CHAMELEONBENCH: EVALUATING THE CROSS-ENVIRONMENT ROBUSTNESS ACROSS HETEROGENEOUS SYSTEMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Language models increasingly automate programming and desktop workflows, yet success in one environment leaves open whether the same tasks can be completed under different execution conditions. We use *environment* broadly to denote the external conditions governing observations, available actions, and execution outcomes, including operating systems, software versions, configurations, permissions, hardware resources, network conditions, and runtime state. We present ChameleonBench, a benchmark of 100 tasks—80 Coding and 20 GUI—constructed around native execution behaviors that vary across environments. We focus on native Linux, macOS, and Windows deployments, which provide differences in filesystem, runtime, and application behavior while supporting equivalent task objectives and success criteria. The tasks form 30 mechanism families, with Coding covering individual mechanisms and their combinations. Using a fixed interaction harness and independent verification, we evaluate six models once per task and deployment, yielding 1,800 runs. With equal weighting across families, the strongest model achieves 75.2% mean success across deployments but 57.9% Robust Success Rate, which requires completing the same task on all three. The other models achieve 7.3–23.0% robust success. Grouping by mechanism family also yields more accurate predictions of performance differences across operating systems, on average, than grouping by interaction type and target OS alone. Our results show why task completion and its preservation across environments should be evaluated together.

1 INTRODUCTION

Language models increasingly carry out user requests by modifying programs, manipulating data, and operating desktop applications through tools (Yang et al., 2024; Wu et al.). Benchmarks such as WebArena, WorkArena, ScienceAgentBench, and OSWorld assess these interactive capabilities through functional or execution-based checks of task completion (Zhou et al., 2024; Drouin et al., 2024; Chen et al., 2025; Xie et al., 2024). Task completion, however, depends on the conditions under which a model acts (Ullrich et al., 2026). Success in one deployment therefore leaves open whether the same objective can be achieved when those conditions change. We use *environment* broadly to denote the external conditions governing observations, available actions, and execution outcomes, including operating systems, software versions, configurations, permissions, hardware resources, network conditions, and runtime state. Cross-environment robustness concerns whether models preserve task completion as these conditions change.

To assess whether models can reliably handle user requests across heterogeneous systems, we need a benchmark that follows the same tasks across changing execution conditions. Building such a benchmark poses two challenges. First, tasks must remain comparable while retaining the native differences that models need to handle. We address this by preserving equivalent inputs, objectives, and success criteria across deployments, with a shared interaction protocol and independent verification. Second, aggregate scores can hide recurring difficulties across different workflows, which may share the same environmental mechanism or require several mechanisms to be handled together. We therefore organize tasks into mechanism families, supporting analysis of shared performance patterns and comparisons between individual requirements and their combinations.

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

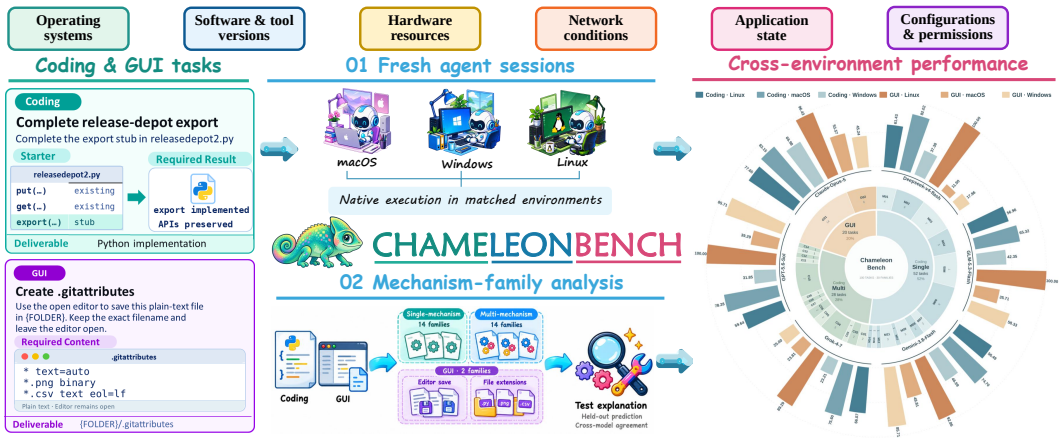


Figure 1: **Overview of ChameleonBench. Top:** Environment spans system, software, resource, and application conditions, instantiated on Linux, macOS, and Windows. **Left:** Coding and GUI examples retain their completion requirements across deployments. **Center:** Independent native runs use a shared harness and independent verification. Tasks are grouped into single- and multi-mechanism Coding families and two GUI families. **Right:** The inner ring shows 52 single-mechanism Coding, 28 combined-mechanism Coding, and 20 GUI tasks; the outer ring subdivides them into 30 mechanism families. Six bars per model report family-weighted SR for Coding and GUI on each OS.

To this end, we introduce **ChameleonBench** (Figure 1), a benchmark for evaluating foundation models' cross-environment robustness. We focus the evaluation on three native deployments—Linux, macOS, and Windows—as concrete instances of this broader notion of environment. Their differences in filesystem behavior, runtimes, applications, and configuration provide variation relevant to both programming and desktop workflows, while allowing the same task objectives and success criteria to be preserved. Models receive native feedback and can adapt their programs or actions accordingly. The benchmark comprises 100 tasks—80 Coding and 20 GUI—paired across all three deployments and organized into 30 mechanism families. Coding tasks cover both individual mechanisms and their combinations, while GUI tasks capture native application conventions. Comparisons within families examine whether environmental differences recur across distinct tasks and models. A fixed interaction harness and independent verifiers assess program behavior and desktop deliveries, supporting comparisons of task completion and its preservation across environments.

We evaluate six models once per task and deployment, obtaining 1,800 outcomes. With equal weight assigned to mechanism families, Claude-Opus-5 achieves the highest Robust Success Rate (RSR), the weighted fraction of tasks completed on all three deployments, at 57.9%, compared with its 75.2% mean SR. The other five models attain 54.4–60.5% mean SR but only 7.3–23.0% RSR. The deployment pattern also changes with the interface: every model performs best on macOS for Coding and on Linux for GUI. Beneath the overall averages, all six models exhibit family-specific reversals of aggregate OS rankings. Grouping tasks by mechanism family yields more accurate predictions of performance differences across operating systems, on average, than grouping by interaction type and target OS alone. Together, these results show that high deployment-level completion can coexist with substantial cross-environment failures, and that mechanism families organize recurring differences beneath aggregate scores.

Our contributions are threefold. We construct Coding and GUI tasks around native execution behaviors that vary across operating systems, using matched objectives and independent verification to test whether models preserve task completion under these changes. We organize tasks into mechanism families that cover individual requirements and their combinations, supporting analysis of recurring performance differences across tasks. Our evaluation of multiple models quantifies the gap between success on individual deployments and completion across environments, and reveals recurring performance patterns across models.

2 CHAMELEONBENCH

ChameleonBench evaluates foundation models' cross-environment robustness through paired Coding and GUI tasks under a fixed interaction harness. We define an *environment* as the external conditions governing observations, available operations, and execution outcomes, including the operating system, software, configuration, permissions, hardware, network conditions, and runtime state. Our experiments instantiate this setting on native Linux, macOS, and Windows deployments. Across deployments, each task preserves its business objective and success criteria while exposing the model to native execution behavior. The collection contains 100 tasks—80 Coding and 20 GUI—organized into 30 mechanism families. We use *agent* to denote a model's running instance within the shared harness; model comparisons follow a common execution protocol within each task interface.

2.1 TASK INTERFACES

Each task specifies a natural-language instruction, initial materials, permitted operations, a required delivery, and completion constraints $\mathcal{C}_t$. These constraints define the intended outcome and the invariants to preserve, including content, object identity, and naming or format requirements. For Coding tasks, models inspect project files, edit code, and execute commands to produce a working implementation that preserves the documented APIs. For GUI tasks, models observe screenshots and use mouse and keyboard actions in native applications to produce the requested desktop delivery. The evaluator maintains reference solutions, independent grading inputs, and mechanism annotations separately from the model-visible materials.

Each task is realized on all three operating systems with semantically equivalent inputs and the same completion constraints $\mathcal{C}_t$. Deployment-specific bindings identify corresponding local resources and initialize the project or desktop, while filesystems, runtimes, and applications retain their native behavior. Models can adapt their implementations, commands, or interface actions to the observed environment. The shared contract makes the resulting program behavior and desktop deliveries comparable across deployments.

2.2 MECHANISM ORGANIZATION AND PAIRED TASK CONSTRUCTION

Mechanism organization connects a task's business constraints to the environmental rules governing its execution. For Coding, an *elementary mechanism* specifies a rule, the operation that depends on it, and the constraint that the operation must preserve. A Coding task may exercise one mechanism or combine several; let $\mathcal{M}(t)$ denote its complete set. A Coding *mechanism family* is defined by the complete set: $g(t) = g(t')$ if and only if $\mathcal{M}(t) = \mathcal{M}(t')$. For GUI, two families organize tasks by native editor save conventions and file-manager extension visibility, linking application behavior to the required file names, formats, and contents. Each task belongs to one mutually exclusive family; Coding tasks additionally retain all of their elementary mechanism annotations.

Task authoring. We translate environment-dependent operations into programming and desktop workflows with explicit objectives and preservation requirements. Each package contains instructions, initial materials, initialization logic, a reference solution, and an independent verifier. Coding packages include reference and naive implementations; GUI packages define initial desktop states and reference operations in native applications. Within Coding, single-mechanism tasks concentrate on one environmental rule, while multi-mechanism tasks combine requirements within an integrated workflow. Instructions describe the required behavior, while mechanism annotations record the execution assumptions examined by the task.

Native pairing and construction checks. We instantiate the shared contract on Linux, macOS, and Windows, aligning input semantics, required deliveries, and success criteria while retaining native execution and application behavior. Initialization and reset procedures establish the task's project or desktop resources on each deployment. Native behavior probes and model trials provide construction evidence for reviewing environment-sensitive operations and revising tasks. Task records retain the applicable checks, construction conditions, and source materials.

Mechanism annotation and grouping. We assign mechanisms by inspecting task contracts, reference solutions, and verifier requirements. Coding operations governed by the same rule share an

elementary label, including filename normalization in record storage and journal ingestion. For Coding tasks with multiple mechanisms, the complete set determines family membership, independently of component order or label abbreviation. GUI membership follows the native editor or file-manager conventions exercised by each task. The resulting catalog fixes task membership and family assignments for the model comparison. Appendix C provides task contracts, grouping boundaries, and construction records.

Independent scoring and versioning. For a fixed model and harness, a run of task t on deployment o in repetition r receives the binary outcome

$$y_{t,o,r} = V_t(z_t, d_{t,o,r}, s_{t,o,r}) \in \{0, 1\}, \tag{1}$$

where z_t specifies expected behavior from trusted task inputs, $d_{t,o,r}$ denotes the delivered program or desktop artifacts, and $s_{t,o,r}$ denotes the inspected execution state. For Coding, the verifier executes the program on independent inputs and checks outputs, object identities, record completeness, names, text, and API behavior as required by the task. For GUI, the verifier inspects the resulting files and task state, checking the required names, formats, and contents. Success requires all verifier checks to pass, including the combined requirements of a multi-mechanism Coding task. For example, record preservation is checked against original keys and payloads, while a desktop export is checked against the requested file name and table contents.

Versioned packages and file manifests retain instructions, initialization resources, reference solutions, and graders. Construction records document the available native checks, while execution records link evaluated deliveries to their model runs and verifier diagnostics. Together, these materials support reproducing task preparation, program or desktop interaction, and outcome verification.

2.3 Benchmark Statistics

ChameleonBench comprises 100 tasks: 80 Coding tasks in 28 families and 20 GUI tasks in two families (Figure 2). Three native deployments per task yield 300 task–deployment pairs. The Coding subset spans 14 elementary mechanisms: its 52 single-mechanism tasks form 14 families, and its 28 multi-mechanism tasks form 14 additional families defined by distinct mechanism combinations. Of the latter tasks, 24 involve three mechanisms and four involve four mechanisms. The Coding families contain one to nine tasks each. The GUI subset contains 14 tasks involving native editor save conventions and six involving file-manager extension visibility.

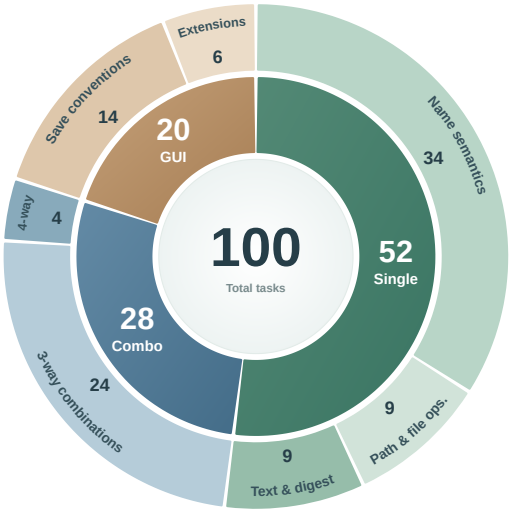


Figure 2: **Task composition.** Single and Combo denote single- and multi-mechanism Coding tasks. Outer sectors summarize broad task categories; numbers and angles encode task counts.

The tasks cover record export, manifest reconciliation, journal ingestion, file cleanup, archive extraction, data import, native document saving, and file management. Families organize the complete mechanism sets in Coding and the native interaction conventions in GUI. Our primary summaries average task outcomes within each family and then assign equal weight to all 30 families (Section 3). Each task contributes once through its assigned family, including Coding tasks with multiple mechanisms. Interface results and Coding single- and multi-mechanism results are normalized within their respective subsets; task-weighted summaries and composition analyses provide complementary views.

Table 1: **Comparison with environment-focused benchmarks.** Rows are grouped by evaluation emphasis; task correspondence describes how instances are linked across conditions.

Benchmark	Task form	Environment / variation	Task correspondence	Verification
Multi-environment task evaluation				
OSWorld	Desktop tasks	Linux / Windows	Migrated subset	State checks
CRAB	Cross-device tasks	Ubuntu + Android	Joint workflow	Graph checks
MMBench-GUI	Hierarchical GUI	Desktop / Web / mobile	Platform-specific tasks	Labels / states
Environment transfer and perturbation robustness				
TransBench	GUI grounding	Versions / platforms / apps	Related UI pages	Click locations
MacArena	Desktop tasks	macOS; Linux ref.	Migrated subset	State checks
AgentHijack	Desktop tasks	Injected corruptions	Clean / perturbed	State checks
ToolRobustBench	Tool-call episodes	Stage-wise perturbations	Clean / perturbed	Calls / results
ChameleonBench	Coding + GUI	Native Linux / macOS / Windows	All 100 tasks, three OS	Program / artifact checks

2.4 Benchmark Comparison

Table 1 compares task form, environmental variation, task correspondence, and verification. Multi-environment benchmarks cover desktop completion with an Ubuntu-to-Windows subset (OSWorld (Xie et al., 2024)), coordinated Ubuntu–Android workflows (CRAB (Xu et al., 2025)), and desktop, Web, and mobile evaluation from understanding and grounding to automation and collaboration (MMBench-GUI (Wang et al., 2025)). Transfer and robustness benchmarks examine GUI grounding across versions, platforms, and applications (TransBench (Lu et al., 2025)), native macOS tasks with a ported OSWorld subset compared against published Linux results (MacArena (Muryn et al., 2026)), desktop tasks paired with controlled environment corruptions (AgentHijack (Sun et al., 2026a)), and matched tool-calling instances with interface, intent, observation, and runtime-feedback perturbations (ToolRobustBench (Zheng et al., 2026)). ChameleonBench pairs all 100 Coding and GUI tasks across native Linux, macOS, and Windows under equivalent objectives, input semantics, deliverables, and success criteria; independent model sessions use the corresponding interface, and task-specific verifiers assess functional correctness. Its 28 Coding families group complete sets drawn from 14 elementary mechanisms, while two GUI families capture native editor and file-manager conventions, supporting task-level robustness measurement and analysis of recurring environment-dependent differences across interfaces.

3 Experimental Setup

Models and environments. We evaluate six foundation models—Deepseek-v4-flash, GLM-5.3-Flash, Gemini-3.8-Flash, Grok-4-7, GPT-5.6-Sol, and Claude-Opus-5—using the fixed `mini-chameleon-agent` harness on native Ubuntu 24.04, macOS 15, and Windows Server 2022. Each deployment retains its filesystem, runtime, software, and configuration (Section 2). All models use both interfaces: Coding supports resource inspection, program editing, command execution, and submission through function calls; GUI supports native application interaction through screenshots and mouse and keyboard actions. Within each interface, instruction templates, action contracts, context policies, and execution limits are shared across deployments, while models adapt to native execution feedback. Appendix D details model settings, prompts, tools, context and screenshot policies, and execution budgets.

Evaluation protocol. Each model runs once per task–deployment combination: 80×3 Coding and 20×3 GUI runs, totaling 300 per model and 1,800 outcomes. Tasks, success criteria, and assignments to 30 mechanism families remain fixed. Each run starts from a reset project or desktop and a fresh model session to solve the task on that deployment under the common execution protocol. An independent verifier checks program behavior or desktop deliveries against the shared contract; success requires every check to pass, including all combined constraints within one delivery for multi-mechanism Coding tasks. Task–OS outcome matrices support paired comparisons across deployments and models, while interaction records link outcomes to resource measurements and execution evidence.

Metrics and aggregation. We report *Success Rate (SR)*, *Robust Success Rate (RSR)*, *Environment Gap (EG)*, and *Cost*. For a fixed model, let $\mathcal{O}$ be the set of three deployments and $p_{t,o} = y_{t,o,1} \in \{0, 1\}$ the observed outcome of task t on deployment o, following Equation 1. For a reported task set S, let G_S be its mechanism families and $T_{g(t),S}$ the tasks in S belonging to the family of t. We assign $w_t^{(S)} = (|G_S|\,|T_{g(t),S}|)^{-1}$, averaging tasks within each family and then equally across the families in S. The completion metrics for S are

$$\mathrm{SR}_o = \sum_{t \in S} w_t^{(S)} p_{t,o}, \quad \mathrm{RSR} = \sum_{t \in S} w_t^{(S)} \min_{o \in \mathcal{O}} p_{t,o}, \quad \mathrm{EG} = \sum_{t \in S} w_t^{(S)} \left(\max_{o \in \mathcal{O}} p_{t,o} - \min_{o \in \mathcal{O}} p_{t,o} \right). \tag{2}$$

SR measures completion on each deployment; the cross-environment mean is $\overline{\mathrm{SR}} = \frac{1}{3} \sum_{o \in \mathcal{O}} \mathrm{SR}_o$. Under the single-run protocol, RSR is the weighted proportion of tasks completed on all three deployments, and EG is the weighted proportion whose pass/fail outcome changes across deployments.

4 Results and Analysis

4.1 Main Results: Cross-Environment Robustness

Table 2 compares six models on 100 tasks across three native deployments, comprising 1,800 outcomes. Each model completes one run per task and OS through the corresponding interface of the same harness. The table separates single- and multi-mechanism Coding tasks, GUI tasks, and the full collection. Families receive equal weight within each panel: 14, 14, two, and 30 families, respectively. On the full collection, Claude-Opus-5 achieves the highest SR on every deployment and the highest RSR at 57.9%, exceeding Gemini-3.8-Flash by 34.9 percentage points. The remaining models attain RSR values of 7.3–23.0%.

Task structure and interaction interface change the performance profile. On Coding tasks, all six models achieve lower completion on the multi-mechanism subset: mean SR decreases by 18.2–31.3 percentage points and RSR by 5.8–32.1 points relative to single-mechanism tasks. Claude-Opus-5 leads both subsets in mean SR and RSR, with RSR decreasing from 75.1% to 43.1%; GPT-5.6-Sol decreases from 28.6% to 0.0%. Deployment-specific rankings also vary: Deepseek-v4-flash leads multi-mechanism macOS SR at 87.9%, whereas Claude-Opus-5 leads all three deployments on single-mechanism tasks.

GUI introduces a different deployment pattern. Every model achieves its highest Coding SR on macOS, but its highest GUI SR on Linux, where scores range from 89.3% to 100.0%. GUI RSR ranges from 0.0% to 48.8%, with Gemini-3.8-Flash leading this subset, followed by Claude-Opus-5 at 41.7% and GPT-5.6-Sol at 39.3%. Deepseek-v4-flash illustrates the distinction between a favorable deployment and robust completion: it passes every GUI task on Linux but has no GUI task that passes on all three OSs. The interface-specific panels therefore reveal differences in both deployment preference and model ordering that a single overall score would omit.

Deployment-level success substantially exceeds paired completion. On the full collection, the five models other than Claude-Opus-5 attain 54.4–60.5% mean SR, while their RSR remains at or below 23.0%. Grok-4-7, for example, achieves 72.1% SR on macOS and 54.4% mean SR, but only 7.3% RSR. Claude-Opus-5 also exhibits a separation between mean SR (75.2%) and RSR (57.9%). Its EG of 32.6 points is smaller than the 72.6–80.0 points observed for the other models. Under the single-run protocol, EG is the family-weighted proportion of tasks whose pass/fail outcomes change across deployments.

Even the weakest aggregate OS score exceeds paired completion. For a fixed model, RSR cannot exceed the lowest deployment-level SR because a task must succeed on every deployment to count as a robust success, and both metrics use the same task weights. The observed difference ranges from 7.6 to 29.9 percentage points. GLM-5.3-Flash, for instance, achieves 43.4% SR on its weakest deployment, Windows, but 13.5% RSR. Claude-Opus-5 similarly decreases from 65.5% Windows SR to 57.9% RSR. Failures thus occur on different tasks across deployments; the lowest OS average does not identify the set completed everywhere.

Table 2: **Main results on ChameleonBench.** Six models use `mini-chameleon-agent`, with one run per task and OS. Panels report single- and multi-mechanism Coding tasks, GUI tasks, and all 100 tasks. Each OS group gives Success Rate (SR, %), API cost (USD/run), elapsed time (seconds/run), and total input-plus-output tokens (Tok., thousands/run). Cross-environment columns give the three-OS arithmetic mean of SR (Avg. SR, %), Robust Success Rate (RSR, %), and Environment Gap (EG, percentage points). Each panel weights its mechanism families equally. Bold indicates the best outcome score within each panel, including ties.

	Linux				macOS				Windows				Cross-environment		
Model	SR ↑	Cost ↓	Time ↓	Tok. ↓	SR ↑	Cost ↓	Time ↓	Tok. ↓	SR ↑	Cost ↓	Time ↓	Tok. ↓	Avg. SR ↑	RSR ↑	EG ↓
Coding: single-mechanism tasks (52 tasks, 14 mechanism families)															
Deepseek-v4-flash	78.6	0.033	61.0	88.1	76.2	0.018	25.3	45.7	53.3	0.042	40.7	114.8	69.4	20.1	79.9
GLM-5.3-Flash	73.2	0.022	475.6	49.1	63.1	0.029	460.8	54.0	63.3	0.033	383.8	68.7	66.5	21.4	75.0
Gemini-3.8-Flash	77.3	0.253	478.8	457.8	70.8	0.244	504.8	420.0	58.7	0.302	285.0	493.1	68.9	28.1	70.2
Grok-4-7	82.1	0.037	75.1	27.0	81.2	0.123	135.0	129.1	42.9	0.070	73.7	66.3	68.7	11.0	89.0
GPT-5.6-Sol	78.6	0.066	39.0	13.7	81.6	0.075	41.9	13.3	56.5	0.069	57.2	14.9	72.2	28.6	71.4
Claude-Opus-5	**87.2**	0.688	186.9	83.0	**93.7**	0.515	137.7	63.0	**82.7**	0.558	155.8	74.9	**87.9**	**75.1**	**24.9**
Coding: multi-mechanism tasks (28 tasks, 14 mechanism families)															
Deepseek-v4-flash	44.3	0.083	87.2	226.0	**87.9**	0.064	71.7	170.4	21.4	0.125	103.5	352.4	51.2	14.3	77.1
GLM-5.3-Flash	40.7	0.008	1050.9	80.5	67.6	0.013	1516.4	113.1	21.4	0.018	1311.1	152.3	43.2	3.6	79.7
Gemini-3.8-Flash	35.7	0.358	454.1	1086.3	78.6	0.359	458.2	1106.1	35.3	0.360	482.3	1132.6	49.9	14.3	78.6
Grok-4-7	50.0	0.186	283.7	107.0	70.0	0.255	305.2	161.7	3.6	0.165	179.3	102.4	41.2	3.6	66.4
GPT-5.6-Sol	40.7	0.044	64.3	17.2	75.0	0.047	71.3	19.8	7.1	0.051	81.0	21.3	41.0	0.0	89.3
Claude-Opus-5	**68.1**	0.171	55.5	23.9	72.7	0.196	59.3	28.8	**51.2**	0.354	110.6	57.2	**64.0**	**43.1**	**36.7**
GUI tasks (20 tasks, 2 mechanism families)															
Deepseek-v4-flash	**100.0**	0.012	61.3	23.5	11.9	0.038	207.9	58.8	17.9	0.041	168.9	59.2	43.3	0.0	100.0
GLM-5.3-Flash	**100.0**	0.007	175.9	33.9	35.7	0.015	759.2	59.6	58.3	0.019	1350.6	67.0	64.7	27.4	72.6
Gemini-3.8-Flash	92.9	0.037	378.2	33.5	48.8	0.052	564.0	47.7	**85.7**	0.031	416.8	26.1	**75.8**	**48.8**	**47.6**
Grok-4-7	89.3	0.156	224.0	74.1	23.8	0.374	786.4	133.7	25.0	0.387	752.8	133.1	46.0	8.3	81.0
GPT-5.6-Sol	**100.0**	0.130	61.7	24.4	39.3	0.141	101.9	24.4	**85.7**	0.151	91.8	26.2	75.0	39.3	60.7
Claude-Opus-5	96.4	0.187	84.5	33.8	**53.6**	0.503	246.5	71.3	45.2	0.457	181.9	67.0	65.1	41.7	58.3
All tasks (100 tasks, 30 mechanism families)															
Deepseek-v4-flash	64.0	0.055	73.3	148.1	77.3	0.041	59.1	104.7	36.1	0.080	78.6	222.0	59.1	16.0	80.0
GLM-5.3-Flash	59.8	0.014	724.1	62.7	63.3	0.021	973.3	82.0	43.4	0.025	881.0	107.6	55.5	13.5	77.0
Gemini-3.8-Flash	58.9	0.288	460.6	722.8	73.0	0.285	487.0	715.4	49.6	0.311	385.9	760.4	60.5	23.0	72.6
Grok-4-7	67.6	0.115	182.3	67.4	72.1	0.201	257.8	144.6	23.3	0.136	168.3	87.6	54.4	7.3	78.0
GPT-5.6-Sol	62.3	0.060	52.3	16.0	75.7	0.066	59.6	17.1	35.4	0.066	70.6	18.6	57.8	16.0	79.0
Claude-Opus-5	**78.9**	0.413	118.7	52.1	**81.2**	0.365	108.4	47.6	**65.5**	0.456	136.5	66.1	**75.2**	**57.9**	**32.6**

Robust successes only partially overlap across models. In unweighted counts, Claude-Opus-5 completes 56 of 100 tasks on all three deployments, while the other models complete 8–21 each. The union across models contains 67 tasks, comprising 57 Coding and 10 GUI tasks, leaving 33 tasks without a model that succeeds on every deployment. Only two tasks pass under every model on all three OSs. The union includes 11 tasks outside Claude-Opus-5's successful set, showing complementary successes despite its overall lead. Appendices A and E provide rankings, complete results, and weighting analyses.

4.2 Mechanism-Level Analysis of Cross-Environment Robustness

Using the same 1,800 outcomes, we examine whether the 30 mechanism families expose structure beneath aggregate scores, predict differences on held-out tasks, and exhibit recurring directions across models.

Aggregate OS rankings conceal family-specific reversals. The aggregate ordering macOS $>$ Linux $>$ Windows holds for all six models, but every model exhibits family-specific reversals for each OS pair (Figure 3). These reversals can be substantial even within the Coding interface. For GPT-5.6-Sol, Linux exceeds Windows by 100 percentage points on trailing-character path normalization, whereas Windows exceeds Linux by the same margin on digest case normalization. Reversals also persist in the strongest model: Claude-Opus-5 achieves 75 percentage points higher SR on Linux than on Windows for path-object identity, but 50 percentage points higher SR on Windows

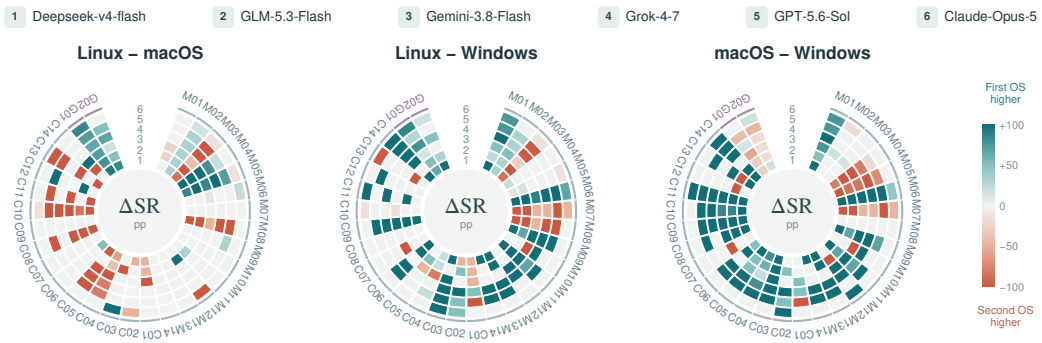


Figure 3: **Signed environmental differences by mechanism family.** Each sector denotes a family and each numbered ring a model, ordered from innermost to outermost according to the key. Teal favors the first OS in the panel title; coral favors the second. All panels share a scale in percentage points. M01–M14 and C01–C14 denote single- and multi-mechanism Coding families; G01–G02 denote GUI families. Appendix B gives the mapping and underlying SR values.

than on Linux for serialized path separators. Thus, the aggregate ranking summarizes opposing family-level differences, and an advantage on a deployment does not consistently extend across execution requirements.

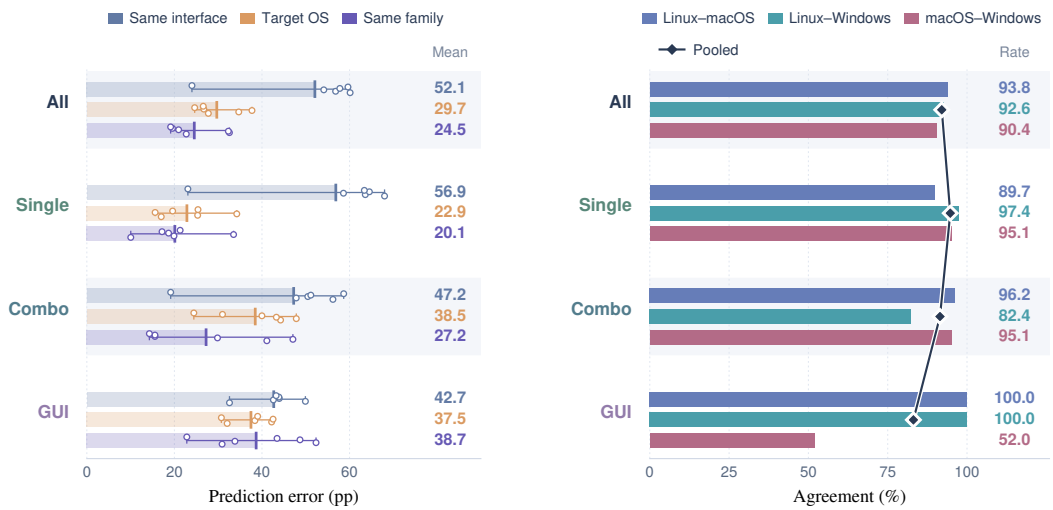
Family membership improves aggregate prediction of held-out differences. For each model and OS pair, we predict a held-out task's ΔSR from the mean difference of the other tasks in its family. Two baselines use other tasks with the same interaction interface, either all such tasks or those sharing the same construction-time target OS set. All predictors exclude the held-out task and use the same evaluation set of 88 tasks in 18 families. Mean absolute error (MAE) weights eligible families, OS pairs, and models equally.

The family predictor achieves 24.5 percentage points of MAE, compared with 52.1 for the same-interface baseline and 29.7 for the target OS baseline (Figure 4a). It improves on both baselines for every model in the pooled comparison; the reduction relative to target OS grouping is 5.1 points (17.3%). The benefit is concentrated in Coding, where family prediction reduces error from 28.7 to 22.8 points. The reduction is larger for combined-mechanism tasks (38.5 to 27.2 points) than for single-mechanism tasks (22.9 to 20.1 points). For GUI, family prediction yields 38.7 points versus 37.5 for target OS grouping. Family membership therefore adds aggregate predictive information, with different gains across the two interfaces. Appendix B reports the predictor definitions and disaggregated comparisons.

Family-level difference directions recur across models. For each family and OS pair, we compare the signs of the six models' ΔSR values. Agreement is 93.8% for Linux–macOS, 92.6% for Linux–Windows, and 90.4% for macOS–Windows. Across OS pairs, it is 94.7% for single-mechanism Coding, 91.5% for combined-mechanism Coding, and 83.1% for GUI. GUI agreement reaches 100% for both comparisons involving Linux, but falls to 52.0% (13/25) for macOS–Windows. These recurring directions, together with held-out prediction, connect mechanism families to environmental response patterns shared by models with different overall completion rates.

5 Related Work

Task benchmarks and environmental robustness. Execution-based benchmarks evaluate repository patches (SWE-bench (Jimenez et al., 2024)), scientific programs and resource use (ScienceAgentBench (Chen et al., 2025)), terminal workflows (Terminal-Bench (Merrill et al., 2026)), and desktop task completion (OSWorld (Xie et al., 2024)) through tests or task-state checks. Environmental robustness studies vary the conditions under which models perform these tasks: ReCode applies semantics-preserving perturbations to code-generation inputs (Wang et al., 2023), OSWorld migrates a subset of tasks from Ubuntu to Windows, and TransBench evaluates GUI grounding



(a) Held-out prediction **(b) Cross-model agreement**

Figure 4: **Predictive value and cross-model recurrence.** Groups: All tasks, single-mechanism Coding (Single), combined-mechanism Coding (Combo), and GUI. **(a)** Leave-one-task-out MAE (pp; lower is better) on 88 tasks. Bars average six models; circles and whiskers show individual values and ranges. Families and OS pairs receive equal weight. **(b)** Diamonds pool comparison counts across OS pairs within each group. See Appendix B for definitions and counts.

across versions, platforms, and applications (Lu et al., 2025). ChameleonBench evaluates 80 Coding and 20 GUI tasks across native Linux, macOS, and Windows deployments under a fixed harness, preserving task objectives and success criteria while exposing models to native filesystem, runtime, and application behavior. Independent program and artifact checks support paired completion comparisons across all tasks; Section 2.4 details the differences in environmental variation and task correspondence.

Behavioral coverage and composition. CheckList organizes behavioral tests by linguistic capability and test type (Ribeiro et al., 2020), while CFQ varies composition distributions while keeping primitive distributions similar (Keysers et al., 2019). COGS evaluates novel combinations of familiar words and structures (Kim & Linzen, 2020); SLOG broadens structural generalization coverage (Li et al., 2023). WorkArena++ evaluates agent planning and reasoning in enterprise workflows (Boisvert et al., 2024). ChameleonBench defines 28 Coding families from sets of 14 elementary mechanisms and two GUI families covering editor save conventions and file-manager extension visibility. These mutually exclusive families distinguish individual requirements from combinations, supporting analyses of OS differences, within-family held-out prediction, and recurring difference directions across models.

6 Conclusion

We introduce ChameleonBench, a benchmark of 100 Coding and GUI tasks paired across native Linux, macOS, and Windows deployments, with 30 mechanism families, a shared interaction harness, and independent outcome verification. Evaluation of six models reveals a substantial gap between deployment-level success and completion of the same tasks across environments: the highest Robust Success Rate is 57.9%, while the other models achieve 7.3–23.0%. Coding and GUI exhibit different deployment preferences, and mechanism families reveal local ranking reversals, improve aggregate prediction of held-out differences, and capture directions that recur across models. These findings motivate evaluating task completion jointly with its preservation under changing execution conditions. ChameleonBench provides a basis for studying how models adapt programs and desktop actions to environmental variation, with further evaluation extending to software versions, configurations, and interaction harnesses.

AI USE STATEMENT

We used generative AI tools for:

- Generating synthetic benchmark task data.

We have not used generative AI tools for:

- Helping develop theoretical models or conceptual frameworks.
- Proposing or refining hypotheses.
- Designing or providing feedback on research methodology or experiments.
- Implementing methods.
- Supporting qualitative and thematic data analysis.
- Interpreting results.

The following tasks are not applicable to this work:

- Formulating mathematical claims.
- Providing critical ingredients for proving mathematical claims.
- Assisting in the writing of proofs.
- Assisting with translation.
- Cleaning and reformatting datasets.

Additionally, we used generative AI tools for:

- Creating and editing scripts for aggregating existing experiment records.
- Discovering research topics or identifying gaps.
- Editing the manuscript to improve clarity and readability.

We have reviewed all AI-assisted work. Generated data were checked for format validity, generated code was tested by the authors, and references suggested by AI tools were checked for accuracy. For manuscript editing, AI tools provided revision suggestions, while the authors retained responsibility for the writing. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the assistance of generative AI.

REPRODUCIBILITY STATEMENT

The appendices provide model configurations, outcome counts, resource measurements, family scores, aggregation sensitivity, and mechanism-level analyses. The generated records include source hashes, the training pools used for every held-out prediction, and the construction and execution evidence behind the appendix examples.

REFERENCES

Léo Boisvert, Megh Thakkar, Maxime Gasse, Massimo Caccia, Thibault L De Chezelles, Quentin Cappart, Nicolas Chapados, Alexandre Lacoste, and Alexandre Drouin. Workarena++: Towards compositional planning and reasoning-based common knowledge work tasks. *Advances in Neural Information Processing Systems*, 37:5996–6051, 2024.

Ziru Chen, Shijie Chen, Yuting Ning, Qianheng Zhang, Boshi Wang, Botao Yu, Yifei Li, Zeyi Liao, Chen Wei, Zitong Lu, et al. Scienceagentbench: Toward rigorous assessment of language agents for data-driven scientific discovery. In *International Conference on Learning Representations*, volume 2025, pp. 96934–96990, 2025.

Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H Laradji, Manuel Del Verme, Tom Marty, Léo Boisvert, Megh Thakkar, Quentin Cappart, David Vazquez, et al. Workarena: How capable are web agents at solving common knowledge work tasks? *arXiv preprint arXiv:2403.07718*, 2024.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024.

Daniel Keysers, Nathanael Schärli, Nathan Scales, Hylke Buisman, Daniel Furrer, Sergii Kashubin, Nikola Momchev, Danila Sinopalnikov, Lukasz Stafiniak, Tibor Tihon, et al. Measuring compositional generalization: A comprehensive method on realistic data. *arXiv preprint arXiv:1912.09713*, 2019.

Najoung Kim and Tal Linzen. Cogs: A compositional generalization challenge based on semantic interpretation. In *Proceedings of the 2020 conference on empirical methods in natural language processing (emnlp)*, pp. 9087–9105, 2020.

Bingzhi Li, Lucia Donatelli, Alexander Koller, Tal Linzen, Yuekun Yao, and Najoung Kim. Slog: A structural generalization benchmark for semantic parsing. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 3213–3232, 2023.

Yuheng Lu, Qian Yu, Hongru Wang, Zeming Liu, Wei Su, Yanping Liu, Yuhang Guo, Maocheng Liang, Yunhong Wang, and Haifeng Wang. Transbench: Breaking barriers for transferable graphical user interface agents in dynamic digital environments. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 12464–12478, 2025.

Mike Merrill, Alexander Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. 2026:40903–40986, 2026.

Victor Muryn, Maksym Shamrai, Sofiia Mazepa, and Yehor Khodysko. Macarena: Benchmarking computer use agents on an online macos environment. *arXiv preprint arXiv:2606.06560*, 2026.

Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. Beyond accuracy: Behavioral testing of nlp models with checklist. In *Proceedings of the 58th annual meeting of the association for computational linguistics*, pp. 4902–4912, 2020.

Jingwei Sun, Jianing Zhu, Yuanyi Li, Tongliang Liu, Xia Hu, and Bo Han. Agenthijack: Benchmarking computer use agent robustness to common environment corruptions. *arXiv preprint arXiv:2605.25707*, 2026a.

Yiyou Sun, Xinyang Han, Weichen Zhang, Yuanbo Pang, Tianyu Wang, Yuhan Cao, Yixiao Huang, Chris Duroiu, Haoyun Zhang, Jeffrey Lin, et al. Agents' last exam. *arXiv preprint arXiv:2606.05405*, 2026b.

Karen Ullrich, Jingtong Su, Claudia Shi, Arjun Subramonian, Amir Bar, Ivan Evtimov, Nikolaos Tsilivis, Randall Balestriero, Julia Kempe, and Mark Ibrahim. Openapps: Simulating environment variations to measure ui agent reliability. In *International Conference on Learning Representations*, volume 2026, pp. 118257–118290, 2026.

Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, et al. Recode: Robustness evaluation of code generation models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13818–13843, 2023.

Xuehui Wang, Zhenyu Wu, JingJing Xie, Zichen Ding, Bowen Yang, Zehao Li, Zhaoyang Liu, Qingyun Li, Xuan Dong, Zhe Chen, et al. Mmbench-gui: Hierarchical multi-platform evaluation framework for gui agents. *arXiv preprint arXiv:2507.19478*, 2025.

Zhiyong Wu, Chengcheng Han, Zichen Ding, Zhenmin Weng, Zhoumianze Liu, Shunyu Yao, Tao Yu, and Lingpeng Kong. Os-copilot: Towards generalist computer agents with self-improvement, 2024. *URL https://arxiv.org/abs/2402.07456*, 2.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

Tianqi Xu, Linyao Chen, Dai-Jie Wu, Yanjun Chen, Zecheng Zhang, Xiang Yao, Zhiqiang Xie, Yongchao Chen, Shilong Liu, Bochen Qian, et al. Crab: Cross-environment agent benchmark for multimodal language model agents. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 21607–21647, 2025.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. 2024.

YiShan Zheng, Yuan Wu, and Yi Chang. Toolrobustbench: Stage-wise perturbation evaluation and failure diagnosis for tool-calling agents. *arXiv preprint arXiv:2608.23635*, 2026.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, volume 2024, pp. 15585–15606, 2024.

A Model Rankings Across Deployments and Evaluation Criteria

Figure 5 compares rankings by deployment-specific SR and RSR, using the six-model results in Table 2. All four comparisons use the same 30 family weights across 80 Coding and 20 GUI tasks. Rankings use unrounded scores, with exact ties sharing a rank.

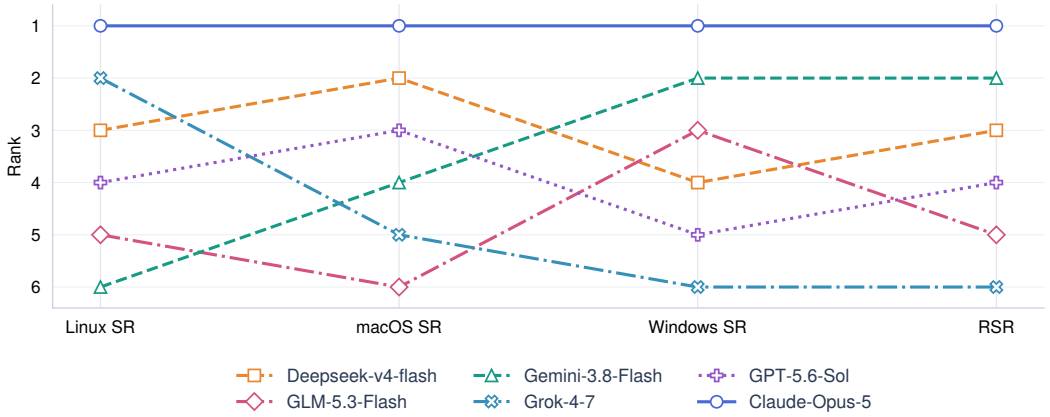


Figure 5: **Model rankings across deployments and paired robustness.** Rank 1 denotes the highest score. Rankings use unrounded family-weighted Linux SR, macOS SR, Windows SR, and RSR on all 100 tasks. Scores appear in Table 2.

Claude-Opus-5 ranks first in all four comparisons. The ordering of the other models varies with the deployment and criterion. Grok-4-7 moves from second on Linux to sixth on Windows and by RSR. Gemini-3.8-Flash ranks sixth on Linux and fourth on macOS, and second on Windows and by RSR. Deepseek-v4-flash ranks second on macOS and third by RSR, while GPT-5.6-Sol moves from third on macOS to fourth by RSR. Thus, rankings based on one deployment do not consistently preserve the ordering by paired cross-environment completion.

B Detailed Mechanism-Level Results

This appendix provides the family mapping, deployment-level scores, held-out predictions, and direction comparisons underlying Section 4.2. All analyses use the same 100 tasks and 1,800 outcomes as the main table. Each task belongs to exactly one of 30 families. Coding families are defined by complete sets of elementary mechanisms: M01–M14 identify single-mechanism families, and C01–C14 identify the 14 observed combinations. G01 and G02 identify the GUI families for native editor save conventions and file-manager extension visibility. Table 3 specifies their membership.

Table 3: **Families used in the statistical analyses.** M identifiers follow the elementary mechanisms in Table 9; C identifiers list complete Coding mechanism sets; G identifiers denote the two GUI workflow families. Membership is mutually exclusive and totals 100 tasks.

ID	Mechanism set or GUI workflow	Tasks	Task identifiers
M01	Path-object identity	4	H28, H33, H43, H44
M02	Case-distinct name preservation	6	C240, C241, P440, P441, P448, P454
M03	Digest case normalization	7	H330, H331, H332, H333, H337, H342, H343
M04	Unicode filename case equivalence	5	K260, K263, K266, K267, K268
M05	Unicode filename normalization	7	J270, N230, N233, N236, N237, N238, N239
M06	Trailing-character path normalization	9	R220, R221, R222, R223, R224, R225, W300, W301, W302
M07	Serialized path separators	2	P438, S290

Continued on the next page

Table 3 (continued)

ID	Mechanism set or GUI workflow	Tasks	Task identifiers
M08	Filename byte-length limits	2	P420, P422
M09	Reserved device names	3	D350, D351, D352
M10	Wildcard case matching	1	G370
M11	Read-only file deletion	3	P442, P443, P444
M12	Archive member-name sanitization	1	P451
M13	Default file-text decoding	1	E311
M14	Standard-input text decoding	1	P453
C01	M05 + M07 + M08	2	P467, P470
C02	M05 + M07 + M11	2	P462, P472
C03	M05 + M07 + M14	1	P473
C04	M05 + M08 + M11	5	P455, P460, P468, P478, P481
C05	M05 + M08 + M14	2	P463, P476
C06	M05 + M11 + M14	2	P465, P480
C07	M07 + M08 + M11	1	P464
C08	M07 + M08 + M14	1	P461
C09	M07 + M11 + M14	1	P469
C10	M08 + M11 + M14	7	P466, P471, P474, P475, P477, P479, P482
C11	M05 + M08 + M11 + M13	1	P458
C12	M05 + M08 + M13 + M14	1	P456
C13	M07 + M08 + M10 + M11	1	P457
C14	M07 + M11 + M13 + M14	1	P459
G01	Native editor save conventions	14	G33, G34, G36, G37, G38, G39, G40, G41, G42, G47, G48, G49, G52, G57
G02	File-manager extension visibility	6	G45, G53, G54, G61, G62, G63

B.1 Benchmark Taxonomy and Model Profiles

Figure 6 summarizes model performance across all 30 mechanism families. Each value averages binary outcomes equally over the three deployments and over tasks within the corresponding family. The upper-left radar contains the 14 single-mechanism Coding families, representing 52 tasks; the upper-right radar contains 14 combined-mechanism families, representing 28 tasks. Two horizontal point plots below them show the GUI families: native editor save conventions with 14 tasks, and file-manager extension visibility with six tasks. All four profiles use a 0–100% scale. The keys give each Coding family's mechanism or complete mechanism set and task count.

Claude-Opus-5 has the highest or joint-highest mean SR in 24 of the 30 families, including 23 of the 28 Coding families. GPT-5.6-Sol leads the editor family at 66.7%, while Gemini-3.8-Flash and Claude-Opus-5 share the highest mean SR in the file-manager family at 94.4%. The profiles also show variation within each model, including lower completion across many combined-mechanism families. These per-family averages connect the benchmark's composition to model performance, while Figure 7 separates the deployments and Figure 3 shows their signed differences.

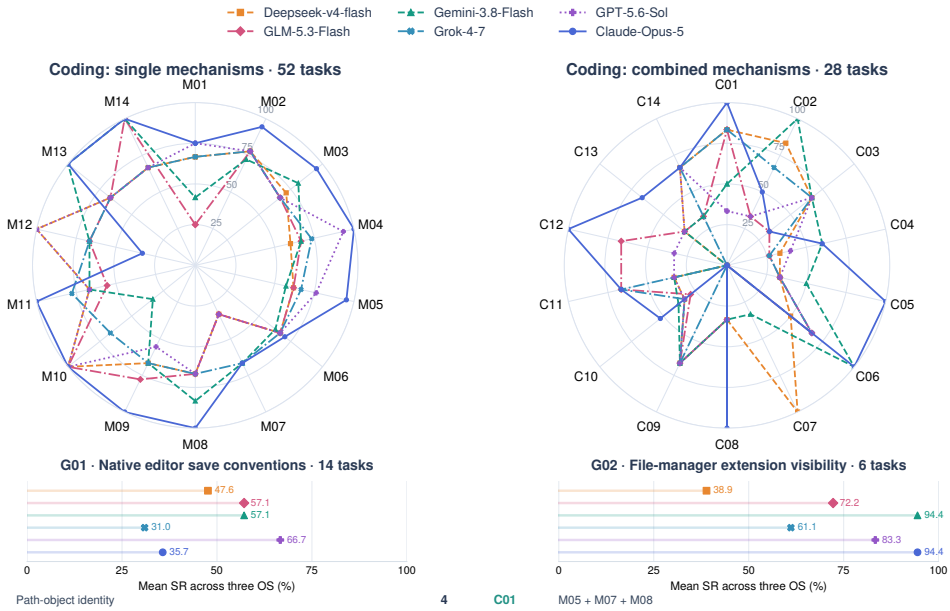
B.2 Signed Differences and Within-Family Scores

For model m, task t, and deployments a and b, define

$$\Delta\mathrm{SR}^{a,b}_{m,t} = 100\big(p_{m,t,a} - p_{m,t,b}\big), \qquad \Delta\mathrm{SR}^{a,b}_{m,f} = \frac{1}{|T_f|}\sum_{t\in T_f} \Delta\mathrm{SR}^{a,b}_{m,t}, \tag{3}$$

where $p_{m,t,o} \in \{0,1\}$ is the observed outcome and T_f contains the tasks in family f. Comparison order is fixed to Linux–macOS, Linux–Windows, and macOS–Windows. A positive difference favors the first deployment. Figure 7 gives the SR values from which the family differences are calculated.

Among the 540 model–family–OS-pair differences, 320 are nonzero. For Linux–macOS, 36 of 85 nonzero differences reverse the corresponding model's overall SR ordering; the corresponding



M01	Path-object identity	4	C01	M05 + M07 + M08	2
M02	Case-distinct name preservation	6	C02	M05 + M07 + M11	2
M03	Digest case normalization	7	C03	M05 + M07 + M14	1
M04	Unicode filename case equivalence	5	C04	M05 + M08 + M11	5
M05	Unicode filename normalization	7	C05	M05 + M08 + M14	2
M06	Trailing-character path normalization	9	C06	M05 + M11 + M14	2
M07	Serialized path separators	2	C07	M07 + M08 + M11	1
M08	Filename byte-length limits	2	C08	M07 + M08 + M14	1
M09	Reserved device names	3	C09	M07 + M11 + M14	1
M10	Wildcard case matching	1	C10	M08 + M11 + M14	7
M11	Read-only file deletion	3	C11	M05 + M08 + M11 + M13	1
M12	Archive member-name sanitization	1	C12	M05 + M08 + M13 + M14	1
M13	Default file-text decoding	1	C13	M07 + M08 + M10 + M11	1
M14	Standard-input text decoding	1	C14	M07 + M11 + M13 + M14	1

Figure 6: **Model performance and benchmark composition by mechanism family.** Radar spokes and horizontal points report mean SR (%) across Linux, macOS, and Windows within one family. Larger radii or positions farther right indicate higher completion. The radars distinguish single and combined Coding mechanisms; the point plots show the two GUI families. Identifiers follow Table 3. All values use the 1,800 outcomes of the main comparison.

counts are 32 of 113 for Linux–Windows and 31 of 122 for macOS–Windows. These reversals occur in 13, 12, and 12 families, respectively. The resulting 99 reversals occur for every model on each OS pair. Using each family's interface-specific aggregate ordering as the comparison yields 82 reversals.

B.3 Leave-One-Task-Out Prediction

For a fixed model and OS pair, each predictor averages the observed differences over a training pool that excludes the task being predicted and uses the same interaction interface. The *interface* pool contains every other Coding task for a Coding prediction and every other GUI task for a GUI prediction. The *target OS* pool further selects tasks whose construction-time target OS set is exactly equal to that of the held-out task. The *same-family* pool contains the other tasks in its Coding mechanism family or GUI workflow family. The first two predictors can draw on singleton-family tasks when they meet the corresponding training criterion.

The common evaluation set T^+ contains 88 tasks in 18 families with at least two tasks; every task also has a nonempty target OS pool. It comprises 48 tasks in ten single-mechanism Coding families, 20 tasks in six combined-mechanism Coding families, and all 20 GUI tasks in two families. The remaining 12 singleton Coding families enter the main metrics and the direction analysis. For predictor h with training pool $U_h(t)$, we compute

$$\widehat{\Delta}^{a,b}_{m,t,h} = \frac{1}{|U_h(t)|} \sum_{u \in U_h(t)} \Delta\mathrm{SR}^{a,b}_{m,u}, \qquad \mathrm{MAE}^{a,b}_{m,h} = \frac{1}{|G^+|} \sum_{f \in G^+} \frac{1}{|T_f|} \sum_{t \in T_f} \left| \widehat{\Delta}^{a,b}_{m,t,h} - \Delta\mathrm{SR}^{a,b}_{m,t} \right|, \tag{4}$$

where G^+ is the set of eligible families. Each model's summary averages MAE equally over the three OS pairs; Figure 4(a) further averages over models. Table 4 reports these model-level errors, and Table 5 separates the OS pairs. Across models, mean MAE is 52.1 points for the interface predictor, 29.7 for target OS, and 24.5 for the family predictor. The family predictor has lower overall error than target OS for all six models.

Table 4: **Predicting held-out task differences.** MAE in percentage points on the same 88 tasks and 18 families. All training pools use the held-out task's interface. Families and OS pairs receive equal weight; Mean also weights models equally. Bold marks the lowest error in each row.

Model	Interface	Target OS	Same family
Deepseek-v4-flash	60.1	27.8	**22.7**
GLM-5.3-Flash	56.8	34.7	**32.6**
Gemini-3.8-Flash	54.1	37.7	**32.3**
Grok-4-7	57.8	26.7	**21.0**
GPT-5.6-Sol	59.6	24.6	**19.6**
Claude-Opus-5	24.0	26.6	**19.1**
Mean	52.1	29.7	**24.5**

Interfaces and Coding composition. Table 6 evaluates the same predictions within each subset, renormalizing family weights to sum to one. For Coding overall, the mean MAE is 53.3 points for the interface predictor, 28.7 for target OS, and 22.8 for the family predictor. For single-mechanism tasks, the mean MAE is 20.1 points for the family predictor, compared with 22.9 for target OS and 56.9 for the interface predictor. The family predictor improves on target OS for five of six models; Grok-4-7 has errors of 18.7 and 15.6 points, respectively. For multi-mechanism tasks, mean errors are 27.2, 38.5, and 47.2 points, respectively, and all six models improve over target OS. All six eligible combined families contain three mechanisms. For GUI, target OS has the lowest mean MAE at 37.5 points, followed by the family predictor at 38.7 and the interface predictor at 42.7. The family predictor improves on target OS for three of the six models in this subset.

B.4 Cross-Model Directional Agreement

For each family and OS pair, we compare all 15 unordered model pairs using unrounded differences. A comparison is *same* when both differences are nonzero and share a sign, *opposite* when their

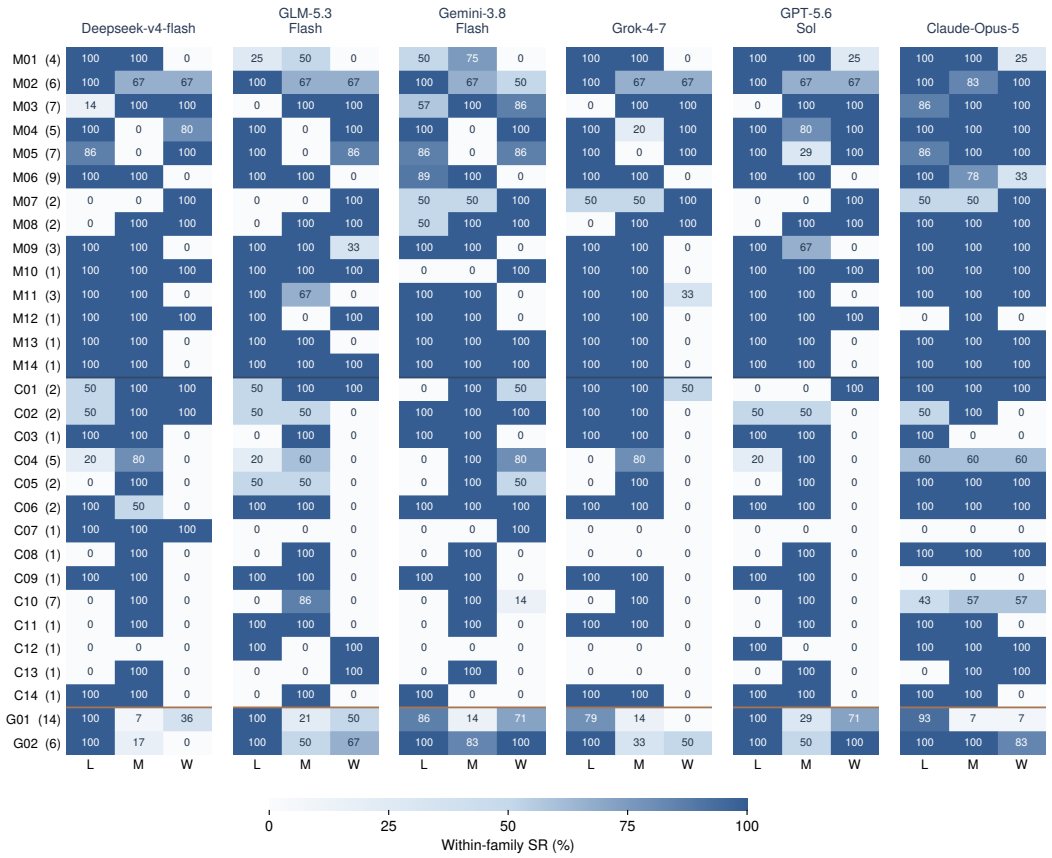


Figure 7: **Success rates by model, family, and deployment.** L, M, and W denote Linux, macOS, and Windows. Values are percentages rounded to integers for display; calculations use unrounded scores. Parentheses give family sizes. Horizontal rules separate single-mechanism Coding, combined-mechanism Coding, and GUI families, with identifiers defined in Table 3.

Table 5: **Leave-one-task-out MAE by OS pair.** Errors are in percentage points, with equal weights across eligible families and tasks within a family. Bold marks the lowest error per row.

Model	OS pair	Interface	Target OS	Same family
Deepseek-v4-flash	Linux–macOS	52.1	**26.9**	28.8
	Linux–Windows	64.1	28.0	**24.4**
	macOS–Windows	64.2	28.4	**14.9**
GLM-5.3-Flash	Linux–macOS	47.6	**24.7**	26.9
	Linux–Windows	61.3	36.8	**35.9**
	macOS–Windows	61.5	42.6	**34.9**
Gemini-3.8-Flash	Linux–macOS	52.2	37.1	**21.6**
	Linux–Windows	51.2	**42.3**	42.3
	macOS–Windows	58.9	33.8	**33.1**
Grok-4-7	Linux–macOS	48.4	19.2	**13.1**
	Linux–Windows	60.7	29.4	**23.1**
	macOS–Windows	64.3	31.6	**26.7**
GPT-5.6-Sol	Linux–macOS	48.1	24.7	**19.5**
	Linux–Windows	63.7	20.8	**16.0**
	macOS–Windows	67.1	28.3	**23.2**
Claude-Opus-5	Linux–macOS	**15.8**	19.1	15.9
	Linux–Windows	28.7	31.3	**24.7**
	macOS–Windows	27.6	29.3	**16.7**

Table 6: **Prediction errors by interface and Coding composition.** Coding contains 68 eligible tasks in 16 families: Single has 48 tasks in ten families and Multiple has 20 tasks in six families. GUI contains 20 tasks in two families. MAE is in percentage points and averages eligible families and OS pairs equally within each subset. Bold marks the lowest error per row.

Model	Subset	Interface	Target OS	Same family
Deepseek-v4-flash	Coding	63.6	27.2	**21.7**
	Single	68.0	17.0	**10.0**
	Multiple	56.2	44.2	**41.1**
	GUI	32.6	32.1	**30.9**
GLM-5.3-Flash	Coding	57.7	33.8	**30.1**
	Single	63.6	25.3	**19.9**
	Multiple	47.8	47.8	**47.1**
	GUI	49.9	**42.2**	52.3
Gemini-3.8-Flash	Coding	55.6	37.6	**32.1**
	Single	58.6	34.2	**33.5**
	Multiple	50.5	43.3	**29.8**
	GUI	42.6	38.4	**33.8**
Grok-4-7	Coding	59.5	24.7	**17.5**
	Single	64.5	**15.6**	18.7
	Multiple	51.2	40.0	**15.6**
	GUI	43.9	**42.5**	48.7
GPT-5.6-Sol	Coding	61.6	23.9	**16.6**
	Single	63.4	19.6	**17.2**
	Multiple	58.7	31.0	**15.6**
	GUI	43.9	**30.7**	43.4
Claude-Opus-5	Coding	21.6	25.0	**18.7**
	Single	23.1	25.4	**21.3**
	Multiple	19.1	24.4	**14.3**
	GUI	43.2	39.0	**22.8**

signs differ, and *zero* when at least one difference is zero. Directional agreement is the number of same-direction comparisons divided by the sum of same- and opposite-direction comparisons. Pooling the 30 families and three OS pairs yields 1,350 comparisons: 532 same, 46 opposite, and 772 involving a zero. The agreement rate is therefore $532/578 = 92.0\%$. Coding contributes 1,260 comparisons and has agreement $473/507 = 93.3\%$; GUI contributes 90 comparisons and has agreement $59/71 = 83.1\%$.

Table 7 separates OS pairs, interfaces, and Coding composition. Each OS pair contributes 450 candidate comparisons overall: 210 each from single- and combined-mechanism Coding families, and 30 from GUI families. Table 8 retains the family-specific counts. When one model is removed and all comparisons among the other five are recomputed, overall agreement ranges from 90.1% to 93.5%.

C Task Construction and Validation

C.1 Task Contracts and Native Realizations

The collection contains 80 Coding and 20 GUI tasks. Each task specifies a functional objective, initial resources, permitted operations, and completion constraints. Its Linux, macOS, and Windows realizations preserve input semantics, required outputs, and business invariants. Local resource paths bind the contract to each deployment, whose filesystem, runtime, and applications retain their native behavior. Coding tasks let the model inspect files, edit code, and execute commands. GUI tasks begin in a designated native application and expose screenshots and mouse and keyboard actions.

The model receives the instruction and initialized project or application state. Reference programs and procedures, independent grading inputs, and mechanism annotations are evaluator resources. The following examples detail the implementation and verification of single- and multi-mechanism Coding tasks, native editor saving, and file-manager renaming.

Table 7: **Cross-model direction counts.** Zero includes comparisons with at least one zero difference. Agreement is Same / (Same + Opposite), in percent. The subsets contain 30 families overall, 28 Coding families, 14 each for Single and Multiple, and two GUI families. Within each subset, all pools the three OS pairs.

Subset	OS pair	Same	Opposite	Zero	Agreement (%)
All	Linux–macOS	137	9	304	93.8
	Linux–Windows	188	15	247	92.6
	macOS–Windows	207	22	221	90.4
	all	532	46	772	92.0
Coding	Linux–macOS	112	9	299	92.6
	Linux–Windows	167	15	238	91.8
	macOS–Windows	194	10	216	95.1
	all	473	34	753	93.3
Single	Linux–macOS	61	7	142	89.7
	Linux–Windows	111	3	96	97.4
	macOS–Windows	97	5	108	95.1
	all	269	15	346	94.7
Multiple	Linux–macOS	51	2	157	96.2
	Linux–Windows	56	12	142	82.4
	macOS–Windows	97	5	108	95.1
	all	204	19	407	91.5
GUI	Linux–macOS	25	0	5	100.0
	Linux–Windows	21	0	9	100.0
	macOS–Windows	13	12	5	52.0
	all	59	12	19	83.1

Single mechanism: record export (N230). The source is a JSON array containing unique Unicode keys and arbitrary JSON payloads. In `intakestore.py`, the model implements `export(root, src)` while preserving the names, arguments, and return shapes of `put`, `get`, and `load_source`. Export must store every record, return `written` and the keys in source order, and allow each payload to be retrieved under its original key. An empty source must return zero and an empty key list. The instruction also requests regression tests and restricts implementation to the Python standard library.

The grader creates an independent source and fresh store. It checks the return values, the number of stored records, retrieval under every original key, existing `put`/`get` behavior, and the empty-source case. The inputs include canonically equivalent Unicode spellings that remain distinct logical keys. Directly using these strings as filenames can merge their records under native filename normalization (M05). The reference implementation encodes each original key's UTF-8 bytes as hexadecimal for the storage filename, preserving both logical identities.

Multiple mechanisms: notes import (P455). The export bundle contains notes, attachments, and a path to the previous import. The model implements three functions in `notesimport.py`: `clear_previous` removes the previous directory and returns its file count; `write_note` writes a title-derived Markdown file and returns its name; and `save_attachment` stores an attachment for retrieval under its original name. The existing `load_export`, `read_attachment`, and `run_import` APIs must retain their signatures and return values. The format specification defines attachment identity by exact string equality, and the task requests regression tests for all three functions.

The independent fixture contains three previous files, including two read-only files; notes with short and long Unicode titles; and attachments with distinct NFC and NFD spellings. The grader checks the removal count and absence of the previous directory, reads the note bodies through the returned filenames, and compares every retrieved attachment with its source content. It also executes the integrated import and checks `load_export`. These requirements exercise read-only deletion (M11), filename byte-length limits (M08), and Unicode filename normalization (M05). The reference implementation makes files writable before removal, limits note stems to 200 UTF-8 bytes, and maps original attachment names to encoded storage names through `index.json`.

Table 8: **Direction counts within each family.** Entries give same-direction / nonzero comparisons, followed by the number involving a zero in parentheses. Each cell contains 15 candidate model pairs; a zero nonzero-count has no defined agreement rate.

Family	Linux–macOS	Linux–Windows	macOS–Windows
M01	1/1 (14)	15/15 (0)	15/15 (0)
M02	15/15 (0)	10/10 (5)	0/1 (14)
M03	15/15 (0)	15/15 (0)	0/0 (15)
M04	10/10 (5)	0/0 (15)	10/10 (5)
M05	10/15 (0)	1/3 (12)	10/10 (5)
M06	0/1 (14)	15/15 (0)	15/15 (0)
M07	0/0 (15)	15/15 (0)	15/15 (0)
M08	10/10 (5)	10/10 (5)	0/0 (15)
M09	0/0 (15)	10/10 (5)	10/10 (5)
M10	0/0 (15)	0/1 (14)	0/1 (14)
M11	0/0 (15)	10/10 (5)	10/10 (5)
M12	0/1 (14)	1/1 (14)	3/6 (9)
M13	0/0 (15)	6/6 (9)	6/6 (9)
M14	0/0 (15)	3/3 (12)	3/3 (12)
C01	3/3 (12)	6/10 (5)	1/3 (12)
C02	1/1 (14)	6/10 (5)	6/6 (9)
C03	0/1 (14)	10/10 (5)	10/10 (5)
C04	10/10 (5)	3/6 (9)	10/10 (5)
C05	6/6 (9)	0/1 (14)	10/10 (5)
C06	0/0 (15)	6/6 (9)	6/6 (9)
C07	0/0 (15)	0/0 (15)	0/0 (15)
C08	6/6 (9)	0/0 (15)	6/6 (9)
C09	0/0 (15)	10/10 (5)	10/10 (5)
C10	15/15 (0)	1/1 (14)	10/10 (5)
C11	3/3 (12)	3/3 (12)	15/15 (0)
C12	1/1 (14)	0/0 (15)	0/0 (15)
C13	6/6 (9)	1/1 (14)	3/6 (9)
C14	0/1 (14)	10/10 (5)	10/10 (5)
G01	15/15 (0)	15/15 (0)	6/10 (5)
G02	10/10 (5)	6/6 (9)	7/15 (0)

GUI: native editor save (G34). A text editor is already open: gedit on Linux, TextEdit on macOS, or Notepad on Windows. The model must save a plain-text file named exactly `orders.csv` in the specified folder, with the following content, and leave the editor open:

```
order_id,customer,total
1041,Atlas Inc,412.60
1042,Boreal Ltd,88.00
1043,Cedar Co,1290.25
```

Permitted interactions use the editor, its dialogs, and the system file manager through the provided mouse and keyboard actions. The native controller initializes the folder, launches the editor, and records the delivered filenames and bytes. The grader requires the exact filename, plain text without an RTF header or NUL bytes, the four expected CSV rows, and no additional files. It normalizes line endings and ignores empty CSV rows before comparing parsed values. Native save defaults, extension handling, and text substitutions can affect these checks.

GUI: exact rename (G61). A file manager is open to a directory containing `spec.txt` and `summary.txt`. The model must rename the former to exactly `spec.yaml` and leave the other file unchanged, using only the file manager and its dialogs. The grader inspects the final directory listing for the new name, removal of the old name, preservation of the `summary.txt` entry, and absence of additional entries.

C.2 Construction and Mechanism Organization

Task production. Authoring begins with an environment-dependent operation and a functional constraint that it must preserve. Coding packages supply the instruction, initial implementation, initialization resources, reference and naïve programs, and an independent verifier. GUI packages supply the instruction, application initialization, native reference procedure, and delivered-state grader. Native realizations align resource semantics, program interfaces, and application objectives across the three systems. Behavior probes and construction trials check the resulting operations and inform revisions before the task package and catalog entry are fixed.

Initial selection records cover 40 Coding tasks and all 20 GUI tasks, with five trials per OS and an inclusion criterion of at most one success on every designated target OS and at least four successes on every control OS. Their retained initial counts satisfy this criterion. The other 40 Coding tasks have native-probe and acceptance records, including workflows that combine several mechanisms. The subsequent six-model comparison uses the fixed 100-task collection, with one result per model–task–OS combination, yielding 1,800 outcomes.

Coding mechanisms and complete families. An elementary Coding mechanism is identified by the environmental rule, the affected operation, and the required business invariant. Annotations are grounded in the task contract, reference and naïve implementations, and verifier checks. For a Coding task's complete mechanism set $\mathcal{M}(t)$, family membership is defined by equality of that set. Every Coding task therefore has one family, while a multi-mechanism task retains each component annotation.

The 52 single-mechanism Coding tasks cover all 14 elementary mechanisms and form 14 families. The 28 multi-mechanism Coding tasks comprise 24 three-mechanism tasks and four four-mechanism tasks, forming another 14 families. Elementary-mechanism coverage in Table 9 includes overlapping memberships for the multi-mechanism tasks. Together with the two GUI families below, these groups form 30 mutually exclusive families; Table 3 provides the complete mapping of all 100 tasks.

Classification boundaries. The mapping preserves the following distinctions and merges:

- **Name identity and digest identity.** M02 combines case-distinct record storage and exact-name lookup because both must preserve differences in spelling. M03 requires equivalent hexadecimal digest spellings to retrieve the same object and therefore has a different invariant.

Table 9: **Coding elementary mechanisms and task coverage.** Single counts Coding tasks involving only the listed mechanism; All also includes multi-mechanism tasks containing it. Single sums to 52, while All permits overlapping membership. The Coding subset contains 80 tasks in 28 mutually exclusive mechanism families; the complete family mapping is provided in Appendix B.

ID	Elementary mechanism	Single	All	Completion requirement
M01	Path-object identity	4	4	Identify the intended objects and preserve access boundaries.
M02	Case-distinct name preservation	6	6	Preserve case distinctions in keys and exact-name matches.
M03	Digest case normalization	7	7	Resolve equivalent digest spellings to the same object.
M04	Unicode filename case equivalence	5	5	Keep Unicode keys distinct under native case rules.
M05	Unicode filename normalization	7	23	Preserve distinct keys and their records or event sequences.
M06	Trailing-character path normalization	9	9	Preserve trailing characters in name identity and lookup.
M07	Serialized path separators	2	12	Resolve serialized paths to their intended files.
M08	Filename byte-length limits	2	23	Create and retrieve files for long Unicode names.
M09	Reserved device names	3	3	Store and retrieve records with reserved-name keys.
M10	Wildcard case matching	1	2	Select only files satisfying the required case pattern.
M11	Read-only file deletion	3	24	Complete cleanup of files and directories.
M12	Archive member-name sanitization	1	1	Preserve every archive member and its contents.
M13	Default file-text decoding	1	4	Preserve source text when reading files.
M14	Standard-input text decoding	1	17	Preserve text received through standard input.

- **Unicode rules.** M04 concerns Unicode case equivalence; M05 concerns canonical filename normalization. M05 combines record overwriting and journal merging because both arise from normalization-equivalent names, despite their different write operations.
- **Trailing characters.** M06 combines path lookup and file creation affected by normalization of trailing dots or spaces.
- **Operation layer.** M01 concerns path-object equality, hashing, or ancestry; M02 concerns actual filenames and logical keys; M10 concerns wildcard selection; and M12 concerns archive-member rewriting during extraction.
- **Text entry points.** M13 covers file-text decoding and M14 covers standard-input decoding. P456 and P459 check both entry points and retain both annotations.

The canonical mapping retains the original source labels alongside the full mechanism set and the corresponding implementation and verifier locations.

GUI families. The 20 GUI tasks form two families organized by the native interaction governing the delivery (Table 10). The 14 editor tasks require exact filenames, plain-text formats, and specified content through native save dialogs. The six file-manager tasks require exact renaming while preserving the other directory entry; extension visibility and rename dialogs govern the displayed and edited names. Family membership follows the application operation and contract. The controller records the native application, platform, and delivered directory state.

Table 10: **GUI families and task coverage.** Each of the 20 GUI tasks belongs to one family. Full task membership appears in Table 3.

Family	Tasks	Completion requirement
Native editor save conventions	14	Save the required filename, plain-text format, and content.
File-manager extension visibility	6	Produce the exact target name and preserve the other directory entry.

C.3 Independent Scoring and Source Validation

The verifier in Equation 1 derives expected behavior from trusted task inputs. Coding checks execute the delivered program on independently initialized resources and inspect returned values, stored content and object identities, existing API behavior, and required final state. GUI checks inspect files and directory entries captured from the completed native workflow. Structured values are compared semantically; required names and text are checked against their specified values. Success requires every recorded check to pass, including all component checks of a multi-mechanism Coding task.

For N230, a reported export count is checked together with the physical record count and every original key–payload association. For P455, the three mechanism-specific checks each contribute 20 diagnostic points; delivery presence, importability, integrated execution, and the existing loader supply the remaining checks. The diagnostic point total supports error inspection, while the reported outcome is binary. The verifier wrapper exposes the individual checks as tests and retains their names and failure details in `trace_results.json`.

For GUI editor tasks, the controller captures the saved bytes and filenames; the task-specific grader checks the delivery's format and content, using CSV, JSON, key–value, or line comparisons as appropriate. File-manager tasks are graded from the actual directory listing. The saved-delivery adapter recomputes the original checks from captured state and artifact bytes, independently of stored reward fields. Native reference controllers include the intended save or rename and a wrong-name control that appends an extra extension.

Source validation checks the task materials independently of execution outcomes. The current audit verifies all 100 catalog instruction hashes, records hashes for 392 Coding reference, naïve, and verifier files, and verifies 20 GUI package manifests covering 240 files. The GUI audit also checks the model-visible instruction hashes and the mapped native-controller, reference, and grading entry points. For the 40 Coding tasks covered by the earlier execution audit, 232 implementation and verifier file pairs match their preserved copies and recorded hashes. The construction audit checks the source line or JSON entry for each retained explicit Coding control outcome and the preserved sources of the GUI construction summaries.

C.4 Native Construction Evidence

Coding controls. Native control records cover 79 Coding tasks with explicit outcomes on all three operating systems (Table 11). For 40 tasks, the source audit links execution logs and result artifacts to their preserved records. For another 39 tasks, the current catalog's `gate1` fields retain per-OS acceptance outcomes and associated run identifiers. P461 has a run-level acceptance summary (run 36013847269); the table's denominators count explicit per-OS records. Duplicate copies of a record count once per task, OS, and control.

Table 11: **Retained native controls within the 80-task Coding subset.** Entries are passing / recorded task–OS pairs. Source groups distinguish execution artifacts from catalog acceptance cells. Their union covers 79 Coding tasks; the six-model evaluation covers all 100 Coding and GUI tasks.

Control	Linux	macOS	Windows	All pairs
Execution logs and results (40 tasks)				
Reference	40/40	40/40	40/40	120/120
Naïve	32/40	25/40	20/40	77/120
Acceptance cell records (39 tasks)				
Reference	39/39	39/39	39/39	117/117
Naïve	11/39	14/39	5/39	30/117
All explicit records (79 tasks)				
Reference	79/79	79/79	79/79	237/237
Naïve	43/79	39/79	25/79	107/237

Across the explicit records, reference implementations pass 237/237 task–OS checks and naïve implementations pass 107/237. N230's naïve export fails on macOS and passes on Linux and Windows. P455's naïve implementation fails on all three deployments, with different component requirements

implicated by its combined workflow; its reference implementation passes on all three. These controls provide construction evidence for the native realizations and their graders.

GUI construction records. GUI construction summaries identify three-system editor-save probes (runs 35424189620 and 35424354619) and a file-manager rename probe (run 35429369456). Sixteen per-task construction records retain probe and reference-operation summaries; seven specify complete reference-run identifiers, including run 35424730627 for G34. The reference controllers implement native save and rename procedures for each platform. Their summaries retain the recorded run and batch granularity.

The GUI catalog records initial five-trial screening counts for all 20 tasks. Sixteen tasks additionally retain per-trial records covering 240 trials. These construction records are linked to the task packages and graders in the material audit; the formal comparison uses the separate six-model set of 360 GUI outcomes.

D Environment and Harness Configuration

D.1 Models and Native Deployments

Table 12 lists the six model configurations used throughout the analysis. Each contributes 300 runs: 240 Coding runs and 60 GUI runs, covering the same 100 tasks on three deployments. The `mini-chameleon-agent` harness provides a common interface and execution policy across models within each task category.

Table 12: **Evaluated model configurations.** Each model covers 80 Coding and 20 GUI tasks on three deployments, with one run per task and deployment.

Model	Coding runs	GUI runs	Total
Deepseek-v4-flash	240	60	300
GLM-5.3-Flash	240	60	300
Gemini-3.8-Flash	240	60	300
Grok-4-7	240	60	300
GPT-5.6-Sol	240	60	300
Claude-Opus-5	240	60	300

The native deployments use Ubuntu 24.04, macOS 15, and Windows Server 2022, with Python 3.12. An OS label denotes the full deployment, including its filesystem, runtime, installed software, and configuration. Each run starts from reset task state and an independent model session. Task resource bindings supply local paths while preserving the shared objective and completion criteria. Coding tasks initialize a project workspace with the supplied program and task materials. GUI tasks initialize the target directory and open the native application: gedit, TextEdit, or Notepad for editor workflows, and Nautilus, Finder, or Explorer for file-manager workflows on Linux, macOS, and Windows, respectively.

D.2 Coding Context, Tools, and Budget

The function-calling loop exposes `run_command`, `read_file`, `write_file`, `list_dir`, and `finish`. The initial prompt supplies the workspace, platform, shell, and task instruction. The model receives the accumulated conversation and tool feedback, allowing it to inspect resources, implement a solution, test it, and revise the delivery. Commands use the native shell, including `cmd.exe` on Windows. File writes use UTF-8 and preserve supplied line endings; file reads and individual command-output streams are capped at 20,000 characters.

A run allows up to 40 model turns with a nominal 1,500-second execution budget, checked only before starting each turn. Reaching this budget does not interrupt the current turn's model request, retries, or tool calls; recorded elapsed times can therefore exceed 1,500 seconds. A turn can contain multiple tool calls, and each response has an 8,192-token output limit. Requests leave sampling and reasoning controls at provider defaults. Returned reasoning content is retained when required by the provider's tool-call protocol.

D.3 GUI Context, Actions, and Budget

At each step, the controller captures the desktop, requests one JSON action, executes it through the native mouse and keyboard interface, and captures the next observation. The model receives the task instruction with local resource bindings, the current screenshot, and the six most recent action–result pairs. Screenshots are resized to a maximum long edge of 1,280 pixels. Pointer locations use coordinates normalized to the range 0–1,000 and are mapped to the desktop's logical dimensions.

The action vocabulary comprises click, double-click, right-click, drag, type, key combination, scroll, wait, and done, with modifier keys available for selection. The interface operates through the displayed application using these mouse and keyboard actions. The controller resets task files, opens the application, captures the final desktop, and records the delivery for independent evaluation.

Editor tasks allow 40 action steps and file-manager tasks allow 30. The GUI model–action process has a 2,100-second limit within a 2,400-second native-runner limit, and each model response has an 8,192-token budget. The recorded Deepseek-v4-flash GUI configuration enables reasoning at the high setting; the other five GUI configurations leave reasoning controls at provider defaults. Run metadata and runtime manifests record the provider settings and the code used for each run.

D.4 Resource Measurements and Execution Records

Resource summaries include elapsed time, accumulated input and output tokens, API cost, and tool-call counts. Accumulated input usage includes the context supplied on successive requests, including images for GUI calls. Coding interaction counts are executed tool calls, including `finish`; GUI counts are entries in the recorded action history, including completion responses. API expenditure is reported in US dollars at official model prices. Coding elapsed time spans the model–tool loop between its recorded start and end timestamps. GUI elapsed time uses the native runner's recorded duration, covering initialization, interaction, and final-state collection. Missing resource measurements are estimated by the observed mean within the same family; a fully unobserved family uses the mean across observed families within the same interface, model, and deployment. Appendix E.3 gives the token breakdown and interaction counts.

Coding execution records contain the interaction trace, final driver summary, and verifier diagnostics for the delivered program. GUI records contain the action history, retained screenshots, native delivery state, runtime manifest, and evaluator checks. The official outcome files associate each binary score with its model, task, and deployment. These identifiers define the 1,800-outcome matched matrix used for both primary metrics and mechanism-level analyses.

D.5 Prompts and Interaction Protocol

The following boxes document the shared instructions and interaction formats implemented in the Coding and GUI controllers. Placeholders denote values inserted at runtime; task-specific instructions are omitted here. JSON action names and field names follow the controller interface.

Coding: system message

You are a coding agent. You work inside the workspace directory `{workspace}` on a machine running `{platform}`; the run_command tool uses this machine's default shell (`{shell}`). Complete the task described by the user by editing files in the workspace. Use the tools to inspect, edit and test your work. When you are done, call finish.

The task instruction follows as a separate user message. Each request includes the accumulated conversation and the function definitions below. Assistant tool calls and their results are appended as assistant and tool messages, respectively; provider-returned reasoning content is preserved when present.

Coding: function-calling interface (schema summary)

Function	Arguments and behavior
`run_command`	Required: `command` (string). Optional: `timeout_seconds` (integer; default 120). Runs a command in the workspace using the native shell and returns stdout, stderr, and the exit code.
`read_file`	Required: `path` (string). Reads a file; paths may be absolute or relative to the workspace.
`write_file`	Required: `path`, `content` (strings). Writes the complete file, overwriting existing content and creating parent directories as needed.
`list_dir`	Optional: `path` (string; defaults to the workspace root). Lists directory entries.
`finish`	Required: `summary` (string). Signals completion and summarizes the changes in one paragraph.

For responses without tool calls, the controller can append the following reminder, up to twice; a further such response ends the run. A response truncated at the output limit instead terminates with an error.

Coding: completion reminder (user message)

If the task is complete, call the finish tool with a summary. Otherwise continue working with the tools.

GUI: controller instruction (user message)

You are operating the graphical interface of a `{platform}` computer. The screenshot dimensions are `{width}×{height}`.
Task: `{task instruction with local resource bindings}`
Output only one JSON object, with no other text. The available actions are listed below. Click actions may include `modifiers`, such as `["ctrl"]` or `["shift"]`, to hold modifier keys while clicking, for example for multiple selection. Modifiers use physical key names: ctrl, command, shift, and alt.
All click coordinates use normalized values from 0 to 1000, not pixels: the screenshot's top-left corner is (0, 0), the bottom-right is (1000, 1000), and the center is (500, 500). The x coordinate increases to the right and y increases downward. Perform only one action at a time. Output only JSON, without tool-call markers or other text.

GUI: action objects (format examples, not task solutions)

Action	JSON format
Click	`{"action":"click","x":500,"y":300}`
Double-click	`{"action":"double_click","x":500,"y":300}`
Right-click	`{"action":"right_click","x":500,"y":300}`
Type	`{"action":"type","text":"<text>"}`
Key combination	`{"action":"key","keys":["ctrl","s"]}`
Scroll	`{"action":"scroll","dx":0,"dy":-3}`
Drag	`{"action":"drag","x":100,"y":100,"to_x":500,"to_y":500}`
Wait	`{"action":"wait","seconds":2}`
Done	`{"action":"done","reason":"<completion reason>"}`

Key names refer to physical keys (e.g., ctrl, command, shift, alt, enter, tab, backspace, home, and end).

Scrolling occurs at the current pointer position; `dy` is measured in wheel increments, with negative values scrolling downward. Drag endpoints also use normalized coordinates.

Unlike Coding, the GUI controller sends its instruction as a user message, not a system message. It then includes up to six recent action–result pairs: each action is an assistant JSON message, followed by a user feedback message using the template below. The current screenshot is attached to the final user message; on the first step, it is attached to the initial instruction.

GUI: action feedback (user message)

Previous action result: `{result}`. This is the latest screenshot; please provide the next action.

E Complete Results and Aggregation Details

The following tables expand the main comparison using the same 1,800 binary outcomes. Each model contributes one result for each of 100 tasks on each native deployment: 80 Coding tasks and 20 GUI tasks.

E.1 Pass and Fail Counts

Table 13 gives unweighted counts. The final column counts tasks passed by the same model on all three deployments. These counts complement the family-weighted scores: a task contributes $1/(30|T_f|)$ to an overall metric when it belongs to family f.

Table 13: **Observed pass and fail counts.** Deployment columns give pass / fail, totaling 100 tasks per column and model. All three counts tasks passed on every deployment.

Model	Linux	macOS	Windows	All three
Deepseek-v4-flash	70/30	63/37	38/62	10
GLM-5.3-Flash	67/33	58/42	45/55	13
Gemini-3.8-Flash	67/33	67/33	55/45	21
Grok-4-7	69/31	65/35	32/68	8
GPT-5.6-Sol	69/31	71/29	48/52	20
Claude-Opus-5	85/15	75/25	64/36	56

E.2 Complete Family-Level Metrics

Table 14 reports all 180 model–family combinations. Within each family, SR averages the observed outcomes on one deployment, Avg. SR averages the three SR scores, RSR counts tasks passing all three, and EG counts tasks whose outcomes differ across deployments. Each is expressed as a percentage of the tasks in that family, with EG in percentage points. Family identifiers follow Table 3.

Table 14: **Complete family-level results.** All 30 families are shown for each model: 28 Coding families and two GUI families. SR, Avg. SR, and RSR are percentages; EG is in percentage points.

Family	Tasks	Linux SR	macOS SR	Windows SR	Avg. SR	RSR	EG
Deepseek-v4-flash							
M01	4	100.0	100.0	0.0	66.7	0.0	100.0
M02	6	100.0	66.7	66.7	77.8	66.7	33.3
M03	7	14.3	100.0	100.0	71.4	14.3	85.7
M04	5	100.0	0.0	80.0	60.0	0.0	100.0
M05	7	85.7	0.0	100.0	61.9	0.0	100.0
M06	9	100.0	100.0	0.0	66.7	0.0	100.0
M07	2	0.0	0.0	100.0	33.3	0.0	100.0
M08	2	0.0	100.0	100.0	66.7	0.0	100.0
M09	3	100.0	100.0	0.0	66.7	0.0	100.0
M10	1	100.0	100.0	100.0	100.0	100.0	0.0
M11	3	100.0	100.0	0.0	66.7	0.0	100.0
M12	1	100.0	100.0	100.0	100.0	100.0	0.0
M13	1	100.0	100.0	0.0	66.7	0.0	100.0
M14	1	100.0	100.0	0.0	66.7	0.0	100.0
C01	2	50.0	100.0	100.0	83.3	50.0	50.0

Continued on the next page

Table 14 (continued)

Family	Tasks	Linux SR	macOS SR	Windows SR	Avg. SR	RSR	EG
C02	2	50.0	100.0	100.0	83.3	50.0	50.0
C03	1	100.0	100.0	0.0	66.7	0.0	100.0
C04	5	20.0	80.0	0.0	33.3	0.0	80.0
C05	2	0.0	100.0	0.0	33.3	0.0	100.0
C06	2	100.0	50.0	0.0	50.0	0.0	100.0
C07	1	100.0	100.0	100.0	100.0	100.0	0.0
C08	1	0.0	100.0	0.0	33.3	0.0	100.0
C09	1	100.0	100.0	0.0	66.7	0.0	100.0
C10	7	0.0	100.0	0.0	33.3	0.0	100.0
C11	1	0.0	100.0	0.0	33.3	0.0	100.0
C12	1	0.0	0.0	0.0	0.0	0.0	0.0
C13	1	0.0	100.0	0.0	33.3	0.0	100.0
C14	1	100.0	100.0	0.0	66.7	0.0	100.0
G01	14	100.0	7.1	35.7	47.6	0.0	100.0
G02	6	100.0	16.7	0.0	38.9	0.0	100.0
GLM-5.3-Flash							
M01	4	25.0	50.0	0.0	25.0	0.0	50.0
M02	6	100.0	66.7	66.7	77.8	66.7	33.3
M03	7	0.0	100.0	100.0	66.7	0.0	100.0
M04	5	100.0	0.0	100.0	66.7	0.0	100.0
M05	7	100.0	0.0	85.7	61.9	0.0	100.0
M06	9	100.0	100.0	0.0	66.7	0.0	100.0
M07	2	0.0	0.0	100.0	33.3	0.0	100.0
M08	2	0.0	100.0	100.0	66.7	0.0	100.0
M09	3	100.0	100.0	33.3	77.8	33.3	66.7
M10	1	100.0	100.0	100.0	100.0	100.0	0.0
M11	3	100.0	66.7	0.0	55.6	0.0	100.0
M12	1	100.0	0.0	100.0	66.7	0.0	100.0
M13	1	100.0	100.0	0.0	66.7	0.0	100.0
M14	1	100.0	100.0	100.0	100.0	100.0	0.0
C01	2	50.0	100.0	100.0	83.3	50.0	50.0
C02	2	50.0	50.0	0.0	33.3	0.0	50.0
C03	1	0.0	100.0	0.0	33.3	0.0	100.0
C04	5	20.0	60.0	0.0	26.7	0.0	80.0
C05	2	50.0	50.0	0.0	33.3	0.0	50.0
C06	2	100.0	100.0	0.0	66.7	0.0	100.0
C07	1	0.0	0.0	0.0	0.0	0.0	0.0
C08	1	0.0	100.0	0.0	33.3	0.0	100.0
C09	1	100.0	100.0	0.0	66.7	0.0	100.0
C10	7	0.0	85.7	0.0	28.6	0.0	85.7
C11	1	100.0	100.0	0.0	66.7	0.0	100.0
C12	1	100.0	0.0	100.0	66.7	0.0	100.0
C13	1	0.0	0.0	100.0	33.3	0.0	100.0
C14	1	0.0	100.0	0.0	33.3	0.0	100.0
G01	14	100.0	21.4	50.0	57.1	21.4	78.6
G02	6	100.0	50.0	66.7	72.2	33.3	66.7
Gemini-3.8-Flash							
M01	4	50.0	75.0	0.0	41.7	0.0	75.0
M02	6	100.0	66.7	50.0	72.2	50.0	50.0
M03	7	57.1	100.0	85.7	81.0	42.9	57.1
M04	5	100.0	0.0	100.0	66.7	0.0	100.0
M05	7	85.7	0.0	85.7	57.1	0.0	100.0
M06	9	88.9	100.0	0.0	63.0	0.0	100.0
M07	2	50.0	50.0	100.0	66.7	50.0	50.0
M08	2	50.0	100.0	100.0	83.3	50.0	50.0
M09	3	100.0	100.0	0.0	66.7	0.0	100.0
M10	1	0.0	0.0	100.0	33.3	0.0	100.0
M11	3	100.0	100.0	0.0	66.7	0.0	100.0
M12	1	100.0	100.0	0.0	66.7	0.0	100.0
M13	1	100.0	100.0	100.0	100.0	100.0	0.0

Continued on the next page

Table 14 (continued)

Family	Tasks	Linux SR	macOS SR	Windows SR	Avg. SR	RSR	EG
M14	1	100.0	100.0	100.0	100.0	100.0	0.0
C01	2	0.0	100.0	50.0	50.0	0.0	100.0
C02	2	100.0	100.0	100.0	100.0	100.0	0.0
C03	1	100.0	100.0	0.0	66.7	0.0	100.0
C04	5	0.0	100.0	80.0	60.0	0.0	100.0
C05	2	0.0	100.0	50.0	50.0	0.0	100.0
C06	2	100.0	100.0	100.0	100.0	100.0	0.0
C07	1	0.0	0.0	100.0	33.3	0.0	100.0
C08	1	0.0	100.0	0.0	33.3	0.0	100.0
C09	1	100.0	100.0	0.0	66.7	0.0	100.0
C10	7	0.0	100.0	14.3	38.1	0.0	100.0
C11	1	0.0	100.0	0.0	33.3	0.0	100.0
C12	1	0.0	0.0	0.0	0.0	0.0	0.0
C13	1	0.0	100.0	0.0	33.3	0.0	100.0
C14	1	100.0	0.0	0.0	33.3	0.0	100.0
G01	14	85.7	14.3	71.4	57.1	14.3	78.6
G02	6	100.0	83.3	100.0	94.4	83.3	16.7
Grok-4-7							
M01	4	100.0	100.0	0.0	66.7	0.0	100.0
M02	6	100.0	66.7	66.7	77.8	50.0	50.0
M03	7	0.0	100.0	100.0	66.7	0.0	100.0
M04	5	100.0	20.0	100.0	73.3	20.0	80.0
M05	7	100.0	0.0	100.0	66.7	0.0	100.0
M06	9	100.0	100.0	0.0	66.7	0.0	100.0
M07	2	50.0	50.0	100.0	66.7	50.0	50.0
M08	2	0.0	100.0	100.0	66.7	0.0	100.0
M09	3	100.0	100.0	0.0	66.7	0.0	100.0
M10	1	100.0	100.0	0.0	66.7	0.0	100.0
M11	3	100.0	100.0	33.3	77.8	33.3	66.7
M12	1	100.0	100.0	0.0	66.7	0.0	100.0
M13	1	100.0	100.0	0.0	66.7	0.0	100.0
M14	1	100.0	100.0	0.0	66.7	0.0	100.0
C01	2	100.0	100.0	50.0	83.3	50.0	50.0
C02	2	100.0	100.0	0.0	66.7	0.0	100.0
C03	1	100.0	100.0	0.0	66.7	0.0	100.0
C04	5	0.0	80.0	0.0	26.7	0.0	80.0
C05	2	0.0	100.0	0.0	33.3	0.0	100.0
C06	2	100.0	100.0	0.0	66.7	0.0	100.0
C07	1	0.0	0.0	0.0	0.0	0.0	0.0
C08	1	0.0	0.0	0.0	0.0	0.0	0.0
C09	1	100.0	100.0	0.0	66.7	0.0	100.0
C10	7	0.0	100.0	0.0	33.3	0.0	100.0
C11	1	100.0	100.0	0.0	66.7	0.0	100.0
C12	1	0.0	0.0	0.0	0.0	0.0	0.0
C13	1	0.0	0.0	0.0	0.0	0.0	0.0
C14	1	100.0	100.0	0.0	66.7	0.0	100.0
G01	14	78.6	14.3	0.0	31.0	0.0	78.6
G02	6	100.0	33.3	50.0	61.1	16.7	83.3
GPT-5.6-Sol							
M01	4	100.0	100.0	25.0	75.0	25.0	75.0
M02	6	100.0	66.7	66.7	77.8	66.7	33.3
M03	7	0.0	100.0	100.0	66.7	0.0	100.0
M04	5	100.0	80.0	100.0	93.3	80.0	20.0
M05	7	100.0	28.6	100.0	76.2	28.6	71.4
M06	9	100.0	100.0	0.0	66.7	0.0	100.0
M07	2	0.0	0.0	100.0	33.3	0.0	100.0
M08	2	0.0	100.0	100.0	66.7	0.0	100.0
M09	3	100.0	66.7	0.0	55.6	0.0	100.0
M10	1	100.0	100.0	100.0	100.0	100.0	0.0
M11	3	100.0	100.0	0.0	66.7	0.0	100.0

Continued on the next page

Table 14 (continued)

Family	Tasks	Linux SR	macOS SR	Windows SR	Avg. SR	RSR	EG
M12	1	100.0	100.0	100.0	100.0	100.0	0.0
M13	1	100.0	100.0	0.0	66.7	0.0	100.0
M14	1	100.0	100.0	0.0	66.7	0.0	100.0
C01	2	0.0	0.0	100.0	33.3	0.0	100.0
C02	2	50.0	50.0	0.0	33.3	0.0	50.0
C03	1	100.0	100.0	0.0	66.7	0.0	100.0
C04	5	20.0	100.0	0.0	40.0	0.0	100.0
C05	2	0.0	100.0	0.0	33.3	0.0	100.0
C06	2	100.0	100.0	0.0	66.7	0.0	100.0
C07	1	0.0	0.0	0.0	0.0	0.0	0.0
C08	1	0.0	100.0	0.0	33.3	0.0	100.0
C09	1	100.0	100.0	0.0	66.7	0.0	100.0
C10	7	0.0	100.0	0.0	33.3	0.0	100.0
C11	1	0.0	100.0	0.0	33.3	0.0	100.0
C12	1	100.0	0.0	0.0	33.3	0.0	100.0
C13	1	0.0	100.0	0.0	33.3	0.0	100.0
C14	1	100.0	100.0	0.0	66.7	0.0	100.0
G01	14	100.0	28.6	71.4	66.7	28.6	71.4
G02	6	100.0	50.0	100.0	83.3	50.0	50.0
Claude-Opus-5							
M01	4	100.0	100.0	25.0	75.0	25.0	75.0
M02	6	100.0	83.3	100.0	94.4	83.3	16.7
M03	7	85.7	100.0	100.0	95.2	85.7	14.3
M04	5	100.0	100.0	100.0	100.0	100.0	0.0
M05	7	85.7	100.0	100.0	95.2	85.7	14.3
M06	9	100.0	77.8	33.3	70.4	22.2	77.8
M07	2	50.0	50.0	100.0	66.7	50.0	50.0
M08	2	100.0	100.0	100.0	100.0	100.0	0.0
M09	3	100.0	100.0	100.0	100.0	100.0	0.0
M10	1	100.0	100.0	100.0	100.0	100.0	0.0
M11	3	100.0	100.0	100.0	100.0	100.0	0.0
M12	1	0.0	100.0	0.0	33.3	0.0	100.0
M13	1	100.0	100.0	100.0	100.0	100.0	0.0
M14	1	100.0	100.0	100.0	100.0	100.0	0.0
C01	2	100.0	100.0	100.0	100.0	100.0	0.0
C02	2	50.0	100.0	0.0	50.0	0.0	100.0
C03	1	100.0	0.0	0.0	33.3	0.0	100.0
C04	5	60.0	60.0	60.0	60.0	60.0	0.0
C05	2	100.0	100.0	100.0	100.0	100.0	0.0
C06	2	100.0	100.0	100.0	100.0	100.0	0.0
C07	1	0.0	0.0	0.0	0.0	0.0	0.0
C08	1	100.0	100.0	100.0	100.0	100.0	0.0
C09	1	0.0	0.0	0.0	0.0	0.0	0.0
C10	7	42.9	57.1	57.1	52.4	42.9	14.3
C11	1	100.0	100.0	0.0	66.7	0.0	100.0
C12	1	100.0	100.0	100.0	100.0	100.0	0.0
C13	1	0.0	100.0	100.0	66.7	0.0	100.0
C14	1	100.0	100.0	0.0	66.7	0.0	100.0
G01	14	92.9	7.1	7.1	35.7	0.0	100.0
G02	6	100.0	100.0	83.3	94.4	83.3	16.7

E.3 Resource Breakdown

Resource accounting and aggregation. We report API expenditure, elapsed time, and token consumption separately, with interaction counts providing an additional description of execution. For each model, deployment, and reported task set, resource measurements use the same family weights as the completion metrics in Equation 2: measurements are averaged within each family and then equally across the included families. Both successful and failed runs contribute, so the summaries describe resources consumed per attempted task rather than per successful completion. Table 15 assigns equal weight to all 30 families, matching the overall block of Table 2; subset summaries

normalize weights over their respective families. Missing measurements follow the procedure in Appendix D: the observed mean within the same family is used, with the mean across observed families in the same interface, model, and deployment as the fallback for a fully unobserved family.

Measurement units and scope. API expenditure is expressed in US dollars at official model prices, elapsed time in seconds, and token counts in thousands per run. Input and output tokens accumulate across all model calls; input usage includes context supplied on successive requests and images for GUI calls. The main table reports their sum, calculated before rounding, while Table 15 separates the two components. Coding elapsed time spans the recorded model–tool loop; GUI elapsed time covers the native runner's initialization, interaction, and final-state collection. Interaction counts comprise executed Coding tool calls, including `finish`, or recorded GUI action-history entries, including completion responses. These interface-specific definitions should be considered when comparing elapsed time and interaction counts across Coding and GUI tasks.

Resource use and robust completion. The overall results in Table 2 show distinct resource profiles across models. Claude-Opus-5 attains 57.9% RSR with 47.6–66.1 thousand tokens and 108.4–136.5 seconds per run across deployments, at \$0.365–\$0.456. Gemini-3.8-Flash attains 23.0% RSR with 715.4–760.4 thousand tokens and 385.9–487.0 seconds, at \$0.285–\$0.311. GPT-5.6-Sol has the lowest total token usage and shortest elapsed time on each deployment, using 16.0–18.6 thousand tokens and 52.3–70.6 seconds at \$0.060–\$0.066, with 16.0% RSR. GLM-5.3-Flash has the lowest expenditure, \$0.014–\$0.025 per run, but the longest elapsed time, 724.1–973.3 seconds, and reaches 13.5% RSR. Deepseek-v4-flash and Grok-4-7 attain 16.0% and 7.3% RSR at \$0.041–\$0.080 and \$0.115–\$0.201 per run, respectively. Thus, token volume, elapsed time, and expenditure yield different model orderings, and higher resource use does not consistently coincide with higher robust completion in this evaluation.

Token composition and interaction counts. Accumulated input tokens exceed output tokens for every model–deployment pair in Table 15. For Gemini-3.8-Flash, 684.9–728.9 thousand input tokens accompany 30.5–32.8 thousand output tokens per run; for GPT-5.6-Sol, the corresponding ranges are 13.5–15.6 thousand and 2.6–3.0 thousand. Because input usage accumulates context across successive requests, these totals measure supplied context as well as generated output. Gemini-3.8-Flash also records the most calls on each deployment, at 25.0–29.4 per run, compared with 7.0–11.3 for Claude-Opus-5 and 8.7–9.3 for GPT-5.6-Sol. Call counts complement token totals, although the combined Coding and GUI averages include different interaction types and do not represent a uniform amount of work per call.

Variation across deployments. Resource use also changes within a model across operating systems. Deepseek-v4-flash uses 104.7 thousand tokens, 12.9 calls, and \$0.041 per run on macOS, compared with 222.0 thousand tokens, 16.9 calls, and \$0.080 on Windows; its SR decreases from 77.3% to 36.1%. Claude-Opus-5 similarly increases from 47.6 thousand tokens and 108.4 seconds on macOS to 66.1 thousand tokens and 136.5 seconds on Windows, while SR decreases from 81.2% to 65.5%. The pattern differs for Gemini-3.8-Flash: Windows has its largest token total (760.4 thousand) but shortest elapsed time (385.9 seconds). These observations motivate reporting all resource dimensions alongside completion outcomes. The reported averages summarize the single-run evaluation and do not isolate the causes of resource differences or estimate the effect of allocating additional compute to a fixed model.

E.4 Sensitivity to Weighting and Task Composition

The overall analysis gives each of 30 families equal total weight. Task weighting instead gives each of 100 tasks weight $1/100$. As Table 16 shows, Claude-Opus-5 ranks first in SR on every deployment and in RSR under both schemes. Its RSR is 57.9% with family weights and 56.0% with task weights. The intermediate ordering can change: GPT-5.6-Sol's task-weighted RSR is 20.0%, above Deepseek-v4-flash's 10.0%, whereas their family-weighted RSR values are 15.96% and 16.03%, respectively.

Single- and multi-mechanism Coding each contain 14 families, contributing $14/30$ of the overall family weight apiece; the two GUI families contribute $2/30$. Their task counts are 52, 28, and 20, so task weighting assigns them $52/100$, $28/100$, and $20/100$, respectively. Table 17 reports

Table 15: **Resource use by model and deployment across all tasks.** Cost is USD/run, time is seconds/run, tokens are thousands/run, and calls are Coding tool invocations or GUI action-history entries per run.

Model	OS	Cost	Time	Input tok.	Output tok.	Calls
Deepseek-v4-flash	Linux	0.055	73.3	136.8	11.3	13.8
	macOS	0.041	59.1	94.2	10.6	12.9
	Windows	0.080	78.6	206.8	15.2	16.9
GLM-5.3-Flash	Linux	0.014	724.1	51.7	11.0	12.7
	macOS	0.021	973.3	65.9	16.1	14.7
	Windows	0.025	881.0	88.5	19.1	18.2
Gemini-3.8-Flash	Linux	0.288	460.6	690.0	32.8	25.0
	macOS	0.285	487.0	684.9	30.5	27.3
	Windows	0.311	385.9	728.9	31.5	29.4
Grok-4-7	Linux	0.115	182.3	57.7	9.7	11.3
	macOS	0.201	257.8	129.5	15.1	15.1
	Windows	0.136	168.3	76.4	11.2	13.4
GPT-5.6-Sol	Linux	0.060	52.3	13.5	2.6	8.7
	macOS	0.066	59.6	14.3	2.8	8.9
	Windows	0.066	70.6	15.6	3.0	9.3
Claude-Opus-5	Linux	0.413	118.7	43.1	9.0	7.0
	macOS	0.365	108.4	39.5	8.1	8.0
	Windows	0.456	136.5	56.3	9.8	11.3

Table 16: **Family and task weighting.** Family assigns equal total weight to the 30 families; Task weights all 100 tasks equally. Scores are percentages, with EG in percentage points.

Model	Weights	Linux SR	macOS SR	Windows SR	Avg. SR	RSR	EG
Deepseek-v4-flash	Family	64.0	77.3	36.1	59.1	16.0	80.0
	Task	70.0	63.0	38.0	57.0	10.0	88.0
GLM-5.3-Flash	Family	59.8	63.3	43.4	55.5	13.5	77.0
	Task	67.0	58.0	45.0	56.7	13.0	80.0
Gemini-3.8-Flash	Family	58.9	73.0	49.6	60.5	23.0	72.6
	Task	67.0	67.0	55.0	63.0	21.0	76.0
Grok-4-7	Family	67.6	72.1	23.3	54.4	7.3	78.0
	Task	69.0	65.0	32.0	55.3	8.0	84.0
GPT-5.6-Sol	Family	62.3	75.7	35.4	57.8	16.0	79.0
	Task	69.0	71.0	48.0	62.7	20.0	78.0
Claude-Opus-5	Family	78.9	81.2	65.5	75.2	57.9	32.6
	Task	85.0	75.0	64.0	74.7	56.0	37.0

combined Coding, single-mechanism Coding, multi-mechanism Coding, and GUI results for each model. Weights are normalized within each reported subset: 28, 14, 14, and two equally weighted families, respectively. All six models have lower mean SR and RSR on multi-mechanism Coding tasks than on single-mechanism Coding tasks. Claude-Opus-5's RSR decreases from 75.1% to 43.1%, while GPT-5.6-Sol's decreases from 28.6% to 0.0%. On GUI tasks, Gemini-3.8-Flash has the highest RSR at 48.8%, followed by Claude-Opus-5 at 41.7% and GPT-5.6-Sol at 39.3%.

Table 17: **Results by interface and Coding composition.** Coding contains 80 tasks in 28 families; Single and Multiple contain 52 and 28 tasks in 14 families each; GUI contains 20 tasks in two families. Families are weighted equally within each reported subset. SR and RSR are percentages; EG is in percentage points.

Model	Subset	Avg. SR	RSR	EG
Deepseek-v4-flash	Coding	60.3	17.2	78.5
	Single	69.4	20.1	79.9
	Multiple	51.2	14.3	77.1
	GUI	43.3	0.0	100.0
GLM-5.3-Flash	Coding	54.9	12.5	77.3
	Single	66.5	21.4	75.0
	Multiple	43.2	3.6	79.7
	GUI	64.7	27.4	72.6
Gemini-3.8-Flash	Coding	59.4	21.2	74.4
	Single	68.9	28.1	70.2
	Multiple	49.9	14.3	78.6
	GUI	75.8	48.8	47.6
Grok-4-7	Coding	55.0	7.3	77.7
	Single	68.7	11.0	89.0
	Multiple	41.2	3.6	66.4
	GUI	46.0	8.3	81.0
GPT-5.6-Sol	Coding	56.6	14.3	80.3
	Single	72.2	28.6	71.4
	Multiple	41.0	0.0	89.3
	GUI	75.0	39.3	60.7
Claude-Opus-5	Coding	75.9	59.1	30.8
	Single	87.9	75.1	24.9
	Multiple	64.0	43.1	36.7
	GUI	65.1	41.7	58.3

E.5 Omitting Individual Mechanism Families

For each model, we omit one family at a time and assign equal weights to the remaining 29 families. Table 18 reports the minimum and maximum of each metric over the 30 omissions. Claude-Opus-5's RSR ranges from 56.5% to 59.9%; the highest RSR attained by any other model under an omission is 23.8%. Its leading paired completion rate therefore persists across all individual-family omissions.

Table 18: **Leave-one-family-out ranges.** Each interval spans the 30 scores obtained by omitting one family and reweighting the others equally. These are sensitivity ranges. SR and RSR are percentages; EG is in percentage points.

Model	Linux SR	macOS SR	Windows SR	Avg. SR	RSR	EG
Deepseek-v4-flash	62.8–66.2	76.6–80.0	33.9–37.3	57.7–61.2	13.1–16.6	79.3–82.7
GLM-5.3-Flash	58.4–61.9	62.1–65.5	41.5–44.9	54.0–57.4	10.5–14.0	76.2–79.7
Gemini-3.8-Flash	57.5–60.9	72.0–75.5	47.8–51.3	59.1–62.6	20.4–23.8	71.6–75.1
Grok-4-7	66.5–70.0	71.2–74.6	20.7–24.1	53.4–56.2	5.9–7.6	77.2–80.6
GPT-5.6-Sol	61.0–64.5	74.8–78.3	33.2–36.7	56.4–59.8	13.1–16.5	78.3–81.8
Claude-Opus-5	78.2–81.6	80.5–84.0	64.3–67.8	74.4–77.8	56.5–59.9	30.3–33.8

E.6 Task-Level Outcome Heatmaps

Figures 8–10 visualize the complete task–model outcome matrix, following the task-level presentation in Figures 14–16 of Agents' Last Exam (Sun et al., 2026b). Each figure contains 100 tasks—80

Coding and 20 GUI—and six models for one native deployment. Rows are sorted by descending mean outcome across models, and columns by descending mean outcome across tasks; identifiers break ties. Each cell shows the observed binary outcome.

Shared successes and failures vary across deployments. All six models pass 49 tasks on Linux, 37 on macOS, and 23 on Windows. Every model fails seven tasks on Linux, ten on macOS, and 17 on Windows. The remaining 44, 53, and 60 tasks, respectively, have mixed outcomes across models. Thus, both shared failures and model-dependent successes contribute to the deployment-level differences.

Paired coverage across all three environments. At least one model completes all three realizations of 67 tasks: 57 Coding tasks and ten GUI tasks. The remaining 33 tasks have no evaluated model that passes on every deployment. Two Coding tasks pass under every model on all three OSs; the corresponding GUI intersection is empty. Claude-Opus-5 individually completes all three realizations of 56 tasks. These counts measure the overlap of successful task sets without family weighting, complementing RSR in Table 2.

F Execution Evidence and Future Work

F.1 Task-Level Evidence Across Models

We examine four Coding and two GUI task conditions to connect the paired outcomes to requirements on delivered programs and desktop artifacts. N230 and H330 each have one annotated mechanism; P455 and P456 have three and four, respectively. G33 and G61 cover native editor saving and file-manager renaming. Table 19 includes all six models on the same six conditions, spanning the three deployments. The examples explain the checks behind the binary outcomes; aggregate comparisons use the complete 100-task matrix.

Table 19: **Coding and GUI execution examples across all six models.** Each entry is the official pass/fail result for the specified task and deployment. The four Coding conditions cover single- and multi-mechanism tasks; the two GUI conditions cover the two native-workflow families.

	Coding				GUI	
Model	N230 macOS	H330 Linux	P455 Linux	P456 Windows	G33 Windows	G61 Windows
Deepseek-v4-flash	Fail	Fail	Pass	Fail	Fail	Fail
GLM-5.3-Flash	Fail	Fail	Fail	Pass	Pass	Fail
Gemini-3.8-Flash	Fail	Pass	Fail	Fail	Fail	Pass
Grok-4-7	Fail	Fail	Fail	Fail	Fail	Fail
GPT-5.6-Sol	Pass	Fail	Pass	Fail	Fail	Pass
Claude-Opus-5	Pass	Pass	Fail	Pass	Fail	Pass

Single-mechanism tasks: identity and lookup. N230 requires an export to preserve every source key and its payload (M05, Unicode filename normalization). On macOS, the four failing deliveries return the expected export count and ordered key list, but the grader finds three stored records for four source keys. Retrieval under one original key returns the payload belonging to a distinct, canonically equivalent Unicode key. The returned summary is correct while the stored records lose a required distinction. GPT-5.6-Sol and Claude-Opus-5 satisfy all checks.

H330 requires equivalent hexadecimal digest spellings to locate the same stored object (M03, digest case normalization). On Linux, four deliveries report all four requested objects as missing. The evaluator expects three objects to be present, including one with corrupted content, and only one to be absent. These implementations preserve the API and return the expected number of checked entries, but fail the presence, missing-object, and integrity classifications. Gemini-3.8-Flash and Claude-Opus-5 pass. Together, these cases distinguish a correct response shape from correct identity and lookup behavior.

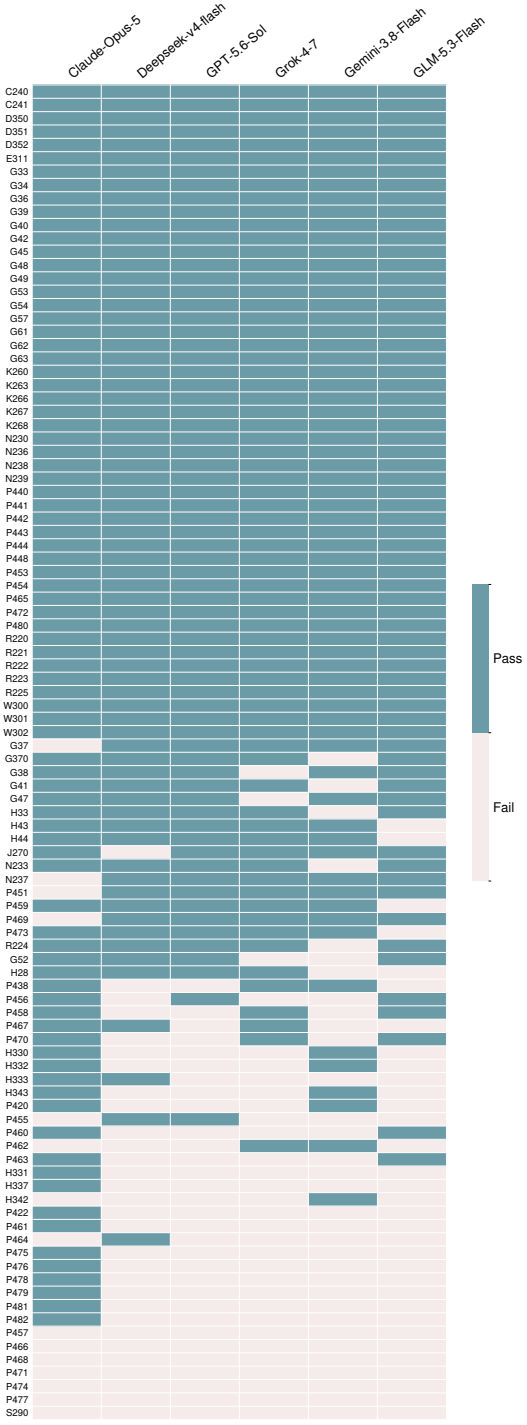


Figure 8: **Task-level outcomes on Linux (100 Coding and GUI tasks).** Rows denote tasks and columns denote models under the shared harness. Teal denotes success; pale coral denotes failure. The color scale is discrete.

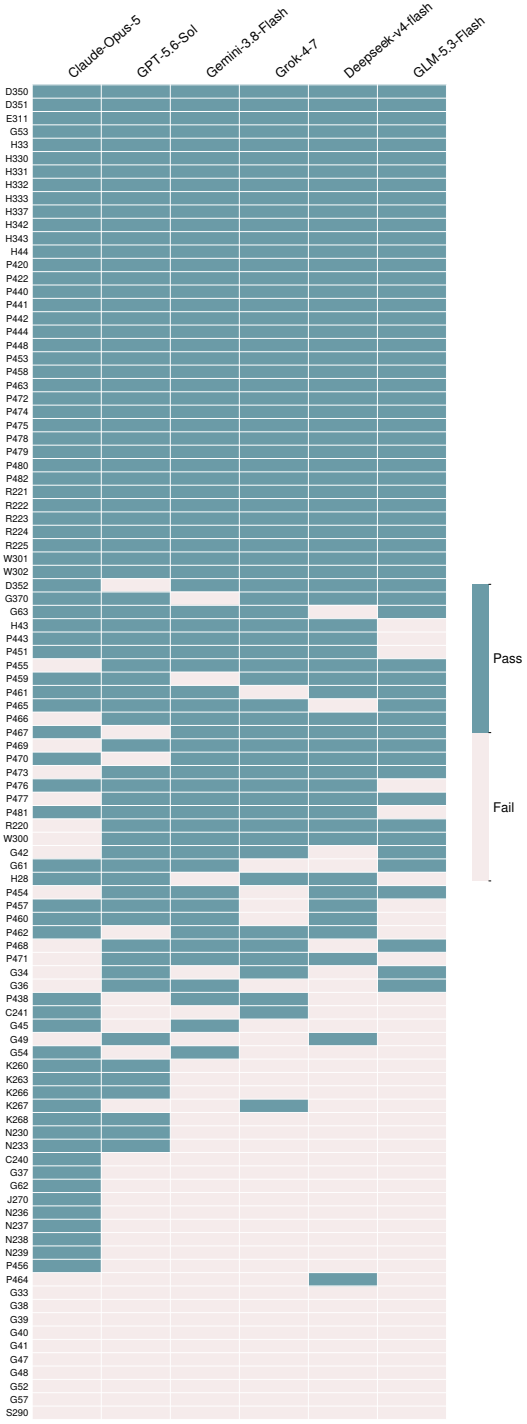


Figure 9: **Task-level outcomes on macOS (100 Coding and GUI tasks).** Layout and colors follow Figure 8; rows and columns are sorted by their mean scores on macOS.

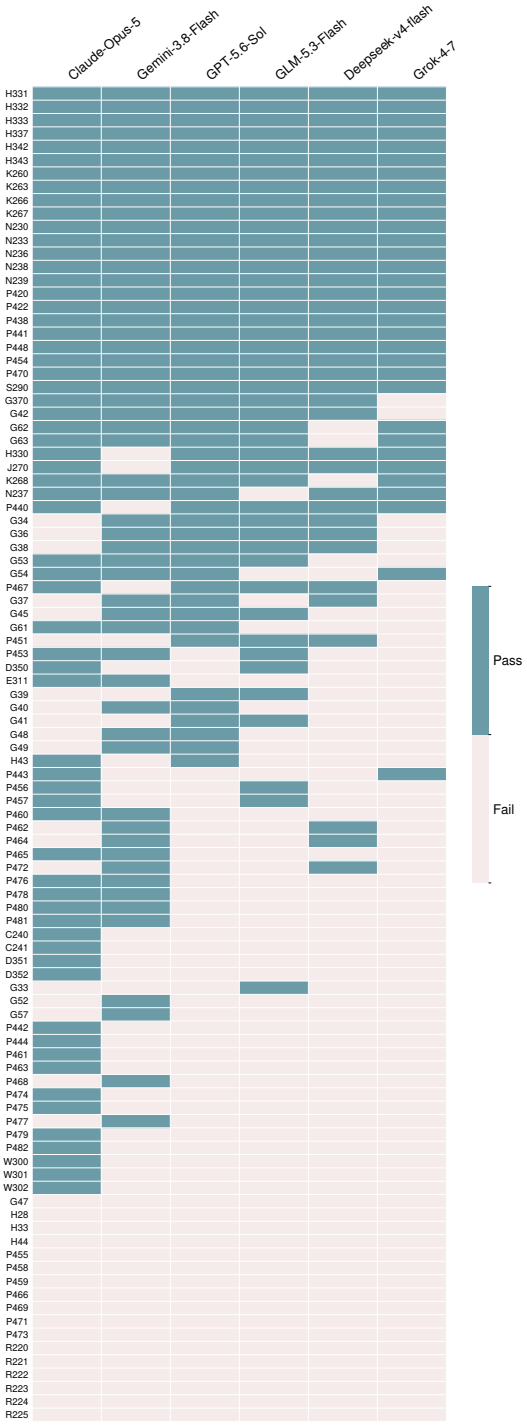


Figure 10: **Task-level outcomes on Windows (100 Coding and GUI tasks).** Layout and colors follow Figure 8; rows and columns are sorted by their mean scores on Windows.

Multi-mechanism tasks: joint delivery requirements. P455 imports a notes bundle by removing the previous import, writing notes, and preserving each attachment's original name and content. Its complete mechanism set is M05+M08+M11: Unicode filename normalization, filename byte-length limits, and read-only file deletion. On Linux, GLM-5.3-Flash, Gemini-3.8-Flash, and Grok-4-7 pass the cleanup and attachment checks, but the long note title produces `OSError: [Errno 36] File name too long`; both note creation and the end-to-end import fail. Deepseek-v4-flash and GPT-5.6-Sol pass every check. The Claude-Opus-5 record contains no tool calls, and the evaluator encounters `NotImplementedError` in all three requested functions. Thus, the four failed deliveries have two distinct observed failure patterns.

P456 combines roster-file decoding, piped updates, distinct team directories, and member-card creation (M05+M08+M13+M14). On Windows, all six deliveries preserve the names read from the roster file and pass the separate count check for the two canonically equivalent team names. Four deliveries fail the exact-text check on piped updates. Grok-4-7 and GPT-5.6-Sol also fail the long-team-name check with `WinError 123` and do not complete the end-to-end synchronization. GLM-5.3-Flash and Claude-Opus-5 satisfy every check. A single binary failure can therefore summarize one or several violated requirements within the same complete mechanism set.

GUI tasks: native actions and exact delivery. G33 asks the model to use the open editor to save an extensionless `Dockerfile` with four specified lines. In the GLM-5.3-Flash Windows record, the action history contains text entry, `Ctrl+S`, and the quoted filename `"Dockerfile"` in the save workflow. The final screenshot shows the document open in Notepad. The recorded file state satisfies all four checks: exact filename, plain-text format, exact lines, and absence of extra files. This condition connects the native save interaction to a precisely specified stored artifact.

G61 asks the model to rename `spec.txt` to exactly `spec.yaml` while preserving the other file. The Claude-Opus-5 Windows action history includes a right-click, entry of `spec.yaml`, and confirmation. The final Explorer screenshot shows `spec.yaml` and `summary.txt`; the recorded checks confirm the new name, removal of the old name, preservation of the other entry, and absence of extra files. Gemini-3.8-Flash and GPT-5.6-Sol also pass this condition. Table 19 reports the formal outcomes for every model on both workflows.

The execution audit links all 36 model–task–deployment records to their evaluator checks, reward files, official outcomes, and execution evidence, with SHA256 hashes for every source. Coding records include trajectory line numbers; GUI records include action indices, retained initial and final screenshots, and the recorded delivery state. The solutions were produced independently on their specified deployments.

F.2 Future Work: Replay and Controlled Modifications

Frozen-procedure replay and controlled modifications remain future work and were not conducted in this study. Replaying a fixed delivered program or GUI action procedure across reset deployments, without further model adaptation, could assess its portability separately from task completion in independent sessions. Comparing targeted modifications with unrelated controls could further test hypothesized sources of environmental differences, using matched initial states and the original evaluators. Such experiments would require predefined case selection and adaptation rules, and should report both recovery on failing deployments and preservation of success elsewhere. The present evidence consists of independently generated solutions, outcome verification, held-out prediction, and cross-model agreement; these analyses do not establish causal effects of individual mechanisms.

G Related Work

G.1 Outcome-Based Evaluation

Executable evaluation specifies both what a model delivers and how completion is established. In SWE-bench, the deliverable is a repository patch evaluated by the repository's testing framework (Jimenez et al., 2024). ScienceAgentBench standardizes its deliverable as a self-contained Python program and evaluates task-specific outputs (Chen et al., 2025). Terminal-Bench verifies the state produced by a sequence of terminal interactions in a prepared environment (Merrill et al.,

2026). These choices allow different implementation strategies to be assessed against a common objective.

ChameleonBench pairs program construction and native desktop workflows across deployments under shared task contracts. Coding evaluators check program behavior and resulting artifacts, including logical record identity, naming constraints, and content. GUI evaluators check the final delivery produced through native application actions, including its name, format, and text. The common contract defines a binary outcome for each task–deployment pair, supporting direct comparison of independently produced solutions. Appendix C describes the two interfaces and their verification.

G.2 Repetition, Coordination, and Deployment Pairing

Reliability can be evaluated along different axes. τ-bench uses pass^k to measure successful completion across repeated trials of the same task, with variation arising from sampled user and agent interactions (Yao et al., 2024). CRAB evaluates workflows that coordinate actions across multiple environments, using a graph of intermediate state checks to measure progress and completion (Xu et al., 2025). ChameleonBench pairs separate attempts at the same objective across Linux, macOS, and Windows. Under the current one-run-per-deployment protocol, RSR measures the weighted fraction of tasks completed on all three deployments.

Deployment pairing permits the model to adapt its program or GUI actions to each environment. In contrast, frozen-procedure replay would assess the portability of a fixed delivery without further adaptation. Replay and controlled modifications are future work (Appendix F.2) and are not part of the present evaluation.

G.3 Mechanism Coverage and Held-Out Prediction

CheckList separates the capability being tested from the test applied to it, providing a structured account of behavioral coverage (Ribeiro et al., 2020). CFQ separates the coverage of primitive elements from the distribution of their compositions when constructing train–test splits (Keysers et al., 2019). ChameleonBench uses elementary mechanisms to describe Coding execution requirements and complete mechanism sets to define Coding families. Its 52 single-mechanism and 28 multi-mechanism Coding tasks form 28 families. The 20 GUI tasks add two families covering native editor saving and file-manager extension visibility, yielding 30 mutually exclusive families for aggregation.

The prediction analysis in Section 4.2 holds out one task within an observed family and estimates its signed OS difference from the remaining tasks in that family. Interface and target-OS predictors supply comparison pools within the same interaction interface. This analysis tests whether the family label carries information about a new instance of the same Coding mechanism configuration or GUI workflow family. The further evaluations in Appendix H extend this design to additional workflows and mechanism configurations.

H Scope and Further Evaluation

The six-model results characterize performance under `mini-chameleon-agent` on 80 Coding and 20 GUI tasks selected for environment sensitivity. Holding task contracts and the two interface policies fixed supports comparison within this setting. The mechanism-level analyses examine how patterns within Coding mechanism sets and GUI workflow families relate to aggregate completion and cross-environment robustness.

An OS condition denotes a complete deployment, including the filesystem, runtime, applications, and configuration. The macro-level definition of environment also includes software versions, permissions, hardware, network conditions, and application state. Evaluating those dimensions would extend the scope beyond the three OS realizations studied here.

Further evaluation can repeat the full six-model matrix, vary interaction budgets under matched conditions, and compare alternative harnesses on the same task contracts. New tasks within represented families can test whether the observed patterns persist across additional workflows. Unseen Coding mechanism combinations, additional elementary mechanisms, and new GUI workflows can

extend coverage. Future work could use frozen-procedure replay and controlled modifications to test hypothesized execution pathways (Appendix F.2); these experiments were not conducted here. Versioned tasks, model identifiers, and deployment manifests support comparisons as models and software evolve.

I Reproduction from Retained Records

The task-bank index `TASK_CATALOG.json` specifies the 100 formal tasks. Coding definitions and membership appear in `MECHANISM_TAXONOMY.json` and `MECHANISM_MAPPING.json`; GUI membership appears in `GUI/MECHANISM_MAPPING.json`. Together, these records specify 80 Coding tasks with 14 elementary mechanisms and 28 complete-set families, plus 20 GUI tasks in two workflow families. Official outcomes and execution traces are indexed by model, task, and native deployment under `results/official/`.

The analysis scripts share `coding_statistics.py` and store outputs in `figure_archive/analysis/coding_gui_20260925/`. Primary scores, all 1,800 outcomes, resource measurements, and source hashes appear in `main_results.json`. Weighting and composition analyses, family omissions, and successful-task overlaps appear in `supporting_analysis.json`. Family differences, held-out predictions and training-task identifiers, prediction errors, and model-pair direction comparisons appear in `mechanism_analysis.json`. CSV exports retain the individual rows. The asset manifest publishes generated TeX tables and PDF figures into both manuscripts.

The appendix exporter calls `summarize_coding_construction.py` and `summarize_coding_execution.py` to audit the two interfaces. Construction sources, initial screening records, reference and naïve Coding controls, and retained native GUI evidence are indexed in `construction_audit.json`. For the 36 execution examples, `execution_audit.json` links each model–task–deployment condition to its checks, official binary outcome, and execution evidence: 24 Coding records and 12 GUI records, with action histories and retained screenshots for GUI. The teaser builder produces editable HTML and SVG, a PDF, and a record of the displayed statistics from the same result summaries. The task-distribution builder verifies the 30-family partition against the task catalog and exports the Coding and GUI rings together with their task and family counts.

From the repository root, the following commands regenerate the results, appendix evidence, figures, tables, and manuscripts:

```
python3 figure_archive/tools/summarize_iclr_main_results.py
python3 figure_archive/tools/summarize_iclr_mechanism_results.py
python3 figure_archive/tools/summarize_iclr_appendices.py
python3 figure_archive/tools/build_coding_teaser.py
python3 figure_archive/tools/build_task_distribution.py
bash compile_iclr.sh
bash compile_paper.sh
```

The summaries use Python, NumPy, and Matplotlib; HTML-to-PDF export uses Chromium, and manuscript compilation uses Tectonic. These commands read the retained outcomes and traces without making model calls or executing benchmark tasks. Source hashes and model–task–OS identifiers link each aggregate to its inputs.

J Extended Case Studies

We examine how environmental semantics affect task completion through data identity, combined execution constraints, and GUI delivery. The cases connect implementation choices and interaction sequences to independently evaluated outcomes. Appendix J provides the corresponding execution records.

Preserving data identity across environments (N236). N236 requires storing distinct keys while preserving their associated payloads. On macOS, GPT-5.6-Sol retained a direct key-to-filename

mapping and passed two self-tests. Independent evaluation nevertheless found that four keys produced only three files, with one key returning another record's payload: normalization-equivalent Hangul keys addressed the same file on the tested APFS deployment. Claude-Opus-5 instead probed collisions through the storage API and encoded the original key bytes in its filename mapping. Its submission preserved all four records and passed evaluation. The contrast identifies a mismatch between logical key identity and native filename equivalence as the source of the observed data loss.

Partial adaptation under combined environmental constraints. Multi-mechanism tasks expose failures in different components of the same workflow. In P474, Gemini-3.8-Flash failed long-title creation on Linux, passed all checks on macOS, and failed input decoding and read-only file deletion on Windows. Its Windows probe reported UTF-8 stdin and 21 self-tests passed, but the evaluation subprocess removed the Python encoding overrides. The submitted reader then produced corrupted names, while deletion separately raised a permission error. P455 reveals a related limitation: Grok-4-7 detected the Linux filename-length limit and introduced a mitigation, then removed it before submission; the long-title and end-to-end checks subsequently failed. In contrast, Claude-Opus-5's Windows implementation of P480 combined explicit byte decoding, indexed storage of distinct Unicode names, and permission-aware deletion. Its tests varied the decoding configuration, and all independent checks passed. These observations distinguish successful handling of individual constraints from satisfying the complete task under its execution conditions.

GUI actions and delivered artifacts (G33). G33 requires a plain-text file named exactly Dockerfile with four prescribed lines. On Windows, Gemini-3.8-Flash entered the unquoted name and declared completion, but the resulting file was Dockerfile.txt. GLM-5.3-Flash entered the filename with quotation marks during its save sequence and produced the required extensionless file. Both artifacts contained the specified text, while only GLM's submission passed all artifact checks. The difference was therefore in the delivered filename rather than the generated content. This case illustrates how application save conventions can invalidate an otherwise correct artifact and why completion must be assessed from the resulting filesystem state.

The following cases examine failures and successful adaptations involving text decoding, filename identity, filesystem constraints, and GUI save behavior. For each task, we relate the agent's implementation or actions to the resulting artifacts and independent evaluation.

Table 20: Task outcomes for the case studies. Each triplet is ordered as Linux/macOS/Windows.

Single-mechanism and GUI tasks						
Case	DeepSeek Flash	GLM 5.3-Flash	Gemini 3.8-Flash	Grok 4-7	GPT 5.6-Sol	Claude Opus-5
E311	P/P/F	P/P/F	P/P/P	P/P/F	P/P/F	P/P/P
N236	P/F/P	P/F/P	P/F/P	P/F/P	P/F/P	P/P/P
H330	F/P/P	F/P/P	P/P/F	F/P/P	F/P/P	P/P/P
G33	P/F/F	P/F/P	P/F/F	P/F/F	P/F/F	P/F/F
G53	P/P/F	P/P/P	P/P/P	P/P/F	P/P/P	P/P/P
G34	P/F/P	P/P/P	P/F/P	P/P/F	P/P/P	P/F/F
Multi-mechanism tasks						
Case	DeepSeek Flash	GLM 5.3-Flash	Gemini 3.8-Flash	Grok 4-7	GPT 5.6-Sol	Claude Opus-5
P455	P/P/F	F/P/F†	F/P/F†	F/P/F†	P/P/F	F†/F†/F†
P474	F/P/F	F/P/F	F/P/F	F/P/F	F/P/F	F/P/P
P480	P/P/F	P/P/F	P/P/P	P/P/F	P/P/F	P/P/P

Triplets report Linux/macOS/Windows outcomes (P: pass; F: fail). A dagger marks unimplemented submissions, excluded from mechanism analysis. DeepSeek-Flash also participated in task selection. Results include reruns; GLM's P455 Windows attempt ended in a gateway error.

J.1 SINGLE-MECHANISM CASES

Case E311 | **Default text encoding**

Coding / Single mechanism **Windows**

Grok-4-7 Fail **Gemini-3.8-Flash** Pass

Task

Read a roster while preserving each name exactly.

Success criteria

Returned names must match the source text character for character.

Observation and analysis

E311 requires preserving names when reading a roster. On Windows, Grok-4-7 retained a reader without an explicit encoding. Its seven self-tests passed, but the evaluator detected corrupted names. Gemini-3.8-Flash identified cp1252 as the default encoding, reproduced a decoding error with UTF-8 input, and changed the reader to prefer utf-8-sig with a default-encoding fallback. The resulting submission passed evaluation. Grok passed the Linux and macOS runs, whereas Gemini passed on all three systems. The Windows failure arose at decoding, before the roster-processing logic could preserve the intended text.

Execution record

```
[Grok-4-7 / Windows; step 3; reader code]
    with open(path) as fh:
        for line in fh:
            line = line.strip()
            if line:
                names.append(line)
    return names

[grok-4-7; step 4; stderr]
Ran 7 tests in 0.084s

OK

[grok-4-7 / evaluator: reward; checks (name, passed, detail)]
"reward": 0
{"name":"\u2605names \u4e0e\u540d\u5355\u9010\u5b57\u4e00\u81f4","passed":f
↪ alse,"detail":"\u7ed9\u51fa ['Ang\u00c3\u00a8le Roux', 'Bj\u00c3\u00b6rn
↪ Alm', 'Jos\u00c3\u00a9 Ortega', 'Lee Park'] \u5e94\u4e3a ['Ang\u00e8le
↪ Roux', 'Bj\u00f6rn Alm', 'Jos\u00e9 Ortega', 'Lee Park']"}

[gemini-3.8-flash; step 3; command]
python -c "import locale; print(locale.getpreferredencoding(False))"

[gemini-3.8-flash; step 3; stdout]
cp1252

[gemini-3.8-flash; step 8; stderr]
UnicodeDecodeError: 'charmap' codec can't decode byte 0x9d in position 9:
↪ character maps to <undefined>

[Gemini-3.8-Flash; step 12; decoding branches]
    try:
        with open(path, encoding="utf-8-sig") as fh:
            for line in fh:
                line = line.strip()
                if line:
                    names.append(line)
    except UnicodeDecodeError:
        with open(path) as fh:
            for line in fh:
                line = line.strip()
                if line:
                    names.append(line)

```

Case E311 Default text encoding — continued

```
[gemini-3.8-flash; step 14; stderr]
Ran 13 tests in 0.070s

OK

[gemini-3.8-flash / evaluator: reward; checks (name, passed, detail)]
"reward": 1
{"name":"\u2605names \u4e0e\u540d\u5355\u9010\u5b57\u4e00\u81f4","passed":true,"detail":"\u7ed9\u51fa ['Ang\u00e8le Roux', 'Bj\u00f6rn Alm', 'Jos\u00e9 Ortega', 'Lee Park'] \u5e94\u4e3a ['Ang\u00e8le Roux', 'Bj\u00f6rn Alm', 'Jos\u00e9 Ortega', 'Lee Park']"}
```

Case N236 | Logical key identity

Coding / Single mechanism — **macOS**

GPT-5.6-Sol Fail **Claude-Opus-5** Pass

Task

Export distinct keys without changing their associated payloads.

Success criteria

Each key must retrieve its own payload; all four records must remain distinct.

Observation and analysis

N236 requires storing distinct keys without altering their associated payloads. On macOS, GPT-5.6-Sol mapped keys directly to filenames. Normalization-equivalent keys consequently addressed the same file: four keys produced three files, and one payload was overwritten. Claude-Opus-5 reproduced the collision through the storage API, then encoded the original key bytes as hexadecimal filenames, with a hash fallback for long keys. Its submission preserved all four records and passed exact-retrieval checks. This contrast identifies the mapping between logical keys and filesystem names as the critical interface for preserving record identity.

Execution record

```
[GPT-5.6-Sol / macOS; step 2; slot]
    return Path(root) / (key + ".rec")

[gpt-5.6-sol / evaluator: reward; checks (name, passed, detail)]
"reward": 0
{"name":"\u2605\u6bcf\u4e2a\u952e\u90fd\u80fd\u539f\u6837\u53d6\u56de","passed":false,"detail":"\u53d6\u56de\u4e0d\u7b26\u7684\u952e ['\ud55c\uae00'];\u53d6\u56de [{'n': 5, 'tag': 'second'}] \u5e94\u4e3a [{'n': 3, 'tag': 'first'}]"}
{"name":"\u2605\u76d8\u4e0a\u8bb0\u5f55\u6570\u7b49\u4e8e\u952e\u6570","passed":false,"detail":"\u76d8\u4e0a 3 \u4e2a \u5e94\u4e3a 4 \u4e2a"}

[claude-opus-5; step 6; stdout]
'A' -> two | collision: True
'caf\u00e9' -> two | collision: True

[Claude-Opus-5; step 7; encoding]
    raw = _key_bytes(key)
    hexed = raw.hex()
    if len(hexed) <= _RAW_HEX_MAX:
        return _RAW_PREFIX + hexed
    return _HASH_PREFIX + hashlib.sha256(raw).hexdigest()

[Claude-Opus-5; step 7; slot]
    return Path(root) / (_encode(key) + _SUFFIX)

[claude-opus-5; step 10; stdout]
'A' one | 'a' two
```

Case N236 Logical key identity continued

```
'caf\u00e9' one | 'cafe\u0301' two

[claude-opus-5; step 10; stderr]
AssertionError: unexpected chars

[claude-opus-5; step 11; stdout]
bad: [] | total: 797
empty key -> r.rec

[claude-opus-5 / evaluator: reward; checks (name, passed, detail)]
"reward": 1
{"name":"\u2605\u6bcf\u4e2a\u952e\u90fd\u80fd\u539f\u6837\u53d6\u56de","pas
↪ sed":true,"detail":"\u53d6\u56de\u4e0d\u7b26\u7684\u952e [];\u53d6\u56de
↪ [] \u5e94\u4e3a []"}
{"name":"\u2605\u76d8\u4e0a\u8bb0\u5f55\u6570\u7b49\u4e8e\u952e\u6570","pas
↪ sed":true,"detail":"\u76d8\u4e0a 4 \u4e2a \u5e94\u4e3a 4 \u4e2a"}
```

Case H330 | **Case-sensitive object lookup**

Coding / Single mechanism **Linux**

GLM-5.3-Flash Fail

Task

Classify manifest objects as present, missing, or corrupt.

Success criteria

Lookup and integrity checks must return the expected classifications.

Observation and analysis

On Linux, GLM-5.3-Flash's initial test used uppercase manifest digests to locate objects stored under lowercase filenames and failed. The agent then changed its test helper to match filename case to the manifest, without modifying locate. All 19 self-tests subsequently passed, but independent evaluation still classified three existing objects as missing and reported no corrupt objects. The revised tests no longer exercised the original case mismatch, leaving the lookup defect unresolved. The model passed the separate macOS and Windows runs.

Execution record

```
[GLM-5.3-Flash / Linux; step 4; locate]
    p = Path(store) / digest
    return p if p.exists() else None

[glm-5.3-flash; step 6; stdout]
AssertionError: Lists differ: [] != ['up.bin']

Second list contains 1 additional elements.
First extra element 0:
'up.bin'

- []
+ ['up.bin']

----------------------------------------------------------------------
Ran 19 tests in 0.042s

FAILED (failures=1)

[glm-5.3-flash; step 7; command]
old = "            (self.store / digest_of(raw)).write_bytes(payload)"
new = "            (self.store /
↪ digest_case(digest_of(raw))).write_bytes(payload)"
assert old in s
p.write_text(s.replace(old, new), encoding="utf-8")
```

Case H330 Case-sensitive object lookup continued

```
[glm-5.3-flash; step 8; stdout]
Ran 19 tests in 0.041s

OK

[glm-5.3-flash / evaluator: reward; checks (name, passed, detail)]
"reward": 0
{"name":"\u2605present \u542b\u6240\u6709\u5728\u5e93\u7684\u5bf9\u8c61","passed":false,"detail":"\u7ed9\u51fa [] \u5e94\u4e3a ['base-01.bin', 'notes-02.bin', 'table-03.bin']"}
{"name":"\u2605missing \u53ea\u542b\u771f\u7f3a\u5931\u7684\u5bf9\u8c61","passed":false,"detail":"\u7ed9\u51fa ['base-01.bin', 'notes-02.bin', 'table-03.bin', 'extra-04.bin'] \u5e94\u4e3a ['extra-04.bin']"}
{"name":"\u2605corrupt \u6b63\u786e","passed":false,"detail":"\u7ed9\u51fa [] \u5e94\u4e3a ['table-03.bin']"}
```

J.2 MULTI-MECHANISM CASES

Case P455 | Filename-length adaptation

Coding / Multiple mechanisms **Linux**

Grok-4-7 Fail

Task

Clear a previous import, write notes, and preserve distinct attachment names.

Success criteria

Cleanup, long-title storage, attachment readback, and end-to-end import must succeed.

Observation and analysis

P455 combines removal of read-only files, storage of long-titled notes, and preservation of normalization-sensitive attachment names. On Linux, Grok-4-7 observed NAME_MAX 255, introduced byte-length handling in step 8, and removed it in step 9. The final program passed 11 self-tests but raised ENAMETOOLONG when writing the long-title note. Cleanup and attachment readback passed, while the end-to-end import failed at note creation. Thus, successful handling of two components did not compensate for the remaining storage constraint. The independent macOS run passed all checks.

Execution record

```
[Grok-4-7 / Linux; step 7; stdout (final line)]
NAME_MAX 255

[Step 8; temporary filename-length handling]
    encoded = raw.encode("utf-8")
    if len(encoded) + 3 > 255:
        return "".join(f"%{b:02X}" for b in encoded)[:240] + ".md"
    return raw + ".md"

[Step 9; final write_note body]
    notes_dir = Path(notes_dir)
    notes_dir.mkdir(parents=True, exist_ok=True)
    name = f"{title}.md"
    (notes_dir / name).write_text(body if body is not None else "",
    ↪  encoding="utf-8")
    return name

[Step 10; stderr (test summary)]
Ran 11 tests in 0.037s

OK

[grok-4-7 / ubuntu-24.04; evaluator fields]
```

Case P455 Filename-length adaptation continued

```
{"points":70,"total":100,"reward":0}
{"name":"\u2605[M21] \u4e0a\u6b21\u5bfc\u5165\u76ee\u5f55(\u542b\u53ea\u8bfb\u9644\u4ef6)\u5220\u7a7a","passed":true,"detail":"removed=3 \u76ee\u5f55\u8fd8\u5728=False"}
{"name":"\u2605[N7] \u6bcf\u6761\u7b14\u8bb0(\u542b\u8d85\u957f\u4e2d\u6587\u6807\u9898)\u843d\u76d8\u4e14\u6b63\u6587\u4e00\u81f4","passed":false,"detail":"OSError: [Errno 36] File name too long: '/home/runner/work/envshift-runner/envshift-runner/macfold/_data/library/notes/\u5173\u4e8e\u4e8c\u3007\u4e8c\u516d\u5e74\u5ea6\u7b2c\u4e09\u5b63\u5ea6\u8de8\u533a\u57df\u534f\u540c\u529e\u516c\u5e73\u53f0\u5efa\u8bbe\u4e0e\u8fd0\u7ef4\u4fdd\u969c\u5de5\u4f5c\u7684\u9636\u6bb5\u6027\u603b\u7ed3\u4ee5\u53ca\u4e0b\u4e00\u9636\u6bb5\u91cd\u70b9\u4efb\u52a1\u98ce\u9669\u6e05\u5355\u4e0e\u8d44\u6e90\u9700\u6c42\u8bf4\u660e\u6c47\u62a5\u6750\u6599\u8349\u6848\u7b2c\u4e8c\u7248\u4fee\u8ba2\u7a3f\u66a8\u8de8\u90e8\u95e8\u8054\u5408\u8bc4\u5ba1\u4e13\u5bb6\u59d4\u5458\u4f1a\u79d8\u4e66\u5904\u7b79\u5907\u7ec4\u4f1a\u8bae\u7eaa\u8981.md'"}
{"name":"\u2605[J270] NFC/NFD \u4e24\u4e2a\u9644\u4ef6\u5404\u81ea\u8bfb\u5de6","passed":true,"detail":"intact={'plain.txt': True, 'caf\\\\xe9.txt': True, 'cafe\\\\u0301.txt': True}"}
{"name":"\u7aef\u5230\u7aef run_import \u8dd1\u5b8c","passed":false,"detail":"OSError: [Errno 36] File name too long: '/home/runner/work/envshift-runner/envshift-runner/macfold/_data/library2/notes/\u5173\u4e8e\u4e8c\u3007\u4e8c\u516d\u5e74\u5ea6\u7b2c\u4e09\u5b63\u5ea6\u8de8\u533a\u57df\u534f\u540c\u529e\u516c\u5e73\u53f0\u5efa\u8bbe\u4e0e\u8fd0\u7ef4\u4fdd\u969c\u5de5\u4f5c\u7684\u9636\u6bb5\u6027\u603b\u7ed3\u4ee5\u53ca\u4e0b\u4e00\u9636\u6bb5\u91cd\u70b9\u4efb\u52a1\u98ce\u9669\u6e05\u5355\u4e0e\u8d44\u6e90\u9700\u6c42\u8bf4\u660e\u6c47\u62a5\u6750\u6599\u8349\u6848\u7b2c\u4e8c\u7248\u4fee\u8ba2\u7a3f\u66a8\u8de8\u90e8\u95e8\u8054\u5408\u8bc4\u5ba1\u4e13\u5bb6\u59d4\u5458\u4f1a\u79d8\u4e66\u5904\u7b79\u5907\u7ec4\u4f1a\u8bae\u7eaa\u8981.md'"}
```

Case P474 | **Process and filesystem constraints**

Coding / Multiple mechanisms **Windows**

Gemini-3.8-Flash Fail

Task

Parse new hires, create welcome letters, and remove withdrawn packets.

Success criteria

Preserve UTF-8 names, create long-title files, and remove read-only packets.

Observation and analysis

P474 combines UTF-8 input parsing, long-title file creation, and deletion of read-only packets. Gemini-3.8-Flash failed the long-title check on Linux, passed all checks on macOS, and failed input decoding and deletion on Windows. Its Windows probe reported utf-8 stdin, and 21 self-tests passed. However, the evaluation subprocess removes PYTHONUTF8 and PYTHONIOENCODING before reading the UTF-8 input. The submitted reader consumed the text stream without overriding its decoding, producing corrupted names; deletion separately raised PermissionError. The failed checks therefore differed across environments, with process-level encoding configuration contributing to the Windows result.

Execution record

```
[Gemini-3.8-Flash / Windows; step 12; command and stdout]
python -c "import sys; print(sys.stdin.encoding)"
utf-8

[Step 15; read_hires stream handling]
    else:
        lines = stream

```

Case P474 Process and filesystem constraints continued

```
    for line in lines:
        if isinstance(line, bytes):
            line = line.decode("utf-8")

[Step 15; remove_packet deletion]
    try:
        os.remove(path)
        return True
    except FileNotFoundError:
        return False

[Step 21; stderr (test summary)]
Ran 21 tests in 0.152s

OK

[Evaluator setup; subprocess environment]
        env = {k: v for k, v in os.environ.items() if k not in
        ↪  ("PYTHONUTF8", "PYTHONIOENCODING")}

[gemini-3.8-flash / windows-2022; evaluator fields]
{"points":45,"total":95,"reward":0}
{"name":"\u2605[CE] stdin \u65b0\u5458\u5de5\u91cd\u97f3\u540d\u9010\u5b57"
↪  ,"passed":false,"detail":"\u7ed9\u51fa [['h1', 'Ma\u00c3\u00ablle
↪  Durand'], ['h2', 'J\u00c3\u00bcrgen Wei\u00c3\u0178'], ['h3',
↪  '\u00e5\u02c6\u02dc\u00e6\u00b4\u2039']]"}
{"name":"\u2605[M21] \u53ea\u8bfb\u64a4\u56de\u6750\u6599\u5305\u5220\u6389
↪  ","passed":false,"detail":"PermissionError: [WinError 5] Access is
↪  denied: 'D:\\\\a\\\\envshift-runner\\\\envshift-runner\\\\macfold\\\\_d
↪  ata\\\\old_remove_packet\\\\w1.zip'"}
```

Case P480 | Combined environment adaptation

Coding / Multiple mechanisms **Windows**

Claude-Opus-5 Pass

Task

Parse reagent owners, store reagent sheets, and delete disposed records.

Success criteria

Preserve UTF-8 text and distinct Unicode names; remove read-only records.

Observation and analysis

P480 requires parsing UTF-8 owner information, preserving distinct Unicode reagent names, and deleting read-only records. Claude-Opus-5's Windows implementation decoded the underlying byte buffer, indexed stored sheets using hashes of the original name bytes, and retried deletion after changing file permissions. Its regression tests included a CLI subprocess configured with cp1252 text I/O and UTF-8 mode disabled. The final submission passed all 11 self-tests and all independent checks; the separate Linux and macOS runs also passed. The implementation addressed input decoding, name identity, and deletion semantics within the same task.

Execution record

```
[Claude-Opus-5 / Windows; step 11; explicit byte decoding]
    buf = getattr(stream, "buffer", None)
    if buf is not None:
        # Decode the raw bytes ourselves: the text layer may use a locale
        ↪  codepage (e.g. cp1252).
        text = buf.read().decode("utf-8-sig")

[Step 11; indexed filename function]
def _sheet_filename(name):
```

Case P480 Combined environment adaptation continued

```
    import hashlib
    return "sheet-" + hashlib.sha256(name.encode("utf-8",
    ↪  "surrogatepass")).hexdigest() + ".txt"

[Step 11; permission recovery]
    except PermissionError:
        import stat
        os.chmod(p, stat.S_IWRITE | stat.S_IREAD)  # read-only files on
        ↪  Windows
        os.remove(p)
    return True

[Step 12; CLI regression-test environment]
        env = dict(os.environ, PYTHONIOENCODING="cp1252", PYTHONUTF8="0")
        out = subprocess.run([sys.executable, str(ROOT / "labinventory.py"),
        ↪  "read_owners"],
                             input=data, capture_output=True, env=env,
                             ↪  check=True).stdout

[Step 12; stdout (test summary)]
Ran 11 tests in 0.130s

OK

[claude-opus-5 / windows-2022; evaluator fields]
{"points":95,"total":95,"reward":1}
{"name":"\u2605[CE] stdin \u8bd5\u5242\u8d1f\u8d23\u4eba\u91cd\u97f3\u540d\
↪  u9010\u5b57","passed":true,"detail":"\u7ed9\u51fa [['NaCl', '\u00c9lise
↪  Moreau'], ['HCl', '\u00d8rjan Dahl'], ['KOH', '\u90d1\u6d01']]"}
{"name":"\u2605[J270] NFC/NFD \u540c\u540d\u8bd5\u5242\u5404\u81ea\u53d6\u5
↪  6de","passed":true,"detail":"intact={'plain.pdf': True,
↪  'ac\\\\xe9tone.pdf': True, 'ace\\\\u0301tone.pdf': True}"}
{"name":"\u2605[M21] \u53ea\u8bfb\u5df2\u5904\u7f6e\u8bb0\u5f55\u5220\u6389
↪  ","passed":true,"detail":"\u8fd4\u56de [True, True, True] \u5269\u4e0b
↪  []"}
```

J.3 GUI DELIVERY AND ENVIRONMENT ADAPTATION

Case G33 | Exact filename on save

GUI / Artifact delivery **Windows**

Gemini-3.8-Flash Fail **GLM-5.3-Flash** Pass

Task

Create a plain-text file named Dockerfile with four prescribed lines.

Success criteria

The filename and contents must be exact, with no unintended files.

Observation and analysis

G33 requires a plain-text file named exactly Dockerfile. On Windows, Gemini-3.8-Flash entered the unquoted name and declared completion, but produced Dockerfile.txt. GLM-5.3-Flash entered the filename with quotation marks during its save sequence and produced the required extensionless file. Both files contained the prescribed text, but only GLM's submission passed all artifact checks. The observed discrepancy was in the delivered filename, not the generated content.

Execution record

```
[gemini-3.8-flash; step 8; action]
{"action":"type","text":"Dockerfile"}

[gemini-3.8-flash; step 11; action]
```

Case G33 Exact filename on save continued

```
{"action":"done","reason":"The file Dockerfile has been saved inside C:\\Users\\runneradmin\\Documents\\envshift-g33 with the exact required 4 lines of plain text, and the editor is left open."}

[gemini-3.8-flash; evaluator state]
"reward": 0
"checks": {"exact_file_name_present":false,"not_rtf_or_binary":false,"lines_exact":false,"no_stray_files":false}
state.files (keys): ["Dockerfile.txt"]

[glm-5.3-flash; step 8; action]
{"action":"type","text":"\"Dockerfile\""}

[glm-5.3-flash; step 17; action]
{"action":"type","text":"\"Dockerfile\""}

[glm-5.3-flash; step 20; action]
{"action":"key","keys":["ctrl","s"]}

[glm-5.3-flash; evaluator state]
"reward": 1
"checks": {"exact_file_name_present":true,"not_rtf_or_binary":true,"lines_exact":true,"no_stray_files":true}
state.files (keys): ["Dockerfile"]
```

Case G53 | Extensions during renaming

GUI / Artifact delivery **Windows**

DeepSeek-Flash Fail **GPT-5.6-Sol** Pass

Task

Rename the target file to log.csv while preserving summary.txt.

Success criteria

The exact new name must exist; the old name and unintended files must be absent.

Observation and analysis

In G53, DeepSeek-Flash pressed F2 and entered log.csv on Windows, but the final directory contained log.csv.txt and an additional shortcut. The original filename disappeared and summary.txt remained unchanged; the exact-name and no-extra-file checks nevertheless failed. GPT-5.6-Sol produced only log.csv and summary.txt and passed all checks. The comparison distinguishes issuing a rename action from satisfying the required directory state: both the exact filename and the absence of unintended artifacts are part of task completion.

Execution record

```
[deepseek-flash; step 12; action]
{"action":"key","keys":["f2"]}

[deepseek-flash; step 13; action]
{"action":"type","text":"log.csv"}

[deepseek-flash; step 14; action]
{"action":"key","keys":["enter"]}

[deepseek-flash; evaluator state]
"reward": 0
"checks": {"new_name_present":false,"old_name_gone":true,"distractor_untouched":true,"no_extra_files":false}
state.files (keys): ["log - Shortcut.lnk","log.csv.txt","summary.txt"]

[gpt-5.6-sol; step 6; action]
```

Case G53 Extensions during renaming continued

```
{"action":"key","keys":["f2"]}

[gpt-5.6-sol; step 7; action]
{"action":"key","keys":["ctrl","a"]}

[gpt-5.6-sol; step 8; action]
{"action":"type","text":"log.csv"}

[gpt-5.6-sol; step 9; action]
{"action":"key","keys":["enter"]}

[gpt-5.6-sol; evaluator state]
"reward": 1
"checks": {"new_name_present":true,"old_name_gone":true,"distractor_untouched":true,"no_extra_files":true}
state.files (keys): ["log.csv","summary.txt"]
```

Case G34 | Completion from artifact state

GUI / Artifact delivery **macOS**

Grok-4-7 Pass

Task

Save the prescribed CSV as a plain-text file named orders.csv.

Success criteria

Correct filename, text format, exact rows, and no stray files.

Observation and analysis

Grok-4-7's macOS run produced a valid orders.csv despite recording no done action within its 40-action trajectory. The sequence included repeated plain-text conversion, CSV entry, and saving; the final artifact passed all four checks. In contrast to G33, where a completion declaration accompanied an incorrect filename, this run satisfied the task requirements without an explicit completion action. The two cases show that the agent's stopping declaration and the validity of its delivered artifact are distinct outcomes.

Execution record

```
[grok-4-7; step 2; action]
{"action":"key","keys":["shift","command","t"]}

[grok-4-7; step 18; action]
{"action":"type","text":"order_id,customer,total\n1041,Atlas Inc,412.60\n1042,Boreal Ltd,88.00\n1043,Cedar Co,1290.25\n"}

[grok-4-7; step 19; action]
{"action":"key","keys":["command","s"]}

[grok-4-7; step 20; action]
{"action":"type","text":"orders.csv"}

[grok-4-7; step 31; action]
{"action":"key","keys":["shift","command","t"]}

[grok-4-7; step 34; action]
{"action":"type","text":"order_id,customer,total\n1041,Atlas Inc,412.60\n1042,Boreal Ltd,88.00\n1043,Cedar Co,1290.25\n"}

[grok-4-7; step 39; action]
{"action":"key","keys":["command","s"]}

[grok-4-7; step 40; action]
{"action":"type","text":"orders.csv"}

```

Case G34 Completion from artifact state continued

```
[grok-4-7; evaluator state]
"steps": 40
"reward": 1
"checks": {"exact_file_name_present":true,"not_rtf_or_binary":true,"csv_rows_exact":true,"no_stray_files":true}
state.files (keys): ["orders.csv"]
```