtime/`

Attaching the interface to an agent. HCU is released as the stdio MCP server `computer-use mcp`. The agent registers that process and makes its AX and GUI schemas available alongside its own tools. The server runs as the signed-in desktop user with the appropriate display, authorization, runtime-directory, and session-bus environment. In the Linux deployment, these include `DISPLAY`, `XAUTHORITY`, `XDG_RUNTIME_DIR`, and `DBUS_SESSION_BUS_ADDRESS`. Diagnostics go to stderr so they do not corrupt the stdio protocol.

The refactored package provides standalone configuration generation and evaluation integrations for Codex, Claude Code, Gemini CLI, and Grok Build. Codex registers the server directly; the documented Claude Code version uses a schema adapter in front of the same server. The adapter translates schemas only and adds no task planner.

Each harness's existing context mechanisms receive the base policy and selected skill, while the public desktop tools remain AX and GUI. The evaluation runner supplies its completion protocol in the task prompt, separately from these product instructions; standalone configuration generation does not add it. Appendix B.6 details the injection paths.

B Experimental Setup and Analysis Protocol

This appendix specifies the controlled interface comparisons in Section 4: task sets, shared agent settings, interface configurations, and metric aggregation. Appendices C–E present the corresponding results and behavioral analyses.

B.1 Benchmarks and Task Sets

Tasks. The main experiment uses the OSWorld 2.0 full set (108 tasks; 0808). The OSWorld 1.0 task set covers 361 tasks, excluding eight Google Drive tasks but retaining external-web tasks; it is not the official offline subset.

By application, the OSWorld 1.0 tasks comprise 93 multi-application tasks, 47 each in LibreOffice Calc and Impress, 46 in Chrome, 26 in GIMP, 24 operating-system tasks, 23 each in VS Code and LibreOffice Writer, 17 in VLC, and 15 in Thunderbird.

B.2 Models, Harnesses, and Configurations

Models and harnesses. GPT-6 Astra and GPT-5.4 run in Codex; Claude Opus 5 and Claude Opus 5.5 run in Claude Code; Gemini 3.8 Flash runs in Gemini CLI; and Grok 4.7 runs in Grok Build. The main experiment uses GPT-6 Astra and Claude Opus 5, each at its highest recorded reasoning effort, with the same effort for HCU and CUA Driver with GPT-6 Astra. The OSWorld 1.0 comparisons use GPT-5.4 (xhigh), Claude Opus 5.5 (low), Gemini 3.8 Flash (high), and Grok 4.7 (xhigh). We report reasoning effort for GPT, Claude, and Grok and thinking level for Gemini; identical setting names do not establish equal compute budgets across providers.

Tool configuration. To focus the comparison on the computer-use interface, we restrict auxiliary harness features and keep the harness execution flow, programmatic tools, and shared base instructions consistent between interfaces within each model. These controls reduce variation from other harness components, making differences in task performance, resource use, and interaction behavior easier to attribute to the interface configuration.

The recorded restrictions differ by harness. For the first runs of the OSWorld 1.0 cross-model comparison, archived Gemini CLI settings exclude `google_web_search`, `web_fetch`, `save_memory`, `write_todos`, `ask_user`, `enter_plan_mode`, `exit_plan_mode`, `invoke_agent`, `activate_skill`, and `update_topic`. Native skills, agents, automatic memory, task tracking, and plan mode are also disabled. Grok Build disables user-question, backend-tool, image-generation, video-generation, voice-mode, remote-fetch, and session-search features, together with memory, subagents, and workflows. Claude Code initialization records expose `Bash`, `Edit`, `Glob`, `Grep`, `Read`, `WebFetch`, and `Write` as built-in tools, alongside each configuration's computer-use MCP tools; its native skill, plugin, and slash-command lists are empty. For each of these three harnesses, the examined settings are present and consistent across all 361 tasks in the first run of each interface. Disabling client-native skills does not remove the HCU skill injected as instruction text.

In Codex, the runner replaces the default base instructions with a general computer-use instruction shared by all five ablation configurations. Appendix F.2 presents the base policy and supplementary skill.

Comparison configurations. The main experiment includes HCU and CUA Driver with GPT-6 Astra and HCU (Claude Opus 5), with reported results as context. The ablations compare HCU, HCU w/o GUI, HCU w/o AX, HCU w/o skill, and CUA Driver. All retain Bash. Concurrency settings are described below.

Table 7 defines the exposed desktop-tool entry points. All configurations retain programmatic tools, including shell access to browser, accessibility, or input automation, so the ablations measure the value of providing dedicated AX and GUI tools within a general agent.

CUA Driver keeps the same programmatic tools, and foreground input is available to it. In the OSWorld 1.0 ablations, CUA Driver ran as the unmodified cua-driver 0.22.0 with all tools exposed;

Table 7: Direct tool access in the channel ablations. A check mark means that the configuration exposes that kind of tool directly.

Configuration	Direct code/CLI	AX	GUI
Programmatic only	✓	–	–
HCU w/o GUI	✓	✓	–
HCU w/o AX	✓	–	✓
HCU	✓	✓	✓

the model could request foreground delivery on each call and did so 1,742 times in the first round, against 148 explicit background requests, with no foreground request denied.

Tool exposure and skill. Configuration determines which desktop channels are exposed and which supplementary skill is supplied. The channel ablations retain the agent's programmatic tools while exposing AX, GUI, or both. Removing the supplementary skill leaves the schemas, tool descriptions, and server instructions available. The hybrid skill covers choosing evidence appropriate to the task, using returned references, batching known steps, interpreting postconditions and uncertainty, and recovering from stale state.

The skill explains tool use and provides recommendations; channel choice remains with the model. We evaluate the tools and skill together as the interface configuration. Codex receives the selected skill through a task-local `AGENTS.md`, while Claude Code receives it in its configured system prompt. Appendix B.6 describes instruction injection, and Appendix F.2 presents the instruction text.

Each cross-model pair shares the model snapshot, reasoning effort or thinking level, harness, programmatic tools, evaluator, initial environment, and budgets. Pairing tasks within each model–harness configuration measures the interface's effect under the same agent setup. Repeating this comparison across models and their respective harnesses tests whether the gains extend to different agent configurations.

B.3 Execution Environment and Budgets

Task isolation and execution settings. A fresh task receives a dedicated desktop and MCP process so references, cursors, snapshots, and screenshot bindings cannot leak from earlier tasks. Each release fixes the binary and skill, and each run records the model, harness, tool exposure, environment, and evaluation settings.

Runs use Linux desktops with foreground input available. The main experiment has one run per task and configuration under a common 500-step ceiling. The OSWorld 1.0 ablations use budgets of 100 successful model responses and 1,800 seconds per task. Table 2 uses three complete rounds of all five configurations, yielding 5,415 official numeric scores. Skill diagnostics cover their HCU traces with and without skill.

Each round supplies one official numeric score per task and configuration, including zero; the same record supplies both outcome and resources. All 5,415 scores were checked against their per-task result records.

In round 1, HCU w/o skill used concurrency twenty and the other configurations used ten. Round 1 also ran a model-based judge after each task; neither its score nor its usage enters these measures, and later rounds did not run it. Time measures agent execution rather than batch wall time. Table 10 reports per-round means and their sample standard deviations.

B.4 Metrics and Aggregation

For task scores $s_i \in [0, 1]$, OSWorld 1.0 Score and OSWorld 2.0 Partial denote the mean task score; OSWorld 2.0 Binary denotes the full-credit rate:

$$Q_{\text{mean}} = \frac{100}{N} \sum_{i=1}^{N} s_i, \qquad \text{Binary} = \frac{100}{N} \sum_{i=1}^{N} \mathbf{1}[s_i = 1]. \tag{1}$$

The OSWorld 1.0 ablation aggregates use the official numeric `evaluator_score`; runner `score`, model-judge scores, and tool success flags are not substituted. Every resource column in Table 2 has an observed value for all 361 tasks per arm in each of three rounds. OSWorld 2.0 scores use each source's own metric definitions. The accompanying implementation's queue summarizer additionally reports `effective_score`, assigning zero to budget- or timeout-truncated tasks while preserving `raw_score`. Its `aggregate.final_score` averages effective scores over the selected task set and remains unavailable until all required results are complete. These effective-score fields are not the evaluator-score aggregates reported here.

The OSWorld 1.0 analysis measures efficiency by successful model responses and cumulative input/output tokens per task, including failed-task usage. In Table 2, "Steps" denotes these successfully delivered responses, following the runner's `agent_steps` counter and the exporter field `successful_response_steps`. Failed transport attempts do not count as delivered model steps, whereas a delivered response can still produce an invalid tool call. The OSWorld 2.0 figures instead use source-reported output tokens and cost in dollars per task. A response may contain multiple tool calls, and a call may batch multiple actions, so we count these units separately. Cumulative input includes cached history.

We report these scores with Steps, Tool calls, and provider-reported input and output tokens, retaining the provider's cache-read, cache-write, uncached-input, and reasoning categories. A cache category is never added to a total that already contains it. All 5,415 ablation records contain observed Steps, input, output, Time, and Cost. For these records, Cost uses observed input at $2.50/M and output at $15/M, without cache discounts; output includes reasoning. Time is agent execution, not batch wall time. Cross-model Cost uses list prices, with cache discounts as reported for Gemini 3.8 Flash and Grok 4.7, cache-read and cache-write prices for Claude Opus 5.5, and none for GPT-5.4.

Resource totals include usage from failed tasks, and zero scores count as outcomes. In the cross-model comparison, configurations with repeated runs, three for each GPT-5.4 configuration and two for HCU (Claude Opus 5.5), report means over those runs: we first average each task's observations within each configuration, so repeated runs of a task count as one task rather than as independent tasks. The other configurations in that comparison ran once. For the complete ablation rounds, this equals averaging metrics over tasks within a round and then equally across rounds, or averaging 1,083 observations per configuration while retaining 361 distinct tasks. Resource changes use ratios of means, not means of task ratios. Additional repetitions for the GUI action-space replacement are specified in Appendix D.4.

Cross-model channel shares average the eligible task-level shares over runs before averaging tasks. The skill-ablation error rates instead pool observed calls across rounds; their task-run rates equal the average of round rates because each round has the same number of tasks. Appendix D.2 gives the denominators, and Appendix E.3 defines the additive Score decomposition.

B.5 Trace Data and Analysis Definitions

Sections 4.3 and 4.4 analyze the OSWorld 1.0 runs call by call. Comparisons between HCU and CUA Driver pair tasks within each model and average the runs of each arm (Appendix B.4). They use the standard task set, with the Claude Opus 5.5 exception specified in Appendix E.1. A tool call is one invocation recorded by the harness, including shell and file tools. One model response can issue several tool calls, and one AX or GUI call can request several actions, so these counts are kept apart. Each call is assigned to the channel of its tool entry point, so a shell command that imports accessibility APIs counts as Shell. The call after an error is the first new call issued after the error was returned; calls already issued in parallel are skipped. Post-error calls (Figure 19) and recorded termination categories (Figure 17) are extracted only from the first run of each arm, so these two figures show that run.

Sources of recorded examples. OSWorld 1.0 examples come from the channel and skill ablations, with GPT-5.4. OSWorld 2.0 examples come from diagnostic runs with HCU using GPT-5.6 Sol (xhigh) and Claude Opus 5 (max). These examples illustrate interface behavior and do not enter the main-result aggregates. Calls to the model-based judge are excluded.

Recorded examples. Table 8 summarizes the recorded examples by source and observed interaction. We chose the examples to show mechanisms.

Table 8: Sources and interactions of the recorded examples.

Source	Observed interaction
OSWorld 1.0 / GIMP	AX typing, focused read, screenshot, AX confirmation
OSWorld 1.0 / Writer	AX transition, stale GUI rejection, retry with adjusted coordinates
OSWorld 1.0 / Writer	Reload reference used after two shell edits
GPT-5.6 Sol	Code allocation followed by AX batches with postconditions
GPT-5.6 Sol	Batch search, returned value, `find`, then result click
Claude Opus 5	AX diff supplies the Reload reference for the next request
Claude Opus 5	Console context lacks the input reference; `find` supplies it

B.6 Instruction Integration

The instruction package separates a common base policy from selected interaction instructions under `instructions/`, and supplies the runner's completion protocol separately. Appendix F.2 reproduces the base policy and hybrid skill. The selected text is injected directly through existing context mechanisms, not discovered or activated as a client-native skill. Omitting the skill retains the base policy and does not itself remove a tool. Channel ablations also change the exposed tool set.

In the refactored runner, Codex receives the base through `model_instructions_file` and the selected text through the task's `AGENTS.md`; standalone configuration instead uses `developer_instructions` for the selected text. Claude Code receives a replacement system prompt with the selected text appended. Gemini CLI receives the base through `GEMINI_SYSTEM_MD` and selected text through an isolated global `GEMINI.md`; Grok Build receives both through its replacement system prompt. The runner selects the skill with `-instructions-file`; an empty value or `no_instructions: true` in batch configuration omits that text while retaining the base policy.

The OSWorld runner constructs the task prompt separately from the base policy and skill, appending the completion and user-question protocol to the task instruction. It requires `TASK_COMPLETE` or `TASK_INFEASIBLE` alone on the final nonempty response line, or `ASK_USER` followed by one concise question when user-only information is required. Additional clarification rules apply when a user-response channel is configured. Standalone configuration generation does not add this evaluation protocol.

C OSWORLD 2.0 RESULTS

We first detail our own runs and the matched CUA Driver comparison, then document the reported results used for context and the construction of the performance–efficiency frontier. The protocol and metric definitions are in Appendix B.

C.1 DETAILED HCU RESULTS

Table 1 reports Partial, Binary, and Cost for HCU with GPT-6 Astra and Claude Opus 5. The former's detailed counters appear with its matched CUA Driver comparison in Appendix C.2; the latter's resource breakdown follows here. Both runs use the main-experiment protocol, while their harness-specific interaction counters remain distinct.

HCU (Claude Opus 5). The run averages 75.8 outer agent turns, 88.0 model calls, and 81.6 tool calls per task. Mean input is 8,161,776 tokens per task: 45,192 uncached, 5,434,047 cache-read, and 2,682,538 cache-write tokens. Mean Time is 6,837 seconds (113.95 minutes) per task. The run follows the main-experiment protocol, the 0808 release with a 500-step ceiling.

C.2 MATCHED COMPARISON WITH CUA DRIVER

HCU and CUA Driver with GPT-6 Astra. Both configurations run GPT-6 Astra in Codex with Code Mode enabled on the full set, under the 500-step ceiling (Table 9). HCU's Partial is 3.55 percentage points higher (72.37% versus 68.82%), with 29.81% fewer tool calls and 9.57% fewer model calls. Mean Time decreases from 2,905 to 2,295 seconds (21.00%); median Time decreases from 2,208 to 1,977 seconds. The main-turn counter increases from 8.50 to 9.37. Input is about 6.9M tokens per task in both configurations; output is about 25.8k versus 25.4k. Cost is $10.05 per task for CUA Driver and $10.03 for HCU, effectively the same.

Table 9: HCU versus CUA Driver with GPT-6 Astra in Codex, OSWorld 2.0 full set (0808). Code Mode; 500-step limit; rounded token counts.

Metric	CUA Driver	HCU
Partial (%)	68.82	72.37
Binary (%)	31.5	32.4
Main turns	8.50	9.37
Model calls	94	85
Tool calls	111.7	78.4
Input tokens / task (M)	6.9	6.9
Output tokens / task (k)	25.8	25.4
Time (min), mean	48.42	38.25
Time (min), median	36.80	32.95
Cost ($)	10.05	10.03

Main turns and model calls are distinct counters in the source.

Resource accounting. Resource comparisons between HCU and CUA Driver with GPT-6 Astra use the two runs in Table 9. Each relative change is $(\text{HCU}/\text{CUA Driver} - 1) \times 100\%$ for that metric; the Partial difference is in percentage points. Main turns, model calls, and tool calls are distinct source counters. Elapsed seconds are divided by 60; input and output tokens are displayed in millions and thousands, respectively. Both input means round to 6.9M. Figure 1 uses Cost to compare efficiency across reported results, while interaction counters support the within-harness comparison in Table 9.

C.3 REPORTED RESULTS AND COMPARABILITY

Source and release selection. We use the leaderboard and interactive-chart data of the official OSWorld 2.0 website as of September 22, 2026 (OSWorld Team, 2026): 35 chart records and six additional leaderboard score–cost pairs. For the leaderboard-only records, we divide each

`estimatedCostUsd` by the site's `datasetSize` of 108, following its display function, and mark the per-task cost as estimated.

Within each scope and model, we select 0808 when available and otherwise 0624. Each plotted score remains attached to the resource value in its source record. The table uses the highest available reasoning effort. Figure 1 (right) shows Claude at its highest reported effort and keeps GPT-5.6 Sol's available effort settings, with names and scores beside the points, on a linear 60–76% score axis.

Additional official record. The official website lists a full-set GPT-5.6 Sol (max) record of 62.72%, 36,139 output tokens, and $13.94 per task in its table but leaves it out of its token and cost plots; we include this pair.

Vendor score-only reported results. Meta (2026a) reports 66.9% Partial and 32.0% Binary for Muse Spark 1.3 at max reasoning effort. Its methodology identifies the OSWorld 2.0 full set, release 0808, and a common internal GUI framework retaining the official task environments and grading (Meta, 2026b). The overall score-selection policy uses the highest comparable primary metric across internal evaluations, official leaderboards, and provider reports; a trial count and step ceiling specific to OSWorld 2.0 are not disclosed.

Google DeepMind (2026a) reports 59.0% Partial for Gemini 3.8 Flash, taking the maximum over three runs with one attempt per run (Google DeepMind, 2026b). Its setup uses the OSWorld 2.0 repository's Gemini CUA harness, official Docker and evaluator, 1080p screenshot-only observations, PyAutoGUI actuation, batched tool calls, and a 500-step ceiling. The runs precede the 0808 patch, and the documentation does not identify the exact earlier release, evaluated task count, or thinking level specific to OSWorld 2.0, so we mark the result as pre-0808.

Kimi Team (2026) reports 58.3% Partial on OSWorld 2.0 for Kimi K3 at max reasoning effort, with temperature 1.0 and agentic top-p 1.0. We classify the result under the 0624 release because the report's August 6 revision predates 0808. The report does not state the task count, step ceiling, harness, or trial count. Its separate OSWorld-Verified score is not imported.

None of these three vendors provides paired resource measurements, and Google and Moonshot AI do not report Binary; dashes denote unavailable values, and all three records are excluded from the efficiency frontiers.

StateAct author report. StateAct reports 61.6% Partial and 26.9% Binary on the OSWorld 2.0 full set using Claude Opus 4.8 with adaptive thinking (Yang et al., 2026). Sections 5.1–5.2 of the paper pair this result with approximately $7.8 and 100k output tokens per task, and report about 155 total model turns (57 main-agent and 98 subagent turns). The main-agent budget is 200 turns per invocation, with a 50-turn cap per subagent delegation. We use these reported results for the 0624 release without recomputing Cost. The report does not specify its pricing and cache-accounting details or whether finish-gate usage is included. Its separate OSWorld-Verified results are not imported.

Time coverage. The official website reports turns, actions, output tokens, and Cost, but no end-to-end task Time.

OpenAI's offline-subset report. OpenAI (2026) reports the 0808 offline Partial metric in its GPT-6 Astra announcement, with GPT-6 Astra at 72.6% and GPT-5.6 Sol at 65.7%, and gives no cost, token, or turn data for these scores.

Public table selection. Table 1 retains only the highest available reasoning effort and does not repeat alternative budgets or sources for a model in the same task scope. Where the chart and the leaderboard both list a model, we use the chart record, and we exclude 150- and 300-step leaderboard records. Our OSWorld 2.0 runs use a 500-step ceiling.

The table marks estimated and approximate resources.

C.4 PERFORMANCE–EFFICIENCY FRONTIER

Frontier construction. Figure 1 (right) places HCU on the performance–efficiency frontier of the displayed results by minimizing Cost and maximizing its paired Partial score. A point is dominated when another has no greater Cost and no lower Partial, with at least one strict inequality. The shaded region is the union of these dominated regions. StateAct and HCU (GPT-6 Astra) form this frontier. Inclusion follows the source and effort selection in Appendix C.3; score-only records cannot enter a score–Cost frontier. The figure compares reported operating points under their respective protocols, with estimated and approximate resources marked.

D INTERFACE COMPONENTS

This appendix supports Section 4.3 with per-round ablation results and the interaction patterns behind them: how the skill changes invocation errors, batching, postconditions, and feedback use. The final subsection examines GUI action-space replacement with two models. The ablations evaluate interface configurations; call-level records describe how their internal mechanisms are used.

D.1 CHANNEL AND SKILL ABLATIONS ACROSS ROUNDS

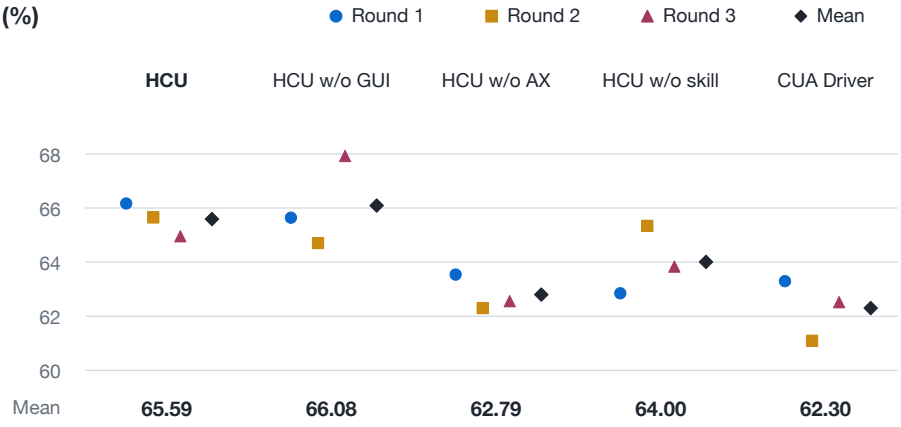
Table 10: Per-round Score (%); mean ± sample standard deviation across three rounds.

Configuration	Round 1	Round 2	Round 3	Mean ± SD
HCU	66.16	65.65	64.95	65.59 ± 0.61
w/o GUI channel	65.64	64.70	67.92	66.08 ± 1.66
w/o AX channel	63.53	62.30	62.55	62.79 ± 0.65
w/o skill	62.84	65.33	63.83	64.00 ± 1.25
CUA Driver	63.29	61.09	62.51	62.30 ± 1.12

In every round, HCU scores above both HCU w/o AX and CUA Driver, and HCU w/o GUI scores above CUA Driver (Table 10). HCU also needs the fewest Steps in every round: 28.4–28.8, against more than 31 for every other configuration.

The GUI channel consistently reduces Steps, while its Score contrast changes sign across rounds. HCU leads HCU w/o GUI by 0.52 and 0.95 points in the first two rounds and trails by 2.97 points in the third, leaving HCU w/o GUI with a 0.50-point higher mean Score (Table 10).

a Score (%)



b Task-level changes

Round pair	HCU	HCU w/o GUI	HCU w/o AX	HCU w/o skill	CUA Driver
1 vs 2	9.4% (34/361)	11.6% (42/361)	13.0% (47/361)	11.4% (41/361)	10.5% (38/361)
1 vs 3	11.9% (43/361)	10.0% (36/361)	11.1% (40/361)	12.7% (46/361)	10.8% (39/361)
2 vs 3	7.8% (28/361)	12.5% (45/361)	11.9% (43/361)	11.4% (41/361)	13.0% (47/361)

Figure 13: Ablation variation across rounds. (a) Round Scores and their mean. (b) Tasks with absolute Score changes above 0.5 between rounds, with counts and denominators.

Run-to-run variation. Repeated runs on the same 361 OSWorld 1.0 tasks show how much Score moves without any change in configuration. Across the three ablation rounds, a configuration's

round Scores span 1.21 points for HCU, 1.23 for HCU w/o AX, 2.20 for CUA Driver, 2.49 for HCU w/o skill, and 3.22 for HCU w/o GUI (Table 10). Between two rounds of the same configuration, 7.8–13.0% of tasks change by more than 0.5 points (Figure 13).

Against these spreads, HCU leads HCU w/o AX in every round, by 2.63, 3.35, and 2.40 points, and leads CUA Driver by 2.87, 4.56, and 2.44. The difference between HCU and HCU w/o GUI changes sign across rounds. HCU also leads HCU w/o skill in every round, by 3.32, 0.32, and 1.12 points, so the size of the skill's Score effect varies across rounds.

D.2 Skill, Invocation Errors, and Repeated Attempts

Table 11: Skill diagnostics pooled across three rounds per configuration (2,166 traces; 361 tasks). Cells report event or task-run counts.

Metric (numerator / denominator)	HCU w/o skill	HCU
Schema rejection / known-status calls	2460/14160	147/10783
Task-runs with schema rejection	601/1083	132/1083
Task-runs with consecutive schema failures	484/1083	4/1083
All Computer errors / known-status calls	2686/14160	638/10783
Stale-category errors / calls	107/14160	88/10783
Execution or assertion errors / calls	66/14160	380/10783
Internal request-construction errors / calls	50/14160	7/10783
Unclassified errors / calls	3/14160	16/10783
Multi-action AX / known-size AX requests	233/4241	495/3496
Task-runs using multi-action AX	117/1083	230/1083
Multi-action GUI / known-size GUI requests	793/4915	342/2379
Task-runs using AX `find`	312/1083	388/1083
Task-runs using `assert_ax`	2/1083	290/1083
Task-runs requesting postconditions	9/1083	173/1083

The paired HCU conditions retain AX, GUI, Bash, shared base instructions, and tool descriptions; the condition without skill disables the detailed hybrid skill, not all instructions. Configuration records bind both conditions to the same executable and tool contracts; the rounds differ in the runtime details described in Appendix B.3. The skill covers fields, target references, stable-action batching, postconditions, default diffs, `find`, and recovery. The analysis covers 2,166 traces. A task-run is one task in one round; each condition has 1,083 task-runs but only 361 distinct tasks. Call rates pool observed events across rounds rather than averaging three round percentages. Task-run rates equal the average of per-round rates because each round has 361 tasks.

Nineteen explicit errors have no assigned subtype (three without skill, sixteen with skill); they count toward total errors but not toward schema errors. Table 12 gives these rates per round; all three rounds show the same direction.

Invocation errors and repeated attempts. The classification uses explicit failure messages and request evidence, separating malformed arguments from stale identities, execution or assertion failures, internal request construction, and unknown causes. Schema errors fall from 2,460 to 147. The stale category contains 107 and 88 errors, respectively: 86 and 73 stale-target, stale-window, or layout-guard rejections, plus 21 and 15 aborted batches with a stale diagnosis and unknown effect. Execution/assertion errors change from 66 to 380. An additional 50 failures without skill and seven with skill report that desktop actions are required despite nonempty model action arrays; we classify these as internal request failures. Total AX and GUI call errors are 2,686/14,160 without skill and 638/10,783 with skill. Execution and assertion failures rise with the skill, alongside increased use of assertions and postconditions (below).

Consecutive schema-error chains contain at least two non-overlapping, consecutive calls to the same AX/GUI tool that receive schema rejections; intervening tool calls break a chain. There are 581

Table 12: Per-round skill-use rates (%). Definitions match Table 11.

Round	Metric	HCU w/o skill	HCU
First	Schema rejection	17.17	1.43
First	Consecutive errors	43.77	0.83
First	Multi-action AX	4.49	15.00
Second	Schema rejection	17.86	1.04
Second	Consecutive errors	45.71	0.00
Second	Multi-action AX	5.87	12.92
Third	Schema rejection	17.08	1.62
Third	Consecutive errors	44.60	0.28
Third	Multi-action AX	6.02	14.60

chains across 484 task-runs without skill, involving 1,811 errors, versus four chains across four task-runs with skill, involving eight errors. Adjacent arguments in these chains differ, so the model varies its parameters rather than resending the same request. The next call is the first new request initiated after the failed call returns; already-issued parallel calls are skipped. When that call is to the same tool, schema rejection repeats in 1,230/1,922 cases without skill versus 4/132 with skill.

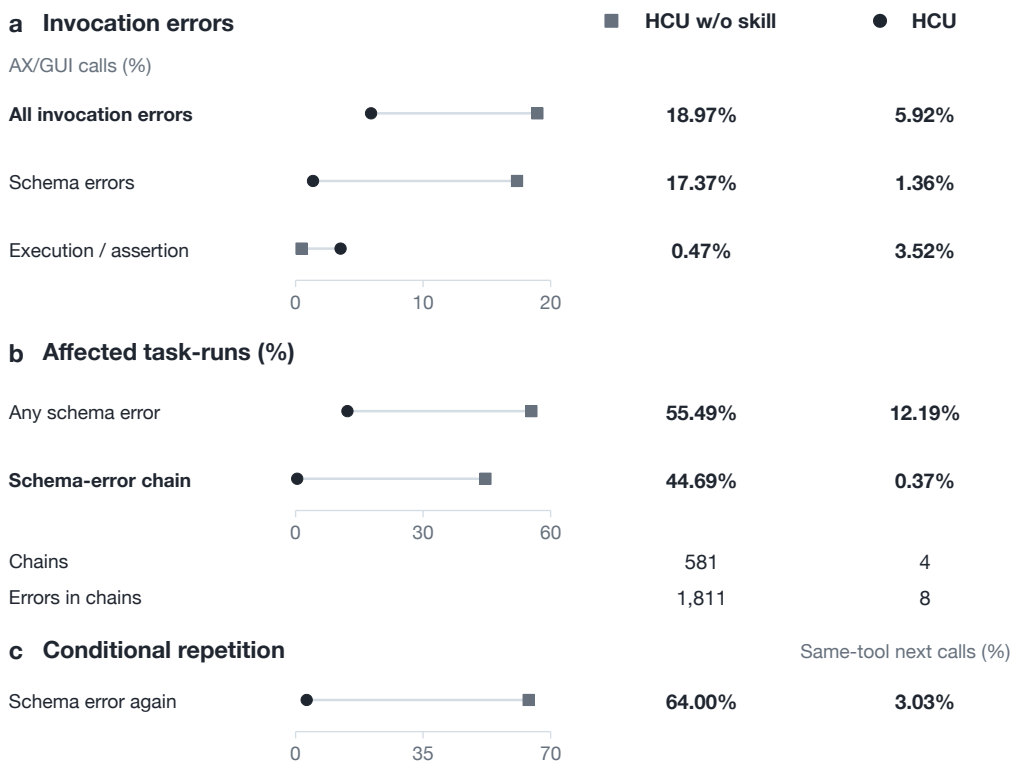


Figure 14: Skill-ablation errors. (a) Percentages of AX/GUI calls, pooled across rounds. (b) Percentages of task-runs. (c) Repeated schema rejection on the next same-tool call. Counts and denominators are reported in Table 11 and the text.

Repeated errors and Steps. The Steps saved by the skill concentrate on tasks prone to repeated invocation errors. Tasks on which the model without the skill repeated schema errors on the same tool in one round make up 45% of task-runs in the other two rounds, but 63% of the Steps reduction

and 67% of the reduction in GUI input actions there. Selecting tasks in one round and measuring them in the others links the skill's efficiency gains to recurring calling difficulties across runs.

Illustrative cases. Two cases, selected for their long error chains, show the chains described above. The first receives full credit in both conditions: its trace without skill contains a six-call chain trying different malformed AX arguments, whereas the trace with skill has one schema error and no chain. The second has a seven-call chain without skill and no schema error with skill, yet both receive zero credit. The cases illustrate fewer repeated rejections with the skill at both full-credit and zero-credit outcomes.

D.3 Batching, Postconditions, and Feedback Use

Batching, assertions, and postconditions. Multi-action AX requests increase from 233/4,241 to 495/3,496. GUI multi-action requests decrease from 793/4,915 to 342/2,379. These counts describe requested batches rather than task outcomes.

Targeted AX search appears in 312 versus 388 task-runs, assertions in two versus 290, and action postconditions in nine versus 173. Effective default-or-explicit diff requests increase from 81.25% to 96.02% of AX action requests. Immediate follow-up observations occur after 1,553/7,372 eligible action calls without skill and 1,258/4,562 with skill: their conditional share rises from 21.07% to 27.58% despite a lower count. Returned MCP image blocks decrease while AX text characters increase.

These counts compare HCU without and with skill, in that order. They document use of batching, assertions and postconditions, and feedback; they are not separate interventions on those mechanisms.

Feedback samples. The OSWorld 1.0 sample takes 12 of the 218 tasks in which HCU issued AX actions, using a fixed deterministic ordering of tasks, and at most the first two AX diffs of each, yielding 11 events from eight tasks. The GPT-5.6 Sol run contributes 24 events drawn using a fixed deterministic ordering of its 342 AX diffs (14 tasks), and the Claude Opus 5 run all 22 of its AX diffs (13 tasks). No selection filters by task outcome.

An AX diff requires both an `ax_diff:` payload and `final_observation_kind=diff`. Each event is annotated by the next execution or verification sequence. Direct new-target use requires checking earlier observations to establish that the reference was newly delivered. Independent reads include `find`, observations, and screenshots. Events whose ordering is unclear are labeled indeterminate.

Table 13 counts what followed each sampled AX diff.

Table 13: Behavior after sampled AX diffs. Sampling rules differ across columns; "Other" includes shell work and assertions.

Subsequent behavior	OSWorld 1.0 (11)	GPT-5.6 Sol (24)	Claude Opus 5 (22)
Direct use of a newly returned target	2	4	7
Other direct continuation	2	10	0
Independent read before execution	3	8	11
Indeterminate ordering	0	1	1
Other execution or verification	4	1	3

All 57 action requests omit `observe`; the returned diffs follow the default policy. These samples illustrate both direct continuation and targeted acquisition of additional evidence. In an OSWorld 1.0 Writer trace, a diff supplied the Reload reference that the model used after two shell edits, without another AX observation. In a GPT-5.6 Sol search, the diff echoed the entered query and `find` supplied the result reference used by the next click; in a Claude Opus 5 FreeCAD interaction, `find` supplied the console input reference. Among the eight GPT-5.6 Sol and eleven Claude Opus 5 independent-read events, only two and one include unrooted window observations; the other read types include target queries, scoped content, subtrees, and screenshots. Read types can overlap within an event.

Continuation depends on the evidence returned and the next operation. The samples also contain a failed action on a newly delivered Archive Project reference, a switch to GUI after `find` and a rooted observation did not expose the desired target, and a reread of an already delivered target. These cases illustrate when direct continuation gives way to further observation or another channel.

D.4 GUI Action-Space Replacement

GUI vocabulary and calling behavior. To examine whether GUI vocabulary changes interface use, we compare GPT-style and Claude-style GUI action spaces within HCU for GPT-5.4 and Claude Opus 5.5. The replacement changes action names, parameter conventions, available operations, and the corresponding GUI-specific skill, while retaining AX, programmatic tools, batching, and layout guards. The GPT-style vocabulary uses a `type` discriminator; the Claude-style vocabulary uses `action`, separate click names, string key chords, and operations including zoom and held-key input (Appendix F.3). Both run inside HCU's wrapper. Results include every task, including those completed without GUI calls.

Table 14: GUI action-space replacement within HCU: Score (%); differences are Claude-style minus GPT-style.

Model	GPT-style	Claude-style	Difference (pp)
Claude Opus 5.5	71.70	72.51	+0.82
GPT-5.4	65.59	63.56	-2.03

With the Claude-style vocabulary, Claude Opus 5.5 gains 0.82 Score points, with +1.75% Steps and +4.74% cumulative input. GPT-5.4 loses 2.03 Score points in an unmatched comparison whose runtime differences are specified below (Table 14).

Variants and runs. Each run uses the standard task set, 100 successful model responses, and a 1,800-second task timeout. Claude Opus 5.5 uses low reasoning effort and two runs per vocabulary, with matched harness, runner, schema adapter, image-export policy, and budgets. The HCU implementation, image, and GUI-specific skill change with the vocabulary; AX and general hybrid instructions are retained. Round 1 runs at concurrency five per arm; round 2 launches both arms together with Claude Code 2.1.233 at concurrency ten per arm. The GPT-style runs are the two HCU runs in the cross-model comparison and score 71.29 and 72.10; the Claude-style runs score 72.16 and 72.86.

GPT-5.4 uses xhigh reasoning effort. Its GPT-style arm averages the three HCU ablation runs (66.16, 65.65, and 64.95); its Claude-style arm averages two runs (63.20 and 63.91). Model, AX/Bash availability, and budgets match, but concurrency, provider lifecycle, cache isolation, and machine load differ between vocabularies. Its Score contrast therefore reflects both vocabulary and runtime differences.

Calling behavior. Call-level metrics use the second run of each arm. Under the Claude-style vocabulary, Claude Opus 5.5 issues GUI input on 3.3 percentage points more tasks and places multiple actions in 4.9 points more GUI requests. Under the GPT-style vocabulary it requests 27 unsupported actions, mainly `triple_click`, across 25 tasks; these rejections disappear with the Claude-style vocabulary. GPT-5.4 instead issues 0.67 more GUI requests per task, mostly standalone `zoom` requests, without changing its multi-action rate.

For this comparison, convention errors count explicit violations of action names, argument structure, or calling requirements, including attributable harness dispatch failures; they are distinct from the schema-error metric in the skill ablation. Per 1,000 AX/GUI requests with a recorded outcome, these errors decrease from 92.49 (80/865) to 48.14 (44/914) for Claude Opus 5.5 and increase from 10.44 (38/3,639) to 41.48 (156/3,761) for GPT-5.4. The vocabulary changes calling behavior differently across the two models.

E AGENT BEHAVIOR ACROSS MODELS

This appendix supports Section 4.4 by connecting each model's performance gains to channel use, observation and execution, and the Steps spent on desktop interaction. The task sets, paired comparisons, counting units, and trace-analysis definitions are given in Appendix B.

E.1 DETAILED CROSS-MODEL RESULTS

Tasks and rounds. The comparison in Section 4.4 pairs HCU and CUA Driver within each of the four configurations defined in Appendix B.2. Resource changes in Table 3 are $(\text{HCU}/\text{CUA Driver} - 1) \times 100\%$ of per-task means, with pricing and aggregation as specified in Appendix B.4.

Every configuration scores all 361 tasks except CUA Driver with Claude Opus 5.5, where the provider's cybersecurity safeguards refused one task; the Claude Opus 5.5 comparison pairs the other 360 tasks (Table 3). GPT-5.4 contributes the three complete runs per configuration of the ablation in Section 4.3. Claude Opus 5.5 contributes two HCU runs and one CUA Driver run; its second HCU run is the GPT-style arm of the action-space comparison, which uses the default HCU tools and skill. Gemini 3.8 Flash and Grok 4.7 contribute one run per configuration. Table 3 averages each task over the runs of a configuration before pairing. Per-run HCU Scores are 66.16, 65.65, and 64.95 for GPT-5.4 and 71.29 and 72.10 for Claude Opus 5.5. The channel-use analysis and Score decomposition use the same paired tasks within each model and average all its runs; the post-error and recorded-termination figures use these task sets but retain their first-run scope (Appendix B.5).

Application	n	GPT-5.4	Claude Opus 5.5	Gemini 3.8 Flash	Grok 4.7
Chrome	46	+7.8	+2.2	+2.2	+2.2
GIMP	26	+5.1	+7.7	0.0	+26.9
Calc	47	+2.1	-7.4	+29.8	0.0
Impress	47	+6.8	+13.8	+38.3	+14.2
Writer	23	+5.8	+17.4	+43.5	+4.3
Multi-app	93	+2.3	+0.2	+7.0	+5.3
OS	24	+1.4	+2.1	0.0	+8.3
Thunderbird	15	+6.7	-10.7*	0.0	-6.7
VLC	17	+1.8	+11.1	0.0	+6.0
VS Code	23	-10.1	-2.2	-13.0	+8.7

Score difference (pp)
HCU minus CUA Driver
-45 -30 0 +30 +45

Figure 15: HCU–CUA Driver Score differences by application; tasks are averaged over rounds before pairing. n: paired tasks; asterisk: Claude Opus 5.5's 14-task Thunderbird comparison.

Gains are not uniform across applications (Figure 15). Gemini 3.8 Flash gains most in LibreOffice Calc, Impress, and Writer (+30, +38, and +43 points), while Grok 4.7 gains most in GIMP and Impress (+27 and +14 points) and loses 7 points in Thunderbird.

Each model runs in its provider's own harness, so the comparison evaluates HCU and its skill as a package within each model–harness configuration.

E.2 CHANNEL CHOICE AND OBSERVATION–ACTION DIVISION

Section 4.4 counts, on each task where HCU issued GUI requests, whether GUI only took screenshots and whether the model also acted through AX. The counts use tasks whose GUI requests all

have recorded actions and average over runs. GUI only took screenshots on 77 of the 189 tasks where GPT-5.4 used it, 134 of 275 for Gemini 3.8 Flash, 100 of 254 for Grok 4.7, and 28 of 158 for Claude Opus 5.5; on 59, 81, 77, and 1 of these tasks, respectively, the model acted through AX. Relative to these eligible tasks with GUI use, these are 31%, 29%, 30%, and 0.6%.

A recorded cross-channel interaction. In the GIMP layer-naming example in Table 8, the agent enters "Square" through AX. The returned text and a focused AX observation omit that value, whereas a subsequent screenshot shows it in the dialog; the next action uses the AX reference for OK. Visual observation thus supplies evidence missing from the returned AX state while execution remains reference-addressed. The whole run has nine Steps, seven completed tool calls, no shell calls, and no recorded tool errors. This selected trace illustrates the division of observation and execution; it does not establish the reason for every screenshot in the aggregate counts.

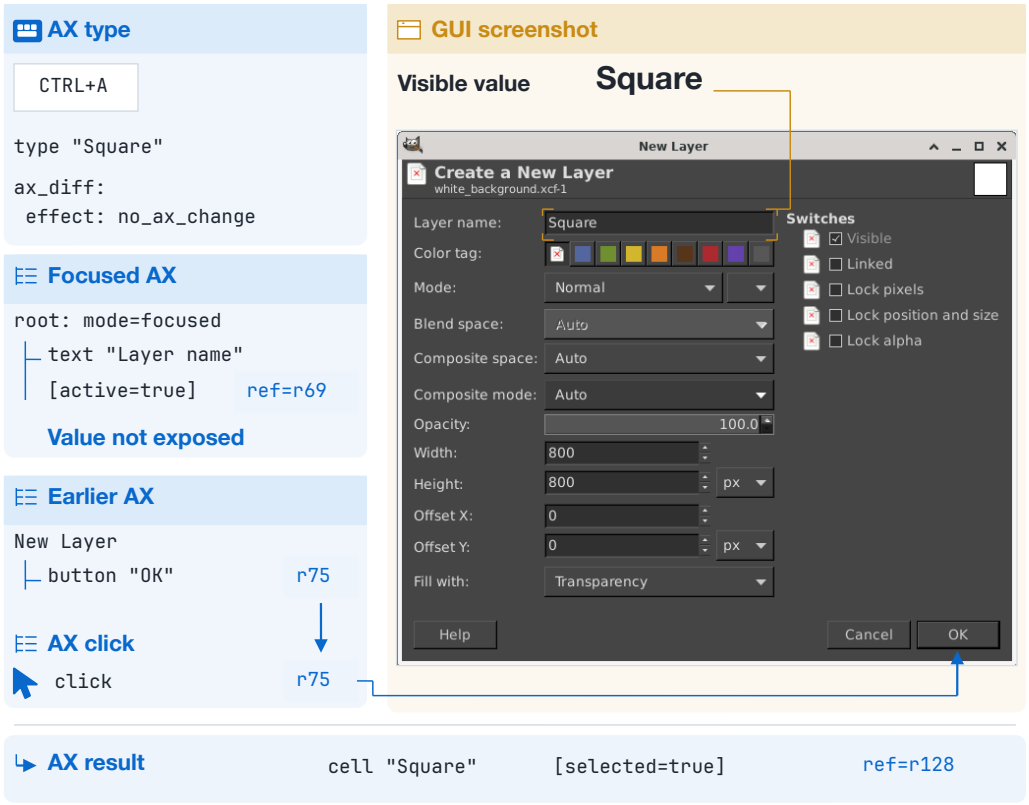


Figure 16: Recorded GIMP interaction, HCU (GPT-5.4), OSWorld 1.0. A screenshot reveals the value omitted by AX; execution resumes through the earlier OK reference. The final AX return does not verify saving. Callouts mark visible controls, not provider bounds.

E.3 Decomposing Score Gains and Steps

An additive Score decomposition. Figure 5 groups tasks within each HCU run by whether the agent called AX or GUI. A task can change groups between runs. For each HCU run, the analysis subtracts the corresponding task's mean CUA Driver score, assigns that difference to the group observed in that HCU run, and then averages contributions over HCU runs and all paired tasks. If R is the number of HCU runs, N the number of paired tasks, and $\bar{s}_i^C$ the mean CUA Driver score for task i, the contribution of group g is

$$\Delta_g = \frac{100}{NR} \sum_{r=1}^{R} \sum_{i=1}^{N} \mathbf{1}[g_{ir} = g](s_{ir}^H - \bar{s}_i^C). \tag{2}$$

The two contributions sum to the total Score difference in Table 3; group task counts are also averaged over runs. This decomposition locates the observed gains without treating behavior selected during execution as an independent intervention.

Steps and recorded desktop-tool use. We retain four groups: both interfaces make desktop requests, only HCU does, only CUA Driver does, or neither does. Desktop use means at least one recorded AX or GUI request for HCU, or a CUA Driver request, including its browser tools, for CUA Driver; it does not identify desktop automation performed through programmatic tools. A task may change groups between runs. For each paired task i, we equally weight every combination of its R_H HCU runs and R_C CUA Driver runs, without assuming that repetition indices are matched. Writing g_{irs} for the group of HCU run r and CUA Driver run s, we obtain

$$w_g = \frac{1}{NR_HR_C}\sum_{i,r,s}\mathbf{1}[g_{irs}=g], \qquad \Delta_g = \frac{1}{NR_HR_C}\sum_{i,r,s}\mathbf{1}[g_{irs}=g](t^H_{ir}-t^C_{is}). \tag{3}$$

Thus $\Delta_g = w_g(\bar{t}^H_g - \bar{t}^C_g)$ and $\sum_g \Delta_g = \bar{t}^H - \bar{t}^C$ exactly, using the same per-task, within-configuration averaging as Table 3. The equivalent task count Nw_g can be fractional and is not a count of distinct tasks. The runs and paired tasks are those specified in Appendix E.1.

Table 15: Steps decomposition by desktop-tool use, weighting run combinations equally within each task. N_g: equivalent task count; Δ_g: Steps contribution per paired task (negative favors HCU). Contributions sum to the overall difference before rounding.

Model	Desktop-use group	N_g	HCU	CUA Driver	Δ_g
GPT-5.4	Both	200.11	33.98	57.00	−12.7590
	HCU only	65.56	32.35	24.69	+1.3915
	CUA Driver only	5.56	16.60	49.28	−0.5029
	Neither	89.78	14.61	15.70	−0.2718
	Overall	361	28.60	40.74	−12.1422
Claude Opus 5.5	Both	83.50	6.71	9.86	−0.7292
	HCU only	78.50	7.94	5.18	+0.6014
	CUA Driver only	4.50	5.56	9.78	−0.0528
	Neither	193.50	4.21	4.00	+0.1139
	Overall	360	5.62	5.69	−0.0667
Gemini 3.8 Flash	Both	298	62.00	76.68	−12.1163
	HCU only	15	54.93	51.73	+0.1330
	CUA Driver only	10	39.80	42.20	−0.0665
	Neither	38	37.45	38.89	−0.1524
	Overall	361	58.51	70.71	−12.2022
Grok 4.7	Both	239	37.12	51.99	−9.8421
	HCU only	54	34.69	24.52	+1.5208
	CUA Driver only	5	6.60	38.40	−0.4404
	Neither	63	13.37	14.29	−0.1607
	Overall	361	32.19	41.11	−8.9224

Across these models, the both-desktop group takes 19.14–40.38% fewer Steps with HCU. For Claude Opus 5.5, this group's 83.5 equivalent tasks contribute -0.7292 Steps per paired task. The 78.5 equivalent HCU-only tasks take 53.26% more Steps and gain 7.64 Score points within the group, contributing $+0.6014$ Steps per paired task. The CUA Driver-only and neither groups contribute -0.0528 and $+0.1139$, respectively. Together these yield -0.0667 Steps per task (-1.17%). Group sizes alone therefore do not explain the net change. These descriptive groups locate differences without isolating the effect of adding desktop use.

Application concentration. LibreOffice Calc, Impress, Writer, and Thunderbird comprise 132 of the 361 OSWorld 1.0 tasks (36.6%). After averaging each task over rounds, these applications account for 70.0% of the net Steps saved by HCU relative to HCU w/o AX with GPT-5.4, and 69.4%, 77.6%, and 40.5% of the savings relative to CUA Driver with GPT-5.4, Gemini 3.8 Flash, and Grok 4.7, respectively. Net savings sum the comparator's Steps minus HCU's Steps across

tasks, including negative differences. With Claude Opus 5.5, these applications instead require 48.5 more Steps with HCU, while the total across its paired tasks falls by 24 Steps. The application-level pattern therefore varies across models, complementing the desktop-use decomposition above.

E.4 Budget Exhaustion and Termination

Figure 17 shows recorded termination reasons and Steps. Budget termination means the runner stopped because the response budget was exhausted. The analysis below instead groups tasks that reach the response limit, including GPT-5.4 tasks that complete on Step 100. In the first run of each GPT-5.4 configuration, one task completes at that limit. Grok 4.7's other-or-unknown category holds normal exits without a completion marker.

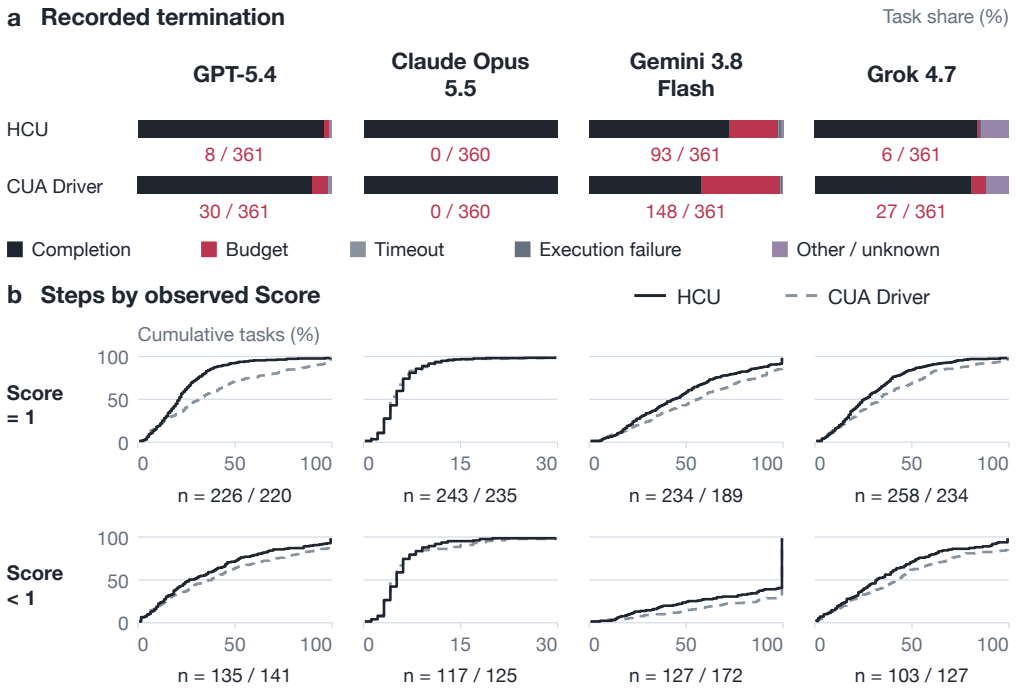


Figure 17: First-run outcomes on each model's paired tasks (Appendix E.1). (a) Termination shares; counts denote termination due to the response budget. Completion need not mean full credit. (b) Within-group cumulative Steps distributions; n: HCU / CUA Driver tasks. Axes differ because harnesses count Steps differently.

Where the step budget binds. Each OSWorld 1.0 task allows 100 model responses. With CUA Driver, Gemini 3.8 Flash reaches this limit on 148 tasks, against 93 with HCU; GPT-5.4 reaches it on about 29 tasks per round and Grok 4.7 on 27, and Claude Opus 5.5 never does. To examine whether Gemini 3.8 Flash benefits especially on long tasks, we define that group using other models: the 82 tasks on which GPT-5.4, in any run, or Grok 4.7 reaches the response limit with CUA Driver. Gemini 3.8 Flash gains 30.5 points per task on these long tasks and 7.7 on the rest, and the long tasks account for 54% of its gain.

Figure 18 additionally groups tasks by whether they reach the response limit in each model's own CUA Driver runs. For Gemini 3.8 Flash, the 148 tasks at the limit contribute +13.38 points to the overall +12.90-point gain; for GPT-5.4 and Grok 4.7, most of the gain comes from tasks below the limit. This same-run decomposition describes where gains occur, but couples group membership to the comparator's outcome. The independently defined long-task group above separates that selection from Gemini 3.8 Flash's own runs.

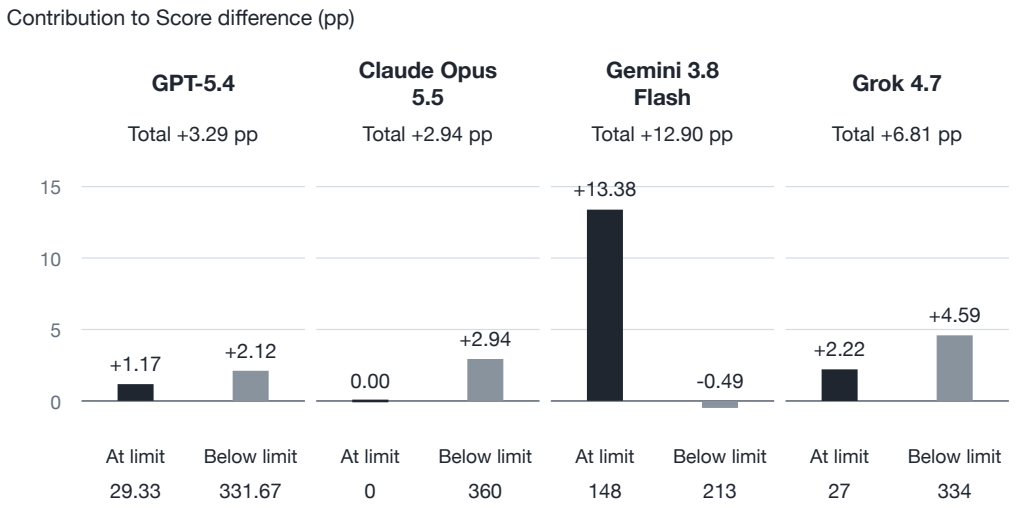


Figure 18: Score differences grouped by whether CUA Driver reaches the response limit, averaging all rounds. Counts are mean tasks per CUA Driver round (three for GPT-5.4, one for other models). The text also examines a long-task group defined using other models.

E.5 Invocation Errors and Subsequent Behavior

Figure 19 breaks down the first new call after each error. For CUA Driver, an error is any failed CUA Driver call, including its browser tools. This sequential analysis uses trajectories in which all calls have terminal records: 343/338 for GPT-5.4 and 359/358 for Gemini 3.8 Flash (HCU/CUA Driver), and all paired trajectories for Claude Opus 5.5 and Grok 4.7.

What follows an error differs across model–harness configurations. After an AX or GUI error, the next new request moves to another channel in 69 of 186 events for GPT-5.4, 64 of 260 for Gemini 3.8 Flash, and 120 of 366 for Grok 4.7, but in only 1 of 55 for Claude Opus 5.5. Claude Opus 5.5 also rarely leaves CUA Driver after its errors (5 of 105), compared with 23–44% for the other models. Its tendency to continue with the same tool appears under both interfaces.

These cross-model continuation counts complement the skill-ablation error analysis in Appendix D.2.

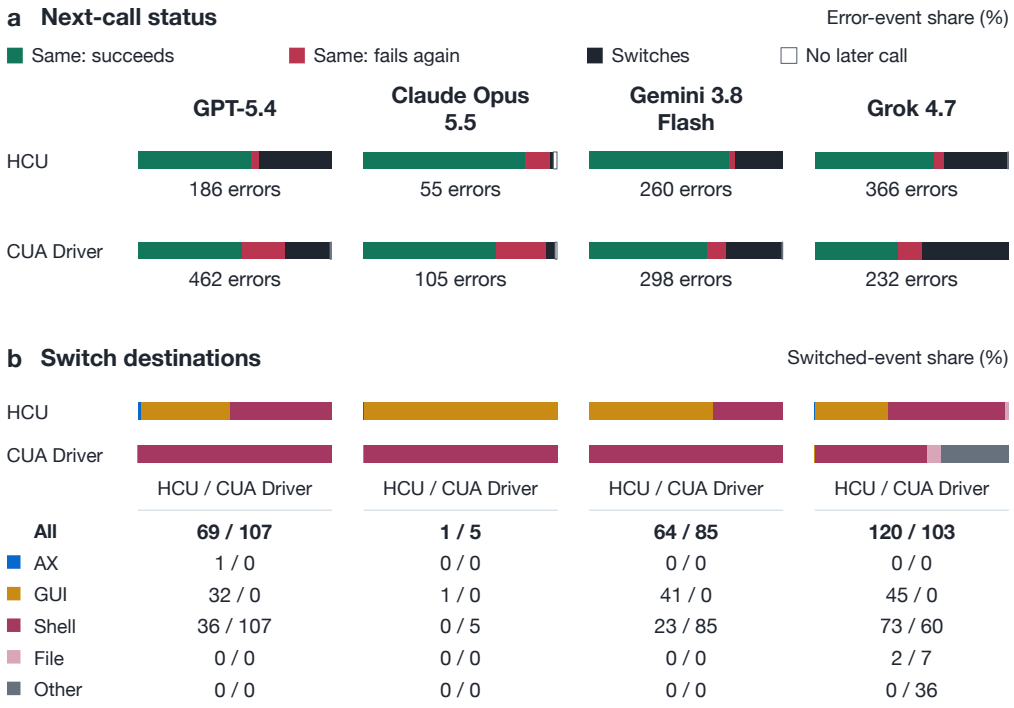


Figure 19: First-run post-error behavior. (a) Next-call shares over error events, including no later call. (b) Destinations conditional on switching. Counts: HCU / CUA Driver; All gives each bar's denominator. Status denotes call errors, not task outcomes.

F TOOL DEFINITIONS AND SKILL

This appendix presents HCU's server instructions, tool descriptions, base policy, and hybrid skill, together with the two GUI action vocabularies and recorded feedback excerpts. The supplementary skill is distinct from built-in tool definitions and server instructions, which remain available without the skill. Section 3.3 describes its role; Appendix B.6 explains instruction injection. Blue boxes contain source text; dark boxes contain recorded feedback.

F.1 SERVER INSTRUCTIONS AND TOOL DEFINITIONS

HCU publishes one server instruction and two MCP tools, reproduced below. Table 4 summarizes the request fields of each mode. The AX `act` mode accepts the actions `activate`, `close`, `click`, `keypress`, `scroll`, `set_value`, and `type`; each can carry a postcondition, and assertions and postconditions share the predicates `exists`, `state`, `value`, `text_contains`, and `window_title_contains`. As compact JSON, the input schemas take 19,083 bytes for AX and 4,384 bytes for GUI.

Server instructions

```
Control the already-running desktop with two complementary tools. AX is app-scoped semantic observation and
↪ action; GUI is full-desktop visual observation and absolute-pixel input. Use AX for named controls and
↪ GUI for pixels or content absent from the accessibility tree. AX find uses explicit equals/contains
↪ fields. Copy returned rN/wN refs exactly; never invent placeholders. A targeted ref owns its app and
↪ window, so omit app, app_ref and window_ref. Use window_ref only when every action is untargeted keyboard
↪ input. Batch only actions valid in one UI generation; split after likely dialog, window or navigation
↪ changes. Window activate/close uses wN and close must be last. AX action acceptance and visual change are
↪ evidence, not proof of task success; AX cannot prove files, persistence, network state or pixels. GUI
↪ keyboard/text uses current focus. AX keyboard/pointer fallbacks use global desktop focus, not background
↪ input. After GUI mutation, refresh AX before reusing semantic state. AX target-window screenshot
↪ coordinates are never GUI coordinates; GUI coordinates use the exact delivered full-desktop image.
```

`AX` tool description

```
Semantic observe/find/act/assert_ax. Observe/find issue refs; act consumes refs only. Reads use app, app_ref,
↪ or window_ref; continue find with fN. Act returns current diff/tree/lifecycle; reread only if required.
↪ Use batch or action postconditions for known results. Element actions require rN, window actions require
↪ wN; act never accepts app, selector, or index. Targeted act rejects app_ref/window_ref; refs own both.
↪ Never invent refs. One wN per batch; targetless batches require window_ref. Stale refs never rebind.
↪ Working output may hide controls; find searches canonical AX records. Refs resolve live per action;
↪ continuity failure stops the batch. Batch known same-window steps. Split before window changes or unknown
↪ next targets. AX proves only AX-visible facts, not pixels/files/persistence/task completion. Limits: 32
↪ actions, 20000 text chars.
```

`GUI` tool description

```
Full-desktop action batch; screenshot alone observes. Batch validates before input and returns one final
↪ screenshot. Coordinates are delivered-image pixels. AX activates/closes windows. type/keypress use
↪ current focus. wait is one second, no observation, max 20s/batch. scroll_x/scroll_y are Computer deltas.
↪ settle_ms checks visual stillness, not semantic success; omitted default 800ms, max 3000ms.
```

F.2 BASE POLICY AND SKILL

The instruction package separates common policy in `instructions/base.md` from the supplementary skill in `instructions/hybrid.md`. Both are plain text injected through the harness's existing context mechanisms, not client-native skills that must be discovered or activated. The base policy covers scope, safety, evidence, and completion; the hybrid skill covers channel choice, AX and GUI interaction, and recovery. The text below is reproduced without rewriting its literal terminology.

Current base policy (`instructions/base.md`)

```
# Computer-Use Principles

Complete the user's instruction end to end using the available tools. Inspect
real state, act within the requested scope, and verify the delivered result.
For actionable tasks, carry the work through execution rather than stopping at
a proposal. Keep progress and final responses concise.
```

```
## Scope And Safety

- Outcome-only tasks may use shell commands, file editing, application APIs,
  browser automation, accessibility, or GUI input. Honor an explicitly exclusive
  method, a method that is itself the user's purpose, and requirements about
  visible state. Ordinary wording such as "use Chrome" does not alone exclude
  an equivalent route; "leave the page open" requires the actual browser state.
- Preserve unrelated data. Completing one change does not authorize removing
  related history, caches, or other independently managed state.
  Before recursive deletion, movement, overwriting, or permission changes,
  resolve the absolute target and preview the affected paths.
- Never fabricate output, files, application state, screenshots, or verification.
  Do not read benchmark task definitions, dataset configuration, evaluator or
  gold internals, hidden checks, other tasks' traces, or task-derived cross-run
  memory, even when reachable or public.
- Before publishing, pushing, sending, or uploading files, inspect the exact
  outgoing set for credentials and secret configuration without printing secret
  values. Remove exposed secrets only when authorized. If preserving the source
  conflicts with safe publication, ask rather than silently leaking or editing it.
- Ask for information only the user can supply when it materially affects scope,
  safety, or an external or irreversible result. Inspect available evidence
  first. Use the configured user-response channel when one is supplied.
- When the user invites questions and a relevant category, range, eligibility,
  status, or assignment rule remains ambiguous, clarify before the affected
  external action. Bundle material ambiguities into one question. For
  classification or aggregation, account for every source item, resolve overlap
  and exclusion rules, and recompute dependent results after clarification.
  A rule about one metric does not establish eligibility for another.

## Evidence And Completion

- Match evidence to the requirement: semantic state for control identity and
  values, pixels for appearance and visible state, structured artifact reads
  for files, and effective runtime state for settings or services.
- Action acceptance or input emission alone does not prove the outcome.
  After the final mutation, obtain evidence that directly covers each requested
  result. A returned postcondition or final observation can provide that
  evidence; a separate call or second channel is unnecessary unless something
  remains unproven or observations conflict.
- Verify exact target, value, type, scope, destination, and required persistence.
  Commit save or confirmation dialogs. Treat defaults and preferences as
  application/user settings unless the instruction limits them to one object.
  Verify them through effective configuration or a newly created object,
  not just an edited existing object.
- After direct edits, account for caches, locks, and running processes. Reload
  or reopen the application when it could overwrite the change or the result
  must remain visible. Reopening solely to reconfirm an already-proven file
  fact is unnecessary.
- Background shell output does not satisfy a request for visible terminal
  output. A saved file does not prove that the requested page or document is
  displayed.
- Stop when every requirement has authoritative final-state evidence. Clean up
  only task-created temporary state that affects the result, lifecycle, or data.
  Report limitations honestly. One failed tool or channel is not a hard blocker
  while another available general-purpose route can complete the task.
```

The hybrid skill makes the three roles in Section 3.3 concrete: channel choice, efficient interaction, and verification and recovery. It distinguishes semantic activation from selection and defers argument shapes to the tool schemas. To sustain progress, it reuses sufficient final observations or postconditions and bounds unsuccessful lookup and repeated attempts. Recovery follows the returned continuation instructions: an explicit stop overrides retry guidance. Known non-delivery concerns the requested input; a pending focus request may still take effect (Appendix A.4). Final screenshots describe normal GUI returns; preparation failures can stop without a recovery observation.

Current hybrid skill (`instructions/hybrid.md`)

```
# Hybrid Computer Use

Use shell, files, browser automation, application APIs, AX, and GUI as
complementary channels. Choose the shortest reliable route to the requested
state. Follow the current tool schemas for argument shapes and supported actions.

## Choose A Channel

- Prefer structured file edits, CLI tools, or documented APIs for durable data.
  Inspect format and metadata before selecting a parser; an extension is not
  proof of format. Use the browser runtime, when available, for web content,
```

```
  navigation, downloads, and browser state.
- Discover local application APIs from live process, configuration, help, or
  protocol information. Make one bounded capability probe and inspect the
  relevant command inventory. A liveness response is not proof of a usable
  command. Do not guess endpoints, bypass authentication, or enable
  network-exposed remote access. A documented user/session-local IPC endpoint
  or isolated headless operation is acceptable only when it exposes no network
  listener and cannot overwrite another instance's unsaved state.
- Use AX for named controls, values, states, and semantic identity. Use GUI for
  pixels, canvases, spatial layout, dragging, or controls absent from AX.
  A partial tree does not invalidate its returned targets. After one targeted
  semantic lookup fails, use GUI when it can answer the next decision.
- A visible transient window may be absent from AX. Take one GUI screenshot
  instead of repeatedly reading the same tree. Return to AX when new semantic
  state or a semantic target is relevant.
- Once source, destination, and operation are known, act. Additional diagnostics
  must answer a concrete blocker. Prefer a channel already producing reliable
  evidence over opening equivalent investigation paths.

## AX Interaction

- Observe the relevant application or window. Use content scope for a page or
  document with one identifiable content root; use window scope for menus,
  toolbars, tabs, dialogs, settings, and other application chrome.
- For read modes, provide one identity source: bootstrap app, app ref, or
  window ref. Prefer returned refs. A window ref already owns its application.
  AX trees are compact projections, not complete visible text or hidden DOM.
- Use `find` for a known role, name, text, value, or state. It searches canonical
  records even when the displayed tree omits neutral containers. Use a focused
  or target-rooted observation for local context before requesting a full
  window tree. Projection summaries are not actionable targets.
- Mutations use current element `rN` or window `wN` refs, never indices,
  selectors, or invented refs. Copy refs exactly. A targeted ref already owns
  its application and window; supply `window_ref` only for an entirely
  untargeted keyboard batch.
- Batch actions only while later targets and focus are expected to remain valid
  in one logical window. Make an action that can open, close, replace, or
  navigate a window the last action, then use returned fresh state.
  Window activation and closure use `wN`; closure must be last.
- Semantic click can invoke default activation, especially on selectable items.
  Do not assume it only selects. If later input depends on an earlier state
  change, attach the expected predicate as that action's `postcondition`.
  It gates subsequent input but cannot undo the earlier action.
- AX input can require foreground focus; semantic addressing does not imply
  background delivery. `type` supplies literal text; `keypress` supplies keys.
- Keep the default post-action diff for ordinary actions and continue from its
  evidence. Unchanged or compact output is still fresh. Use no observation only
  when no immediate readback is needed, or a target-window screenshot when
  pixels are required. Its coordinates are not GUI coordinates.
- Put a known final AX predicate in the batch postcondition. Use a separate
  `assert_ax` when a later decision requires it. Prefer positive next-state
  conditions to disappearance of old nodes. Assertions prove provider-visible
  semantics, not file persistence, pixels, network state, or task completion.

## GUI Interaction

- Start with a GUI screenshot when desktop state is unknown. A screenshot
  action must stand alone. Use coordinates from the exact returned full-desktop
  image and its reported dimensions, not a chat preview, AX window image,
  thumbnail, or independent capture.
- Batch actions determined from the same screenshot and stable focus. One call
  returns one final screenshot, not intermediate observations. End a batch
  after an action that can open, close, switch, resize, or reposition a window
  or dialog; recompute later coordinates from the returned image.
- GUI text and keys use current desktop focus. A visible focus click and its
  dependent input can share a batch if the click cannot change top-level layout.
  GUI does not target apps or semantic refs and does not activate named windows.
- A stale-layout rejection returns fresh pixels without emitting the rejected
  coordinate action. Recompute from that image. If earlier batch actions ran,
  do not replay them automatically.
- Consecutive known keyboard actions already receive a processing barrier.
  Normally leave `settle_ms` at its default: it checks final visual stillness,
  not semantic readiness. Fixed waits only delay; they expose no intermediate
  state. For an observable transition, use a postcondition, assertion, or fresh
  observation. When no predicate is available, use one delayed sample and
  decide from it rather than stacking waits.

## Recovery And Efficient Evidence

- An explicit stop response overrides normal retry and observation guidance.
  Stop desktop automation and report the uncertainty. Even when requested input
  was not emitted, an earlier OS focus request may still take effect; neither
```

```
  a new screenshot nor a worker restart proves it has settled.
- `not_executed` with `input_events_emitted=false` means that action caused no
  later UI change. If input was emitted or its effect is uncertain, inspect
  returned evidence before deciding whether any action remains necessary.
- Reuse fresh refs and observations already returned. Do not repeat an
  observation solely because its diff is small. When a stale ref is rejected,
  discard all targets from that failed call; use current returned refs or one
  fresh `find`/observation. Do not reopen an absent dialog just to retry it.
- After GUI input or possible desktop mutation, refresh AX before reusing
  semantic state. A rejected GUI batch with no input or mutation does not
  itself invalidate AX state.
- An unchanged AX tree does not mean formatting, pixels, or file state stayed
  unchanged. Change the evidence source instead of repeating the action.
- After two attempts based on the same hypothesis without new evidence, stop
  retrying it. Make one targeted read, revise the hypothesis, and change route.
  Do not nudge coordinates, add Enter, or relaunch application instances merely
  to escape uncertainty.
- Combine related read-only shell diagnostics into bounded operations. Prefer
  parsers or APIs for bulk data. If clipboard transfer is necessary, keep its
  owner alive through paste; after one focus-verified failure, change mechanism.
- A final observation or postcondition is enough when it directly proves the
  requested result. Use another channel only for remaining requirements such
  as exact data, persistence, or visible appearance.
```

The companion files `instructions/ax-only.md` and `instructions/gui-only.md` retain programmatic interaction while describing only the selected desktop channel. They contain 743 and 683 whitespace-delimited words, respectively, compared with 1,124 for the hybrid skill and 561 for the base policy. The source snapshot includes all four files. Selecting a channel-specific skill does not itself remove tools; the corresponding channel configuration must also restrict tool exposure.

Appendix B.6 describes how each harness receives these instructions and how the runner supplies the task-completion protocol.

F.3 GUI Action Vocabularies

The GUI action-space replacement of Appendix D.4 swaps HCU's default GUI vocabulary for a Claude-style one and keeps the tool wrapper, batching, final screenshots, and layout guards. Table 16 lists both vocabularies. The Claude-style skill also requires `zoom` and `screenshot` to be issued alone.

Table 16: GUI action vocabularies. The GPT-style space is HCU's default.

Operation	GPT-style (`type` field)	Claude-style (`action` field)
Observe	`screenshot`	`screenshot`, `zoom`, `cursor_position`
Click	`click` with `button` and modifier `keys`; `double_click`	`left_click`, `right_click`, `middle_click`, `double_click`, `triple_click`
Pointer	`move`; `drag` along a `path`	`mouse_move`; `left_click_drag`, `left_mouse_down`, `left_mouse_up`
Scroll	`scroll` with `scroll_x` and `scroll_y` deltas	`scroll` by direction, in wheel clicks
Keyboard	`type`; `keypress` with a key array	`type`; `key` with a key-chord string; `hold_key`
Wait	`wait`, one second	`wait`

F.4 Recorded Feedback and Error Messages

Action responses separate execution evidence from the selected post-action observation (Appendix A.3). Figure 9 shows a recorded AX diff and Figure 11 a layout-guard rejection followed by an unknown effect. The excerpts below come from round 1 of the OSWorld 1.0 ablations, with GPT-5.4 in Codex. The batch-message prefix is normalized to `AX BATCH`. The excerpts distinguish rejection before input, an expired target, and an interrupted sequence. Appendix A.4 explains the stop instructions for uncertain preparation failures.

Schema rejection before any input (HCU)

```
AX BATCH REJECTED BEFORE ACTION 1: actions[0] type "click" does not accept field "ref"
action_outcome=not_executed
actions_completed_without_call_error=0
No accessibility action was attempted.
```

Stale window reference (HCU)

```
stale_window_ref: the referenced logical window no longer exists.
continuation: state=observe_required retry_same_request=false route=AX
```

Batch stopped by a failed postcondition (HCU)

```
AX BATCH STOPPED AFTER ACTION 1/1 (click): its declared postcondition did not pass, so 0 later action(s) were
↪  not emitted. A fresh current-state observation is included when available.
```

Batch stopped without a final observation (HCU)

```
AX BATCH STOPPED AT ACTION 1/1 (click). 0 earlier action call(s) returned without a call error. This does not
↪  verify their delivery or task effect. No final batch observation was returned; observe fresh state before
↪  recovery.
```

Window changed inside a batch (HCU)

```
batch_window_changed: action 2/4 completed, but the logical window is changed before action 3. The remaining
↪  actions were not attempted; observe current state first.
```