DEVICESWORLD: BENCHMARKING CROSS-DEVICE AGENTS IN HETEROGENEOUS ENVIRONMENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Many real-world tasks require coordination among multiple independent devices, but existing device-control benchmarks primarily focus on a single interactive environment. Large-scale executable benchmarks for evaluating such cross-device coordination remain scarce. We introduce DevicesWorld, a large-scale executable benchmark for cross-device coordination in heterogeneous environments, comprising 5,894 tasks across mobile, desktop, and IoT environments, including 5,614 cross-device tasks. DevicesWorld provides a unified environment for cross-device interaction and evaluation, enabling agents to execute tasks across multiple independent devices with automatic evaluation. We evaluate 13 agent baselines on DevicesWorld-Lite; the best achieves a Task Success Rate of 40.00%, well below the human reference of 84.00%. To further examine how these failures relate to single-device subtask execution capabilities, we select 60 cross-device tasks from DevicesWorld-Lite, decompose them into 237 single-device subtasks, and conduct independent tests using three representative models, comparing the results with end-to-end execution on the same original tasks. The three models achieve subtask success rates of 79.75%–90.30%; however, for each model, at least two-thirds of the original tasks whose subtasks all pass independent tests still fail end to end. Combined with trajectory analysis, these findings highlight task planning and decomposition, cross-device information linking and task-progress tracking, and feedback-driven replanning as directions for improving end-to-end cross-device execution. By releasing DevicesWorld, we aim to provide a unified infrastructure for future research on cross-device agents.

1 INTRODUCTION

Agents based on vision-language models (VLMs) have been used to execute tasks in mobile applications, web environments, and desktop systems (Zhang et al., 2025b; Wang et al., 2024; Zheng et al., 2024; He et al., 2024; Zhang et al., 2025a). However, many real-world tasks require multiple independent devices to work together. As illustrated in Figure 1, preparing a guest room requires combining a visitor's schedule and preferences on a phone with reception rules on a computer, scheduling operations on home devices, saving a record on the computer, and sending a confirmation from the phone. Beyond operating individual devices, agents must integrate distributed information, track task progress, and coordinate execution across devices.

Existing executable device-control benchmarks primarily focus on a single device or environment. Even when tasks involve multiple applications, execution in these benchmarks is typically confined to a single phone, browser, or desktop system (Rawles et al., 2025; Zhou et al., 2024; Boisvert et al., 2024; Xie et al., 2024; Bonatti et al., 2025). Such settings offer limited support for directly evaluating cross-device task execution. CRAB (Xu et al., 2025a) extends executable evaluation to cross-environment tasks; its released CRAB Benchmark-v0 includes 18 cross-device tasks requiring joint execution across mobile and desktop environments (Table 1). This motivates a broader suite of executable cross-device tasks spanning diverse device combinations and application scenarios, alongside systematic investigation of whether single-device subtask execution capabilities reliably translate into successful completion of full cross-device tasks.

To support this evaluation need, we introduce DevicesWorld, a large-scale executable benchmark for cross-device coordination in heterogeneous environments. DevicesWorld comprises 5,894 tasks,

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.

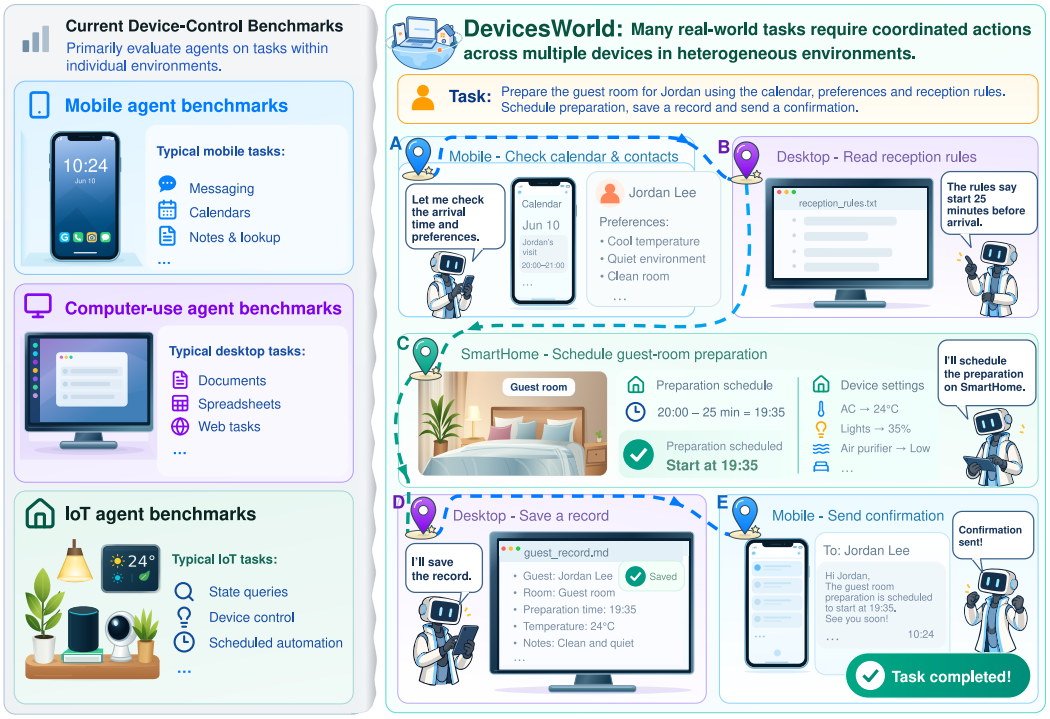


Figure 1: **DevicesWorld's evaluation motivation and an example cross-device task.** The left panel shows that existing device-control benchmarks primarily focus on a single interactive environment. The right panel presents a guest-room preparation workflow with five stages (A–E) spanning a phone, a desktop, and the SmartHome environment. Completing the task requires integrating information and coordinating execution across these environments.

including 5,614 cross-device tasks, spanning mobile, desktop, and Internet of Things (IoT) environments and supporting coordination among devices of the same or different types. Its unified interaction and evaluation environment enables agents to execute tasks across multiple independent devices toward a shared user goal, with automatic evaluation of task completion.

We evaluate 13 agent baselines on DevicesWorld-Lite; the best achieves a Task Success Rate of 40.00%, well below the human reference of 84.00%. We further compare independent single-device subtask tests with end-to-end execution on the corresponding original cross-device tasks (Section 4.3). For each of the three models tested, at least two-thirds of the original tasks whose single-device subtasks all pass independent tests still fail end to end. These paired results show that the single-device subtask execution capabilities demonstrated in independent tests do not yet reliably translate into successful cross-device task completion.

Our main contributions are as follows:

- **A large-scale cross-device task suite.** We construct DevicesWorld, comprising 5,894 tasks, including 5,614 cross-device tasks, covering diverse device combinations and application scenarios across mobile, desktop, and IoT environments and providing a task suite for large-scale executable cross-device evaluation.
- **A unified environment for cross-device interaction and evaluation.** We build an interactive environment connecting heterogeneous devices, enabling agents to execute cross-device tasks across multiple independent devices and undergo automatic evaluation. This provides unified support for evaluating the cross-device capabilities of different models and execution frameworks.
- **Task decomposition experiments and analysis.** We independently test single-device subtasks and compare their outcomes with end-to-end execution on the corresponding original cross-device tasks. The paired comparisons and trajectory analysis provide diagnostic evidence for improving task planning and coordinated execution in cross-device agents.

2 Related Work

Table 1 compares DevicesWorld with representative device-control benchmarks and related datasets.

Table 1: **Comparison of DevicesWorld with representative device-control benchmarks and related datasets.** Env. indicates whether an executable interactive environment is provided. Numbers in parentheses in the Cross-device column indicate the number of cross-device tasks.

Benchmark	Env.	#Tasks	# apps or websites	Platform	Interaction	Cross-device
GAIA	✗	466	–	–	–	✗
Mind2Web	✗	2,350	137	Web	GUI	✗
WebLINX	✗	2,337	155	Web	GUI	✗
AITW	✗	30,378	357+	Mobile	GUI	✗
AndroidControl	✗	15,283	833	Mobile	GUI	✗
OmniAct	✗	9,802	60+	Desktop	GUI	✗
HomeBench	✗	173,647	–	IoT	API	✗[3]
MiniWoB++	✓	125	1	Web	GUI	✗
WebArena	✓	812	6	Web	GUI	✗
AndroidWorld	✓	116	20	Mobile	GUI / API	✗
MobileWorld	✓	201	20	Mobile	GUI / API	✗
OSWorld[1]	✓	369	9	Desktop	GUI	✗
WindowsWorld	✓	181	17	Desktop	GUI	✗
OSWorld 2.0	✓	108	56	Desktop	GUI / CLI	✗
SimuHome	✓	600	–	IoT	API	✗[3]
CRAB	✓	120	25	Mobile + Desktop	GUI	✓ (18)
DevicesWorld (ours)	✓	**5,894**	**42**[2]	**Mobile + Desktop + IoT**	**GUI / CLI / API**	**✓ (5,614)**

[1] OSWorld uses the 369-task Ubuntu main set. The count of 9 follows the published application/OS-category count; additional Windows tasks are excluded.

[2] For DevicesWorld, 42 combines 25 distinct applications and 17 website categories.

[3] Cross-device requires task-dependent interaction across at least two independent device environments. Multiple IoT endpoints controlled within the same SmartHome environment are counted as one environment.

2.1 Device-Control Benchmarks in Individual Environments

Device-control benchmarks primarily evaluate instruction understanding and interaction within a single environment (Kapoor et al., 2025; Xu et al., 2025b; Yao et al., 2022). AITW (Rawles et al., 2023), AndroidControl (Li et al., 2024), Mind2Web (Deng et al., 2023), and WebLINX (Lu et al., 2024) use human interaction trajectories to evaluate action prediction, navigation, and cross-task generalization in mobile or web environments. MiniWoB++ (Liu et al., 2018), AndroidWorld (Rawles et al., 2025), WebArena (Zhou et al., 2024), OSWorld (Xie et al., 2024), and Windows Agent Arena (Bonatti et al., 2025) provide executable environments where agents interact based on execution feedback, and task completion is evaluated from the resulting environment states.

Research has further expanded to long-horizon and cross-application workflows (Mialon et al., 2024; Trivedi et al., 2024; Cao et al., 2024). GUI Odyssey (Lu et al., 2025) and MobileWorld (Kong et al., 2026) study cross-application execution and context transfer on mobile devices, while VisualWebArena (Koh et al., 2024) incorporates visual understanding into web interaction. WorkArena (Drouin et al., 2024) and WorkArena++ (Boisvert et al., 2024) cover enterprise knowledge work and compositional planning, whereas WindowsWorld (Li et al., 2026a) and OSWorld 2.0 (Yuan et al., 2026) introduce professional workflows, real-world task materials, dynamic environments, cross-source information, and multimodal processing.

Smart-home and IoT benchmarks extend evaluation to dynamic automation with temporal dependencies, covering state queries, device control, reasoning, scheduling, and verifiable execution (Li et al., 2025; Seo et al., 2026; Li et al., 2026b). Although a smart-home environment may contain

multiple IoT endpoints, we treat it as a single interaction environment when these endpoints share a unified environment state and control interface. Overall, these benchmarks already support complex workflows within individual environments. DevicesWorld examines an additional dimension: connecting information, operations, and intermediate results across independent device environments toward a shared user goal, rather than merely increasing task length or application count.

2.2 Cross-Device Interaction and Cross-Environment Agents

MMBench-GUI (Wang et al., 2026) evaluates multi-platform GUI agents on interface understanding, element grounding, task automation, and cross-application collaboration, but cross-platform adaptation differs from coordinating independent devices within the same task. CRAB (Xu et al., 2025a) constructs cross-environment workflows through subtask composition and uses graph-based evaluation to track intermediate states and task completion progress.

For execution architectures, UFO^3 (Zhang et al., 2026) organizes multi-device execution through dynamic task graphs with data and control dependencies, while H-RePlan (Yao et al., 2026) distinguishes within-device recovery from cross-device replanning to address execution failures at different levels. Building on these efforts, DevicesWorld expands the scale and coverage of executable cross-device tasks across mobile, desktop, and IoT environments, with unified execution and outcome verification.

3 DevicesWorld

3.1 Benchmark Overview

DevicesWorld evaluates cross-device task execution across mobile, desktop, and IoT environments. Agents must acquire and process information from different devices toward a shared user goal and coordinate operations according to task progress and cross-device dependencies. The benchmark contains 5,894 tasks, including 5,614 cross-device tasks, covering coordination among devices of the same or different types.

As shown in Figure 2(a), desktop–mobile combinations account for the largest share of the full task set at 31.83%, while 16.46% of tasks involve all three environment types. Tasks cover to-do management, note-taking, communication, and calendar operations on mobile; document and structured-data editing, code processing, web interaction, and image and audio processing on desktop; and device-state queries, immediate control, and scheduled operations in IoT. Figure 2(b) reports the number of tasks involving each mobile and desktop application.

Each task begins by initializing the participating device environments and resources according to its configuration. The agent uses the task instruction, current observations, and interaction history to decide which device to act on and what action to take, then continues execution based on feedback. When the agent stops or reaches an execution limit, the verifier checks the relevant device states and task outputs against the task-specific completion conditions. Interaction interfaces are detailed in Appendix A, and experimental configurations and evaluation metrics are specified in Section 4.1.

3.2 Task Construction

Constructing cross-device tasks poses two key challenges. First, a task must establish meaningful cross-device dependencies around a shared user goal rather than simply concatenate independent single-device operations. Different devices must play necessary and interdependent roles, with information or states on one device affecting subsequent decisions and operations on another. Second, tasks must remain executable across heterogeneous environments. Mobile, desktop, and SmartHome environments differ in applications, state representations, resource formats, and interaction interfaces. A semantically reasonable task design must therefore be instantiated into an executable task whose initial states, required operations, and evaluation conditions are consistent with the task requirements. To address these challenges, we organize task construction into four stages: configuration sampling, task design, executable instantiation, and validation and repair. Task design establishes device roles and cross-device dependencies, linking required results to visible sources and task rules. Executable instantiation creates the corresponding resources, initial states, and comple-

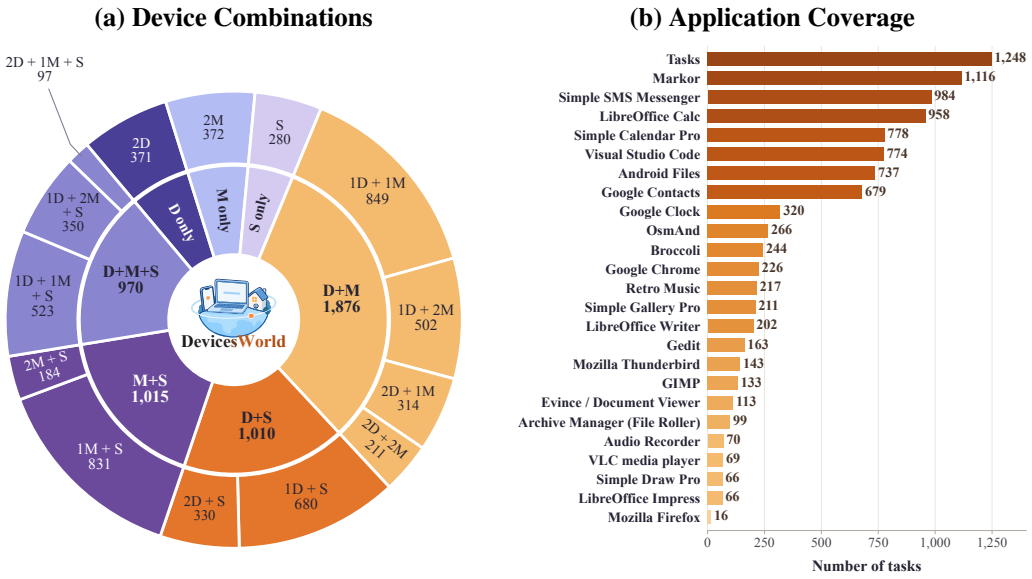


Figure 2: **Device combinations and application coverage in DevicesWorld.** (a) The inner ring represents combinations of participating environment types, while the outer ring further distinguishes the numbers of concrete devices. D, M, and S denote desktop devices, mobile devices, and the SmartHome environment, respectively. (b) The number of tasks involving each application on mobile and desktop. A task may involve multiple applications, so application-level task counts are not mutually exclusive; HTML resources are counted separately.

tion checks. Validation examines resource consistency and task initialization, and tests whether the completion checks distinguish correct from incomplete or incorrect outcomes. Detected issues are returned to the relevant stages for repair (Appendix B).

In addition to automated checks, we conducted manual quality control on DevicesWorld-Lite, the subset used for the main experiments (Section 4.1). Ten participants spent approximately 400 hours reviewing tasks and conducting execution tests in the benchmark environments. The review focused on whether user goals and device roles were reasonable, whether the intended cross-device dependencies held, and whether tasks could be completed in the corresponding environments. Identified issues were addressed through revisions to task designs, resources, or execution components, followed by rechecking.

3.3 Unified Cross-Device Environment

To support task execution, we build a multi-device interactive environment spanning mobile, desktop, and IoT. It supports task-specified combinations of different device types and multiple instances of the same type. Each device has a unique identifier and an independent task state. A unified controller connects the device interfaces and routes actions by device identifier, while checking compatibility with the target interface.

We extend the mobile and desktop environments for cross-device tasks based on AndroidWorld (Rawles et al., 2025) and OSWorld (Xie et al., 2024), respectively. The mobile environment runs Android, provides screenshots and UI element information, and supports GUI actions. The desktop environment runs Linux, provides screenshots, and supports GUI and command-line actions. The IoT component is implemented through our SmartHome simulation environment, which covers 11 device categories, including lighting, temperature regulation, air treatment, and household appliances. SmartHome supports state and capability queries, immediate control, time advancement, and scheduled operations, with configurable observation exposure. Environment configurations, observation and action interfaces, and SmartHome device categories are detailed in Appendix A; the observation configuration used in our experiments is specified in Section 4.1.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Evaluation Tasks. We select 200 tasks from DevicesWorld to form a fixed evaluation subset, DevicesWorld-Lite, for the main experiments. Task selection considers major device combinations, application scenarios, and task requirements. All baselines use the same task set, initialization configuration, and outcome verification mechanism.

Baselines. We evaluate 13 baseline settings, including direct-interaction settings for 11 models and two agent frameworks, CRAB-MABE[1] (Xu et al., 2025a) and UFO^3 (Zhang et al., 2026), both using GPT-5.5 (OpenAI, 2026a) as the base model. The direct-interaction models include GUI-optimized vision-language models GUI-Owl (Xu et al., 2026) and UI-Venus (Team et al., 2026). The complete list of baselines is provided in Table 2.

Interaction Protocol. Each complete task run is limited to at most 50 agent steps and 1,800 seconds for all agent baselines. Direct-interaction models use a unified observation and history configuration. At each step, the input includes the task instruction, currently visible cross-device observations and screenshots, execution feedback, and the textual interaction history from the most recent 10 steps; historical screenshots are not included. For SmartHome, current structured observations are provided automatically at each step. This avoids repeated state-query steps and reliance on earlier query results in the limited interaction history (Appendix A.1).

Metrics. DevicesWorld distinguishes scoring checks, which measure completion of target outcomes, from non-scoring checks, which enforce mandatory constraints. Task success requires all evaluation checks to pass, whereas the partial score is computed only from scoring checks. For example, in a program-repair task, the original test script must remain unmodified so that an agent cannot pass by changing the tests instead of repairing the program. Satisfying this constraint contributes no score, because leaving the script unchanged does not by itself indicate progress on the repair.

For task T, let E_T denote the set of all evaluation checks and $E_T^{\text{score}} \subseteq E_T$ its scoring subset. Let $b_e \in \{0, 1\}$ indicate whether check e passes. The task-success outcome y and partial score ρ are defined as

$$y = \prod_{e \in E_T} b_e, \qquad \rho = \frac{\sum_{e \in E_T^{\text{score}}} b_e}{|E_T^{\text{score}}|}.$$

If all scoring checks pass but at least one non-scoring check fails, then $\rho = 1$ while $y = 0$.

Task Success Rate is the proportion of evaluated tasks with $y = 1$, and Mean Score is the equally weighted average of ρ across tasks. We also report agent steps (Steps), execution time (Duration), and token usage (Tokens per Task), averaged over all evaluated tasks. Token usage includes input and output tokens from all model calls within a task run. Budget Exhaustion Rate is the proportion of the 200 task runs terminated by reaching the step or time limit.

Human Performance. We recruit five computer science undergraduates for human execution tests on the same 200-task evaluation set. We report Task Success Rate and the average number of recorded steps over all tasks.

4.2 MAIN RESULTS AND FAILURE ANALYSIS

On DevicesWorld-Lite, GPT-6 achieves the highest Task Success Rate among the 13 agent baselines at 40.00%, well below the human reference of 84.00% (Table 2). GPT-6 (OpenAI, 2026b) outperforms GPT-5.5 by 11 percentage points, with a similar average step count and fewer tokens per task. With GPT-5.5 as the base model, CRAB-MABE and UFO^3 achieve success rates of 27.50% and

[1]CRAB-MABE denotes our adaptation of CRAB's Multi-Agent by Environment architecture (Xu et al., 2025a) to DevicesWorld, referred to as *By Env* in the original paper. A main agent performs overall planning, while environment-specific subagents execute actions based on assigned instructions and observations from their respective environments.

Table 2: **Main results on DevicesWorld-Lite.** All 13 agent baselines and the human reference are evaluated on the same 200 tasks. Success and Mean Score denote Task Success Rate and average partial score, respectively. Budget Exh. is computed over all 200 tasks. Framework baselines indicate their base models in parentheses. "—" denotes metrics that are not reported or not applicable.

Baseline	Success (%) ↑	Mean Score ↑	Steps	Dur. (min)	Budget Exh. (%) ↓	Tok./Task ↓
UI-Venus-1.5-30B-A3B	6.00	0.1884	44.35	6.37	86.50	650,336
GUI-Owl-1.5-32B-Instruct	10.00	0.2903	40.42	7.03	78.00	603,200
Qwen3.7-Plus	19.50	0.4248	28.70	15.42	58.50	463,452
Gemini 3.1 Pro Preview	20.50	0.4325	25.29	12.00	37.50	352,704
Kimi K2.6	21.50	0.4571	19.59	20.26	52.00	344,600
Doubao-Seed-2.0-Pro	23.00	0.4513	27.99	19.50	58.50	412,300
DeepSeek-V4.1-Flash	36.00	0.5848	26.57	6.22	34.50	428,719
Claude Opus 4.8	37.50	0.5845	20.45	10.23	27.50	476,284
Claude Fable 5.1	39.50	0.5946	22.32	12.26	27.00	568,596
GPT-5.5	29.00	0.5358	21.00	12.40	25.50	343,897
GPT-6	**40.00**	**0.6306**	20.60	6.93	**20.00**	316,743
UFO^3 (GPT-5.5)	25.50	0.4664	26.11	13.90	31.50	**277,222**
CRAB-MABE (GPT-5.5)	27.50	0.4947	27.81	19.29	42.50	577,093
Human reference	84.00	—	21.34	—	—	—

25.50%, respectively, both below the 29.00% achieved by direct interaction. Both frameworks use more agent steps on average; compared with direct interaction, token usage increases for CRAB-MABE but decreases for UFO^3. Figure 3 shows that GPT-6 fails with positive scores on 42.50% of tasks, compared with 17.50% of tasks that fail with zero scores. The same pattern holds for all direct-interaction models except GUI-Owl and UI-Venus, showing that progress on scoring checks often falls short of task success.

To better understand failures in cross-device tasks, we analyze the agents' execution trajectories and observe errors in cross-device information association and target-device localization, omissions of required actions, and unproductive interaction loops. For example, in dinner preparation, the agent writes `tofu`, an ingredient in the computer-side rules, as the recipe name in a to-do item instead of matching a recipe on the phone (Appendix C.1.1); in humidification scheduling, it completes the schedule but searches the first phone for a to-do item located on the second, leaving the update unfinished (Appendix C.1.2). In the climate control task, the agent reads the rules and queries device capabilities, then saves a JSON result marked `applied`. However, it terminates without performing the required control operations, leaving the relevant air conditioners and heaters off (Appendix C.2). Unproductive interaction loops also expose limitations in dynamic replanning: in calendar synchronization, despite having generated subtasks, the agent repeatedly attempts to retrieve meeting information without making progress, yet neither consults the Linux schedule for search cues nor revises its cross-device execution plan. After the initial planning step, all 49 subsequent steps are spent on the phone calendar, and execution ends in the monthly view before the computer-side rule-based schedule updates and change-log generation begin (Appendix C.3).

4.3 Task Decomposition Study

To examine the relationship between cross-device task failures and single-device subtask execution capabilities, we select 60 tasks from DevicesWorld-Lite and decompose them into 237 single-device subtasks with explicit goals according to their execution devices and task dependencies. We conduct independent tests using Gemini 3.1 Pro Preview (Google DeepMind, 2026), GPT-5.5 (OpenAI, 2026a), and Claude Opus 4.8 (Anthropic, 2026). For subtasks that depend on preceding results, we provide the necessary information that would be obtained after the preceding steps are completed correctly and initialize the relevant environments to the corresponding states. Agents must still read or process information, or perform device operations, as required to complete the current subtask. Task selection, decomposition rules, initialization procedures, and execution configurations are detailed in Appendix D.

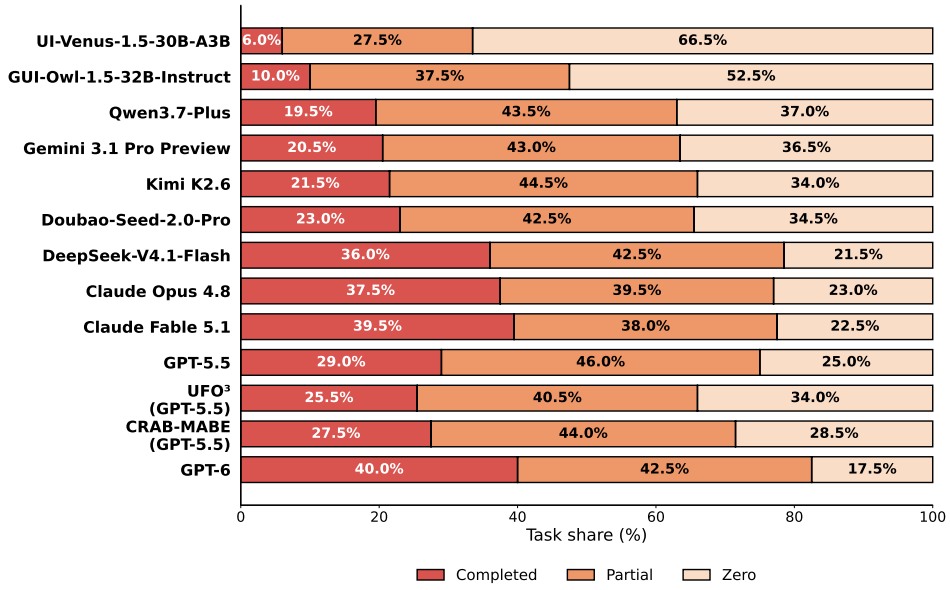


Figure 3: **Task outcomes on DevicesWorld-Lite.** *Completed* denotes task success, requiring all evaluation checks to pass; *Partial* denotes task failure with a positive score; and *Zero* denotes task failure with a score of zero. For most baselines, failures with positive scores are more common than zero-score failures.

We compare three metrics. **End-to-End Success Rate (E2E)** denotes the proportion of original tasks that succeed in end-to-end execution. **All-Subtask Success Rate (All-Subtask)** denotes the proportion of original tasks for which all constituent subtasks pass independent testing. **Subtask Success Rate (Subtask)** denotes the pass rate of independently tested subtasks. E2E and All-Subtask are both computed over the same 60 original tasks, while Subtask is computed over the 237 subtasks.

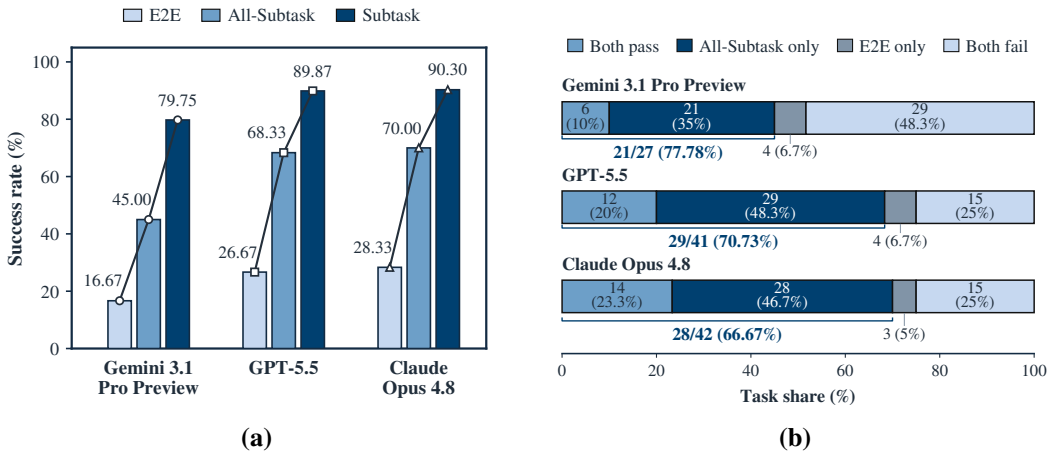


Figure 4: **Comparison between single-device subtask execution and end-to-end cross-device task completion.** (a) E2E and All-Subtask are computed over the same 60 original tasks, while Subtask is computed over the 237 independently tested single-device subtasks. (b) Paired comparison of All-Subtask and E2E outcomes on the same original tasks for each model. Bar segments are expressed as percentages of the 60 original tasks; the fractions below each bar report the E2E failure rate among tasks whose subtasks all pass independently. Even when all constituent single-device subtasks pass independently, a substantial fraction of the corresponding original cross-device tasks still fail in end-to-end execution.

Agents exhibit strong execution capabilities on single-device subtasks, while their end-to-end success rates on cross-device tasks remain low. As shown in Figure 4(a), under independent testing with explicit goals and satisfied prerequisites, the three models achieve Subtask success rates ranging from 79.75% to 90.30%, with GPT-5.5 and Claude Opus 4.8 each independently completing approximately 90% of the subtasks. On the same 60 original tasks, the All-Subtask rates of Gemini, GPT, and Claude reach 45.00%, 68.33%, and 70.00%, respectively, exceeding their corresponding E2E success rates by 28.33, 41.67, and 41.67 percentage points. The All-Subtask results further show that, under these independent testing conditions, the evaluated models can separately complete all constituent subtasks of many original tasks, rather than only some of their components.

Even when all subtasks pass independent testing, the original task still frequently fails end to end. We further conduct a paired comparison of All-Subtask and E2E outcomes for the same model on the same original tasks, as shown in Figure 4(b). Among the original tasks for which all subtasks pass independently, 77.78%, 70.73%, and 66.67% still fail end to end for Gemini, GPT, and Claude, respectively. In contrast, each model has only 3–4 original tasks that succeed end to end while not all of their subtasks pass independently. These paired results show that the single-device subtask capabilities demonstrated under independent testing do not yet reliably translate into end-to-end cross-device success. Combined with the trajectory analysis in Section 4.2, these findings point to three directions for improving cross-device execution: decomposing tasks at an appropriate granularity and planning their cross-device execution; maintaining critical information, its associations with task entities and device instances, and actual completion states across subtasks; and adapting both local execution strategies and the overall task plan to execution feedback.

5 Discussion

DevicesWorld currently supports cross-device evaluation across mobile, desktop, and IoT environments. Future work could incorporate community feedback to expand its device and application coverage and include tasks involving more devices, longer time horizons, and more complex cross-device dependencies. Beyond the current SmartHome simulation, cross-device execution could also be evaluated with physical IoT devices under varying operating conditions.

To improve end-to-end cross-device execution, future research could explore how to decompose user goals into subtasks with clear objectives and suitable granularity, and determine their execution devices and order according to device capabilities and task dependencies. During execution, further work could investigate how to retain and correctly use critical information, associate it with the appropriate task entities and device instances, and maintain an up-to-date view of task progress. This includes keeping track of completed and pending subtasks across device switches, so that completing a local objective does not lead the agent to overlook unfinished requirements. Feedback-driven replanning could help agents break out of unproductive interaction loops and try alternative strategies for completing single-device subtasks. At the overall task level, agents could revise their solution strategy and reorganize the remaining cross-device workflow based on execution outcomes and task progress.

6 Conclusion

We introduce DevicesWorld, a large-scale executable benchmark for cross-device coordination across mobile, desktop, and IoT environments. It comprises 5,894 tasks covering diverse device combinations and application scenarios and provides a unified environment for cross-device interaction and automatic evaluation. Among the 13 agent baselines evaluated on DevicesWorld-Lite, the highest Task Success Rate is 40.00%, well below the human reference of 84.00%. These results indicate substantial room for improvement in end-to-end cross-device task execution. To further examine how these failures relate to single-device subtask capabilities, we conduct a task decomposition study. The results show that models frequently fail full cross-device tasks even when all constituent single-device subtasks pass independent tests. Combined with trajectory analysis, these findings motivate research on task planning and decomposition, cross-device information linking and task-progress tracking, and feedback-driven replanning. We hope DevicesWorld will support the development and systematic evaluation of more reliable cross-device agents.

AI USE STATEMENT

We used generative AI to assist with benchmark task construction, coding, literature search, and manuscript preparation. In benchmark construction, generative AI was used to support the generation and refinement of task specifications, task resources, and related evaluation components, as described in Appendix B. We also used AI to assist coding, identify and retrieve relevant and recent literature, and support manuscript preparation, including the refinement of author-written text and figures. All AI-generated or AI-assisted content incorporated into this work was carefully reviewed and verified by the authors before inclusion. The authors take full responsibility for all contents and materials to which AI tools contributed and that were incorporated into the paper or benchmark.

ETHICS STATEMENT

We considered the ethical implications of human participation and cross-device agent evaluation in this work. The human-reference experiments involved five computer science undergraduates who performed benchmark tasks in controlled evaluation environments. All participants were informed about the experiments and agreed to participate. DevicesWorld uses synthetic task data and controlled benchmark environments, and does not contain personal data collected from real users. While cross-device agents may raise privacy and security concerns when deployed on real-world devices, our work focuses on controlled benchmarking and does not involve deployment on users' personal devices.

REPRODUCIBILITY STATEMENT

Details of the benchmark environments, interaction interfaces, task construction procedure, evaluation protocol, and experimental settings are provided throughout the paper and appendices. The experimental configurations and implementation details are documented in detail to facilitate reproduction of the reported results. We will release DevicesWorld, including the benchmark tasks, benchmark environments, and related resources, to support reproducible evaluation and future research.

REFERENCES

Anthropic. Claude Opus 4.8 system card, 2026. URL https://www.anthropic.com/claude-opus-4-8-system-card.

Léo Boisvert, Megh Thakkar, Maxime Gasse, Massimo Caccia, Thibault Le Sellier De Chezelles, Quentin Cappart, Nicolas Chapados, Alexandre Lacoste, and Alexandre Drouin. Workarena++: Towards compositional planning and reasoning-based common knowledge work tasks. In *Advances in Neural Information Processing Systems*, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/0b82662b6c32e887bb252a74d8cb2d5e-Paper-Datasets_and_Benchmarks_Track.pdf.

Rogerio Bonatti, Dan Zhao, Francesco Bonacci, Dillon Dupont, Sara Abdali, Yinheng Li, Yadong Lu, Justin Wagle, Kazuhito Koishida, Arthur Bucker, Lawrence Keunho Jang, and Zheng Hui. Windows agent arena: Evaluating multi-modal OS agents at scale. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=W9s817KqYf.

Ruisheng Cao, Fangyu Lei, Haoyuan Wu, Jixuan Chen, Yeqiao Fu, Hongcheng Gao, Xinzhuang Xiong, Hanchong Zhang, Yuchen Mao, Wenjing Hu, Tianbao Xie, Hongshen Xu, Danyang Zhang, Sida Wang, Ruoxi Sun, Pengcheng Yin, Caiming Xiong, Ansong Ni, Qian Liu, Victor Zhong, Lu Chen, Kai Yu, and Tao Yu. Spider2-v: How far are multimodal agents from automating data science and engineering workflows? In *Advances in Neural Information Processing Systems*, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/c2f71567cd53464161cab3336e8fc865-Paper-Datasets_and_Benchmarks_Track.pdf.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web. In *Advances in Neural Information Processing Systems*, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/5950bf290a1570ea401bf98882128160-Paper-Datasets_and_Benchmarks.pdf.

Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H. Laradji, Manuel Del Verme, Tom Marty, David Vazquez, Nicolas Chapados, and Alexandre Lacoste. Workarena: How capable are web agents at solving common knowledge work tasks? In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=BRfqYrikdo.

Google DeepMind. Gemini 3.1 Pro model card, 2026. URL https://deepmind.google/models/model-cards/gemini-3-1-pro/.

Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. WebVoyager: Building an end-to-end web agent with large multimodal models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024. URL https://aclanthology.org/2024.acl-long.371/.

Raghav Kapoor, Yash Parag Butala, Melisa Russak, Jing Yu Koh, Kiran Kamble, Waseem AlShikh, and Ruslan Salakhutdinov. Omniact: A dataset and benchmark for enabling multimodal generalist autonomous agents for desktop and web. In *Computer Vision – ECCV 2024*, 2025. URL https://doi.org/10.1007/978-3-031-73113-6_10.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Russ Salakhutdinov, and Daniel Fried. VisualWebArena: Evaluating multimodal agents on realistic visual web tasks. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024. URL https://aclanthology.org/2024.acl-long.50/.

Quyu Kong, Xu Zhang, Zhenyu Yang, Nolan Gao, Chen Liu, Panrong Tong, Chenglin Cai, Hanzhang Zhou, Jianan Zhang, Liangyu Chen, Zhidan Liu, Steven Hoi, and Yue Wang. MobileWorld: Benchmarking autonomous mobile agents in agent-user interactive and MCP-augmented environments. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2026. URL https://aclanthology.org/2026.acl-long.278/.

Jinchao Li, Yunxin Li, Chenrui Zhao, Zhenran Xu, Baotian Hu, and Min Zhang. WindowsWorld: A process-centric benchmark of autonomous GUI agents in professional cross-application environments. In *Findings of the Association for Computational Linguistics: ACL 2026*, 2026a. URL https://aclanthology.org/2026.findings-acl.750/.

Kuan Li, Shuo Zhang, Huacan Wang, Fangzhou Yu, Zecheng Sheng, Yi Gu, Weipeng Ming, Lei Xue, Chen Liu, Sen Hu, Ronghao Chen, Siyue Lin, Yuqing Hou, Xiaofeng Mou, and Yi Xu. Smh-bench: Benchmarking llm agents for environment-grounded reasoning and action in smart homes, 2026b. URL https://arxiv.org/abs/2606.01912.

Silin Li, Yuhang Guo, Jiashu Yao, Zeming Liu, and Haifeng Wang. $homebench$: Evaluating LLMs in smart homes with valid and invalid instructions across single and multiple devices. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2025. URL https://aclanthology.org/2025.acl-long.597/.

Wei Li, William Bishop, Alice Li, Chris Rawles, Folawiyo Campbell-Ajala, Divya Tyamagundlu, and Oriana Riva. On the effects of data scale on ui control agents. In *Advances in Neural Information Processing Systems*, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/a79f3ef3b445fd4659f44648f7ea8ffd-Paper-Datasets_and_Benchmarks_Track.pdf.

Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, Tianlin Shi, and Percy Liang. Reinforcement learning on web interfaces using workflow-guided exploration. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=ryTp3f-0-.

Quanfeng Lu, Wenqi Shao, Zitao Liu, Lingxiao Du, Fanqing Meng, Boxuan Li, Botong Chen, Siyuan Huang, Kaipeng Zhang, and Ping Luo. Guiodyssey: A comprehensive dataset for cross-app gui navigation on mobile devices. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2025. URL https://openaccess.thecvf.com/content/ICCV2025/html/Lu_GUIOdyssey_A_Comprehensive_Dataset_for_Cross-App_GUI_Navigation_on_Mobile_ICCV_2025_paper.html.

Xing Han Lu, Zdeněk Kasner, and Siva Reddy. WebLINX: Real-world website navigation with multi-turn dialogue. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=mUSPhG4uDW.

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/25ae35b5b1738d80f1f03a8713e405ec-Paper-Conference.pdf.

OpenAI. GPT-5.5 system card, 2026a. URL https://deploymentsafety.openai.com/gpt-5-5.

OpenAI. GPT-6 Astra system card, 2026b. URL https://deploymentsafety.openai.com/gpt-6-astra.

Chris Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, Daniel Toyama, Robert Berry, Divya Tyamagundlu, Timothy Lillicrap, and Oriana Riva. Androidworld: A dynamic benchmarking environment for autonomous agents. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/01a83bc2f2732a58e6aa731e659e7101-Paper-Conference.pdf.

Christopher Rawles, Alice Li, Daniel Rodriguez, Oriana Riva, and Timothy Lillicrap. Android in the wild: A large-scale dataset for android device control. In *Advances in Neural Information Processing Systems*, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/bbbb6308b402fe909c39dd29950c32e0-Paper-Datasets_and_Benchmarks.pdf.

Gyuhyeon Seo, Jungwoo Yang, Junseong Pyo, Nalim Kim, Jonggeun Lee, and Yohan Jo. Simuhome: A temporal- and environment-aware benchmark for smart home LLM agents. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=LCS1WsGvha.

Venus Team, Changlong Gao, Zhangxuan Gu, Yulin Liu, Xinyu Qiu, Shuheng Shen, Yue Wen, Tianyu Xia, Zhenyu Xu, Zhengwen Zeng, Beitong Zhou, Xingran Zhou, Weizhi Chen, Sunhao Dai, Jingya Dou, Yichen Gong, Yuan Guo, Zhenlin Guo, Feng Li, Qian Li, Jinzhen Lin, Yuqi Zhou, Linchao Zhu, Liang Chen, Zhenyu Guo, Changhua Meng, and Weiqiang Wang. Ui-venus-1.5 technical report, 2026. URL https://arxiv.org/abs/2602.09082.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024. URL https://aclanthology.org/2024.acl-long.850/.

Junyang Wang, Haiyang Xu, Jiabo Ye, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. Mobile-agent: Autonomous multi-modal mobile device agent with visual perception. In *ICLR 2024 Workshop on Large Language Model (LLM) Agents*, 2024. URL https://openreview.net/forum?id=jE6pDYCnVF.

Xuehui Wang, Zhenyu Wu, JingJing Xie, Zichen Ding, Bowen Yang, Zehao Li, Zhaoyang Liu, Qingyun Li, Xuan Dong, Zhe Chen, Weiyun Wang, Xiangyu Zhao, Jixuan Chen, Haodong Duan, Tianbao Xie, Chenyu Yang, Shiqian Su, Yue Yu, Yanting Zhang, Xiangyu Yue, Weijie Su, Xizhou Zhu, Wei Shen, Jifeng Dai, and Wenhai Wang. Mmbench-gui: A unified hierarchical evaluation framework for multi-platform gui agents. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2026. URL https://openaccess.thecvf.com/content/CVPR2026/html/Wang_MMBench-GUI_A_Unified_Hierarchical_Evaluation_Framework_for_Multi-Platform_GUI_Agents_CVPR_2026_paper.html.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *Advances in Neural Information Processing Systems*, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/5d413e48f84dc61244b6be550f1cd8f5-Paper-Datasets_and_Benchmarks_Track.pdf.

Haiyang Xu, Xi Zhang, Haowei Liu, Junyang Wang, Zhaozai Zhu, Shengjie Zhou, Xuhao Hu, Feiyu Gao, Junjie Cao, Zihua Wang, Zhiyuan Chen, Jitong Liao, Qi Zheng, Jiahui Zeng, Ze Xu, Shuai Bai, Junyang Lin, Jingren Zhou, and Ming Yan. Mobile-agent-v3.5: Multi-platform fundamental gui agents, 2026. URL https://arxiv.org/abs/2602.16855.

Tianqi Xu, Linyao Chen, Dai-Jie Wu, Yanjun Chen, Zecheng Zhang, Xiang Yao, Zhiqiang Xie, Yongchao Chen, Shilong Liu, Bochen Qian, Anjie Yang, Zhaoxuan Jin, Jianbo Deng, Philip Torr, Bernard Ghanem, and Guohao Li. CRAB: Cross-environment agent benchmark for multimodal language model agents. In *Findings of the Association for Computational Linguistics: ACL 2025*, 2025a. URL https://aclanthology.org/2025.findings-acl.1113/.

Yifan Xu, Xiao Liu, Xueqiao Sun, Siyi Cheng, Hao Yu, Hanyu Lai, Shudan Zhang, Dan Zhang, Jie Tang, and Yuxiao Dong. AndroidLab: Training and systematic benchmarking of android autonomous agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2025b. URL https://aclanthology.org/2025.acl-long.107/.

Shu Yao, Yuhua Luo, Qian Long, Jingru Fan, Zhuoyuan Yu, Yuheng Wang, Lin Wu, Yufan Dang, Huatao Li, and Chen Qian. Beyond global replanning: Hierarchical recovery for cross-device agent systems, 2026. URL https://arxiv.org/abs/2606.20487.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. In *Advances in Neural Information Processing Systems*, 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/82ad13ec01f9fe44c01cb91814fd7b8c-Paper-Conference.pdf.

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

Chaoyun Zhang, Liqun Li, Shilin He, Xu Zhang, Bo Qiao, Si Qin, Minghua Ma, Yu Kang, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Qi Zhang. UFO: A UI-focused agent for windows OS interaction. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, 2025a. URL https://aclanthology.org/2025.naacl-long.26/.

Chaoyun Zhang, Liqun Li, He Huang, Chiming Ni, Bo Qiao, Si Qin, Yu Kang, Minghua Ma, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. Ufo3: Weaving the digital agent galaxy, 2026. URL https://arxiv.org/abs/2511.11332.

Chi Zhang, Zhao Yang, Jiaxuan Liu, Yanda Li, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. Appagent: Multimodal agents as smartphone users. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, 2025b. URL https://doi.org/10.1145/3706598.3713600.

Boyuan Zheng, Boyu Gou, Jihyung Kil, Huan Sun, and Yu Su. GPT-4v(ision) is a generalist web agent, if grounded. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=piecKJ2DlB.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/4410c0711e9154a7a2d26f9b3816d1ef-Paper-Conference.pdf.

A INTERACTIVE ENVIRONMENTS

A.1 UNIFIED INTERACTION PROTOCOL

DevicesWorld integrates Linux desktops, Android phones, and SmartHome through a unified interaction protocol. Each runtime instance has a unique logical identifier, such as `linux_0`, `android_0`, or `home_0`. Separate Linux or Android instances do not share file systems, clipboards, or application states by default. Cross-device information transfer is performed by the agent reading information in one environment and acting in another.

Interaction formulation. We describe cross-device task execution as sequential interaction with heterogeneous device environments. The set of supported environment types is

$$\mathcal{R} = \{\text{Android}, \text{Linux}, \text{SmartHome}\}. \tag{1}$$

For a task $\mathcal{T}$ with a natural-language instruction g, let the participating device instances be

$$\mathcal{D} = \{d_1, \ldots, d_n\}, \qquad \tau(d_i) \in \mathcal{R}, \tag{2}$$

where $\tau(d_i)$ denotes the environment type of device d_i. At environment-interaction index t, device d_i has an internal state s_i^t, and the joint state is

$$s^t = (s_1^t, \ldots, s_n^t). \tag{3}$$

The task-specific initialization procedure configures the initial device states:

$$s^0 = \text{Init}_{\mathcal{T}}(\mathcal{D}). \tag{4}$$

The cross-device observation available at interaction t is

$$O^t = \left\{ \left(d_i, \tau(d_i), \mathcal{O}_{\tau(d_i)}(s_i^t)\right) \right\}_{i=1}^{n}, \tag{5}$$

where $\mathcal{O}_{\tau(d_i)}$ maps a device state to the observation exposed by its environment. Observation formats differ across environments and need not reveal the complete internal state.

Using the instruction, current observations, and the supplied history representation $\widetilde{H}^t$, the agent selects a target device and an action:

$$(d_t, a_t) \sim \pi\Big(\cdot \mid g, O^t, \widetilde{H}^t\Big), \qquad d_t \in \mathcal{D}, \quad a_t \in \mathcal{A}_{\tau(d_t)}(d_t), \tag{6}$$

where $\mathcal{A}_{\tau(d_t)}(d_t)$ is the action space supported by the target device. Executing the action updates the joint state through the corresponding transition function,

$$s^{t+1} = \mathsf{T}_{\tau(d_t)}(s^t, d_t, a_t), \tag{7}$$

and returns an action result or error signal r_t. The full interaction history after T environment interactions is

$$H^T = \left((O^t, d_t, a_t, r_t)\right)_{t=0}^{T-1}. \tag{8}$$

Here, $\widetilde{H}^t$ denotes the history actually supplied to the policy rather than necessarily the full trajectory. The policy may perform internal planning before producing a device action; the environment-interaction index t therefore need not equal the number of model calls.

After execution ends, the task verifier checks the final device states and task-relevant results:

$$(y, \rho) = \mathcal{V}_{\mathcal{T}}(s^T, Z^T), \qquad y \in \{0, 1\}, \quad \rho \in [0, 1], \tag{9}$$

where Z^T comprises results generated or modified during execution, y indicates task success, and ρ is the partial score. The verifier checks required local outcomes and their cross-device correspondence. Task success requires all mandatory checks to pass; the partial score uses only scoring checks, as defined in Section 4.1.

Action routing. Each environment interaction submits one top-level action object:

```
{
  "device_id": "<runtime instance>",
  "type": "<action interface>",
  "<payload field>": "<interface-specific payload>"
}
```

The runtime uses `device_id` to select the target instance and `type` to select an interface permitted by its action allowlist. One top-level action is not necessarily one low-level operation: a Linux GUI code snippet or a SmartHome workflow can contain multiple operations. The global action `{"type":"done"}` ends execution but does not determine task success. The global action `{"type":"wait"}` returns the available observations without a device operation or SmartHome clock advancement; it is distinct from environment-specific waiting actions. These global actions do not require a target device. Linux and Android retain visual interfaces, whereas SmartHome uses structured observations; its web console is a visualization tool rather than the default agent interface.

Table A.1 summarizes the observation formats, interfaces, and execution feedback supported by the environments.

Table A.1: Observation modalities, supported interfaces, and execution feedback.

Environment	Observations	Interfaces	Feedback
Linux	Desktop screenshots; accessibility-tree and terminal observations when enabled	`gui.computer_13`, `gui.pyautogui`, `cli.execute`	GUI execution results; CLI standard output, standard error, exit code, and execution status
Android	Screen images and structured UI-element information when available	`android.json_action`	Updated interface observations and action execution results
SmartHome	Structured time, room and appliance states, schedules, workflows, and reports	`smarthome.*`	Query results, state changes, and structured errors

Configuration used in the main experiments. The input includes screenshots from all participating Linux and Android devices, together with the Android UI-element tree. Linux accessibility-tree and terminal observations are disabled, although CLI actions still return standard output, standard error, exit codes, and execution status. SmartHome automatically returns structured information, including the current simulation time, room readings, appliance states, schedules, workflows and their progress, submitted reports, and the latest action result; these observations do not require an explicit query. Observation exposure is configurable, but the main experiments use this automatic SmartHome input.

All direct-interaction models, including GUI-optimized models, use the same prompt and action interfaces. Inputs contain the task instruction, the current cross-device observations, execution feedback, and the textual interaction history from the most recent 10 steps; historical screenshots are not included. The common prompt exposes `gui.pyautogui` and `cli.execute` for Linux, but not the runtime-supported `gui.computer_13` interface. Android actions and SmartHome APIs are listed in Tables A.3 and A.5, respectively. Direct `smarthome.write_attribute` and `smarthome.advance_time` calls are not included in the agent action allowlist; property writes remain available within workflow steps.

Each run is limited to 50 agent steps and 1,800 seconds, where one agent step corresponds to one model call. For multi-agent frameworks, the step budget is shared across the planner and all subagents, including planning and replanning calls.

A.2 Linux Observations and Action Space

The Linux environment uses an adapter based on OSWorld's `DesktopEnv` (Xie et al., 2024). It supports full-desktop screenshots and optional accessibility-tree and terminal observations. CLI actions return `stdout`, `stderr`, `exit_code`, and execution status as action feedback, independently of whether terminal observations are enabled.

A.2.1 Structured GUI Actions

The interface `gui.computer_13` accepts an action object with `action_type` and `parameters`. OSWorld translates these structured actions into mouse and keyboard operations without requiring the agent to generate Python code. Table A.2 lists the 13 GUI operations. Coordinates refer to the current screen, and a question mark denotes an optional parameter.

Table A.2: Structured Linux GUI operations.

Action	Parameters	Operation
`MOVE_TO`	`x`, `y`	Move the pointer to the target
`CLICK`	`x?`, `y?`, `button?`, `num_clicks?`	Click at the specified or current position
`MOUSE_DOWN`	`button?`	Hold a mouse button
`MOUSE_UP`	`button?`	Release a mouse button
`RIGHT_CLICK`	`x?`, `y?`	Right-click at the specified or current position
`DOUBLE_CLICK`	`x?`, `y?`	Double-click at the specified or current position
`DRAG_TO`	`x`, `y`	Drag to the target with the left button held
`SCROLL`	`dx`, `dy`	Scroll horizontally and vertically
`TYPING`	`text`	Enter text
`PRESS`	`key`	Press and release a key
`KEY_DOWN`	`key`	Hold a key
`KEY_UP`	`key`	Release a key
`HOTKEY`	`keys:string[]`	Send a key combination

A.2.2 GUI Code and CLI Interfaces

The `gui.pyautogui` interface accepts a `code` string containing PyAutoGUI operations, including pointer movement, clicking, dragging, scrolling, text entry, and key combinations. The `cli.execute` interface accepts a shell command through `command` and operates on files, directories, and data within the selected Linux instance. Cross-environment operations use the corresponding environment's interface; the Linux CLI is not an alternative control interface for Android or SmartHome.

A.3 Android Observations and Action Space

The Android environment is based on AndroidWorld (Rawles et al., 2025). It provides screen images and UI-element information from the interface hierarchy. Element records can include text, content descriptions, bounding boxes, application and resource identifiers, and properties such as whether an element is clickable, editable, scrollable, or selected. When an element list is unavailable, the agent can use screenshot-based coordinate actions. Element indices are valid only for the current interface of the corresponding device and must be reconsidered after navigation or scrolling.

Phone interactions use `android.json_action`, with the operation specified by the inner `action.action_type`. Table A.3 lists the 11 interaction actions. Element indices and coordinates are alternative targeting mechanisms and are not supplied together.

Table A.3: Android interaction actions.

Action	Main parameters	Operation
`click`	`index`, or `x`, `y`	Tap an element or screen coordinates
`double_tap`	`index`, or `x`, `y`	Double-tap the target
`long_press`	`index`, or `x`, `y`	Long-press the target
`input_text`	`text`; optional target and `clear_text`	Enter text, optionally clearing existing content; the underlying implementation then sends Enter
`keyboard_enter`	None	Send Enter
`navigate_home`	None	Return to the home screen
`navigate_back`	None	Return to the previous interface
`scroll`	`direction`; optional `index`	Scroll the screen or a target region
`swipe`	`direction`	Perform a directional swipe
`open_app`	`app_name`	Open an installed application
`wait`	None	Wait for the interface to update; the underlying implementation waits for one second

A.4 SmartHome States, Appliance Types, and APIs

A.4.1 State Representation and Device Identifiers

SmartHome maintains a structured home state containing rooms and appliance instances. The top-level `device_id`, such as `home_0`, selects the SmartHome runtime instance, while `parameters.device_id` selects an appliance within it, such as `guest_room_air_conditioner_1`.

The home state contains the current simulation time, room and appliance states, schedules, workflows and their progress, submitted answers and infeasibility reports, and the latest action result. Room states comprise temperature, relative humidity, illuminance, PM2.5, noise level, and occupancy. Appliance records identify their type, room, and state. The `get_home_state` query additionally provides the operation history. The current simulation time, `current_time`, is represented in ISO format and serves as the reference for scheduling.

A.4.2 Appliance Types and Control Capabilities

SmartHome supports 11 appliance types, each with type-specific state fields, commands, and parameter schemas (Table A.4). Tasks initialize the subset of appliances they require. The agent can query `get_device_capabilities` to inspect an appliance's commands, parameter types and ranges, current property values, and property writability.

Table A.4: SmartHome appliance types, key states and parameter ranges, and commands.

Appliance	Key states and ranges	Commands
Light	Power: `on`/`off`	`turn_on`, `turn_off`
Dimmable light	Power; brightness: 0–100%	`turn_on`, `turn_off`, `set_brightness`
Curtain	Opening: 0–100%; status: `closed`/`partial`/`open`	`open`, `close`, `set_open_pct`
Air conditioner	Power; mode: `auto`/`cool`/`dry`/`fan`/`heat`; target temperature: 16–30 °C	`turn_on`, `turn_off`, `set_mode`, `set_target_temperature`
Heater	Power; target temperature: 16–32 °C	`turn_on`, `turn_off`, `set_target_temperature`
Humidifier	Power; level: `low`/`medium`/`high`	`turn_on`, `turn_off`, `set_level`
Dehumidifier	Power; level: `low`/`medium`/`high`	`turn_on`, `turn_off`, `set_level`

Continued on next page

Table A.4 (continued)

Appliance	Key states and ranges	Commands
Air purifier	Power; level: low/medium/high	turn_on, turn_off, set_level
Washer	Power; cycle: quick/normal/heavy/delicate; remaining time; status	set_cycle, start_cycle, stop_cycle
Dryer	Power; cycle: quick/normal/heavy/delicate; remaining time; status	set_cycle, start_cycle, stop_cycle
Robot vacuum	Power; battery level; status: cleaning/paused/docked	start_cleaning, pause, return_to_dock

Ordinary lights support only on/off control; washers and dryers use cycle commands. Some appliances also affect room variables, such as illuminance, temperature, or PM2.5. Washer and dryer cycles progress when simulated time is advanced. SmartHome is a deterministic state simulator for task execution and verification, not a physically calibrated residential model.

A.4.3 Environment-Level Action Interfaces

Table A.5 presents the core SmartHome APIs. The shared prefix smarthome. is omitted; arguments are supplied in parameters, and a question mark indicates an optional argument.

Table A.5: Core SmartHome APIs and parameters.

API	Parameters	Operation
get_home_state	None	Query the complete home state
list_rooms	None	List rooms
list_devices	room_id?	List appliances, optionally by room
get_room_state	room_id	Query a room state
get_device_state	device_id	Query an appliance state
get_device_capabilities	device_id	Query commands, ranges, and writability
get_device_attribute	device_id, attribute	Read an appliance property
command	device_id, command, args	Execute an appliance command
schedule_command	schedule_id, run_at, device_id, command, args	Create or update a single-command schedule
list_schedules	None	Query schedules
cancel_schedule	schedule_id	Cancel a schedule
schedule_workflow	workflow_id, run_at, steps	Create or update a multi-step workflow
list_workflows	None	Query workflows
get_workflow_status	workflow_id	Query a workflow and its progress
cancel_workflow	workflow_id	Cancel a workflow
report_answer	answer:object	Submit a nonempty structured answer
report_infeasible	category, reason, target?, details?	Submit an infeasibility report

A scheduled run_at must be later than current_time. Workflow steps form a nonempty ordered list of appliance commands or writes to writable properties. They execute sequentially after the workflow is triggered, without a defined inter-step delay parameter. Reusing a schedule or workflow identifier updates the existing entry. Action feedback reports execution status and errors such as missing appliances, unsupported commands, invalid arguments, or invalid scheduling times.

Simulation time and scheduled execution. The SmartHome simulation clock remains fixed during agent execution, independent of model calls and wall-clock time; immediate control can still change appliance states. To verify future scheduled effects, the evaluator advances simulation time after the agent run, allowing due commands or workflows to execute, and checks the resulting appliance states against the task requirements. This advancement belongs to evaluation, not agent interaction. A schedule marked `active` has been created but has not yet executed; schedule-creation requirements can be checked against the configured time and intended effects. The evaluator snapshots the SmartHome state before advancing time and restores it after verification.

The examples below show immediate control and a multi-appliance workflow. They assume the listed appliances exist and `current_time` precedes the scheduled `run_at`.

```
{
  "device_id": "home_0",
  "type": "smarthome.command",
  "parameters": {
    "device_id": "guest_room_air_conditioner_1",
    "command": "set_target_temperature",
    "args": {"temperature_c": 24}
  }
}
```

```
{
  "device_id": "home_0",
  "type": "smarthome.schedule_workflow",
  "parameters": {
    "workflow_id": "guest_prep",
    "run_at": "2026-06-16T19:35:00",
    "steps": [
      {"device_id": "guest_room_air_conditioner_1",
       "command": "turn_on", "args": {}},
      {"device_id": "guest_room_air_conditioner_1",
       "command": "set_target_temperature",
       "args": {"temperature_c": 24}},
      {"device_id": "guest_room_dimmable_light_1",
       "command": "set_brightness",
       "args": {"brightness_pct": 35}}
    ]
  }
}
```

A.5 Task Initialization and Execution Lifecycle

Task configurations specify the instruction, device instances, input resources, initialization and cleanup steps, evaluation conditions, and execution budgets. Runtime configurations specify each instance's backend, observation options, and action allowlist. Before interaction, each environment prepares its task-specific state. Linux uses `DesktopEnv` to reset the virtual machine and prepare files, directories, and application interfaces. Android resets its runtime and prepares files, applications, and data such as messages, contacts, and calendar entries. SmartHome initializes simulated time, room readings, appliances, and existing schedules, and records a baseline state for subsequent change verification.

After initialization, the environments return initial observations and begin round-by-round interaction. Execution ends when the agent declares completion or reaches the model-call or time limit. The evaluator then checks outputs and environment states, followed by task-defined cleanup. Initialization and evaluation may use ADB, file access, or application-data queries. These are infrastructure mechanisms, not additional interfaces granted to the agent during task execution.

B Task Construction

B.1 Task Specification and Cross-Device Dependencies

Each DevicesWorld task combines a user instruction with a specification of the participating devices, initial resources, setup operations, completion checks, and cleanup operations. The instruction describes the user's goal and identifies the relevant applications or resources. The specification assigns source information and initial application states to concrete device instances and defines checks on the required files or states. In cross-device tasks, information or state obtained on one device determines an operation or output on another.

A calendar-reconciliation task from DevicesWorld-Lite asks the agent to reconcile a Linux meeting schedule with Android Simple Calendar Pro and record the changes. The original instruction is reproduced below.

Example task: original user instruction

This week's meeting schedule has conflicts: Android Simple Calendar Pro and Linux `/tmp/schedule/week.csv` disagree on details for meetings with the same names. Follow `/tmp/schedule/rule.txt`: synchronize each matching meeting's date, start, end, and location from Android, leave non-matching meetings unchanged, then write `/tmp/schedule/log.json` with every field that changed and its old and new values.

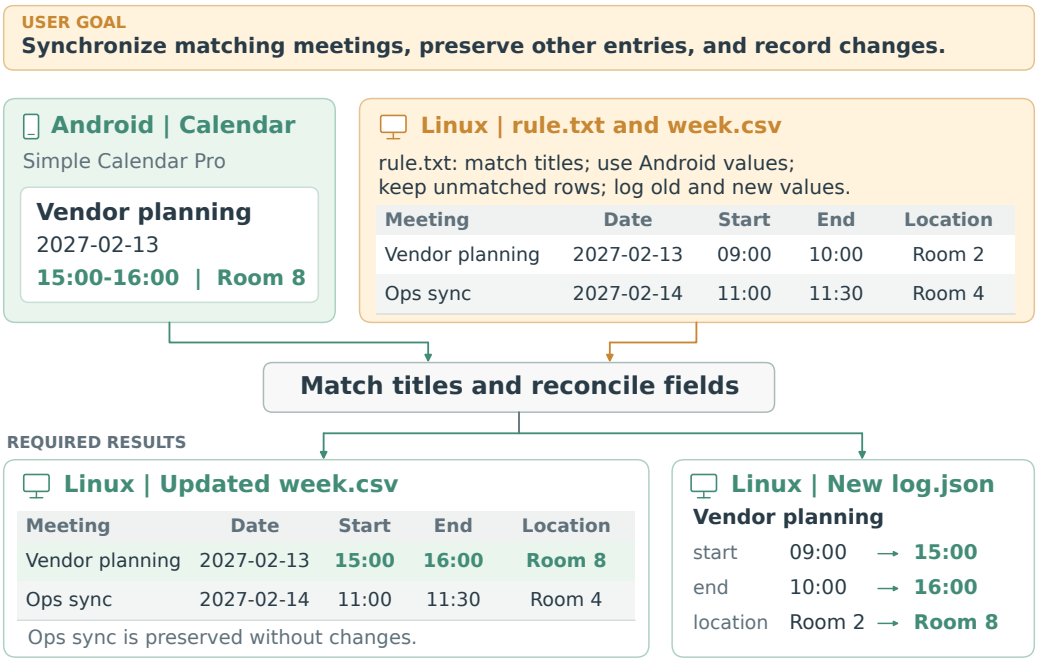


Figure B.1: Task specification and cross-device dependency. The Android calendar and Linux resources jointly determine the corrected schedule and change log. The matching meeting is updated, while "Ops sync" remains unchanged. The lower panels show the required results.

Android holds the authoritative calendar event, while Linux holds the conflicting schedule and the reconciliation rule. Both devices contain a meeting named "Vendor planning." The Linux entry lists 09:00–10:00 in Room 2; the Android event specifies 15:00–16:00 in Room 8 on the same date. Another Linux entry, "Ops sync," has no matching event and must retain its original values. The Android event supplies the new values, while the Linux resources supply the matching rule, previous values, and records to preserve. Figure B.1 illustrates how these inputs determine the two required outputs.

The task has two scored completion checks. The CSV check requires the two expected logical records, including the unchanged meeting, while allowing either row order. The log check verifies the old and new start time, end time, and location for "Vendor planning." Its date is unchanged, so no date change needs to be recorded. The excerpt below shows the device identifiers, resource variables, and evaluation targets used in the configuration.

Example task: selected configuration fields

```
{
  "id": "al_calendar_schedule_conflict",
  "devices": [
    {"id": "android_0", "type": "android"},
    {"id": "linux_0", "type": "linux"}
  ],
  "variables": {
    "week_csv": "/tmp/schedule/week.csv",
    "log_json": "/tmp/schedule/log.json"
  },
  "evaluation": [
    {"device_id": "linux_0", "func": "check_semantic_change",
     "result": {"type": "vm_file", "path": "${log_json}"}},
    {"device_id": "linux_0", "func": "check_csv",
     "result": {"type": "vm_file", "path": "${week_csv}"}}
  ]
}
```

Setup, cleanup, and expected-value rules are omitted from this excerpt.

Setup installs the initial calendar event and Linux source files and checks that the inputs are readable, the schedule and its directory are writable, and the new log does not yet exist. The schedule therefore begins with the conflicting values rather than the corrected result. Cleanup removes the task's calendar entries and working directory.

B.2 Task-Construction Pipeline

We organize task construction into four stages: configuration sampling, task design, executable instantiation, and validation and repair. Figure B.2 illustrates the overall workflow across mobile, desktop, and SmartHome environments. The implementation uses a staged Android/Linux engine and task-family-specific builders for SmartHome and mixed tasks. The following description details the Android/Linux engine and the prompts used within each stage.

Table B.1 groups the prompt templates by their role within the four stages. The boxes present abridged English translations of selected passages from the original Chinese templates, retaining their variable and field names. Configuration sampling is implemented in code; the other stages combine prompt calls with programmatic operations.

Construction stage	Main work	Prompt templates
1. Configuration sampling	Select devices, applications, and the initial-state setting	Programmatic sampling
2. Task design	Define and review the task; repair its instruction or design when needed	Task design; design review; design repair
3. Executable instantiation	Construct resources, setup, evaluation, and cleanup	Resource planning; environment setup; evaluation and cleanup
4. Validation and repair	Check resources, initialization, and evaluation; prepare evaluator tests and repair detected issues	Resource repair; evaluator test planning

Table B.1: Prompt templates within the four-stage construction pipeline. Review and repair also operate within the preceding stages.

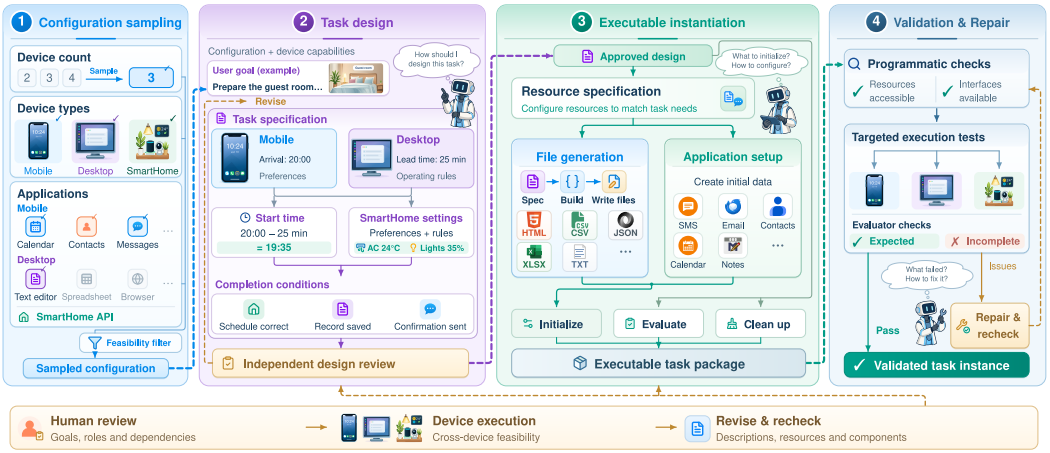


Figure B.2: Four-stage task-construction workflow. Configuration sampling defines the devices and applications; task design specifies the user goal, cross-device dependencies, and completion conditions; executable instantiation creates resources and environment components; validation checks the resulting task and returns issues for repair and rechecking. The lower band shows human review and device execution across the workflow.

Stage 1: Configuration sampling

Configuration sampling determines the devices and applications available for task design. The sampler cycles through six combinations of Android phones (M) and Linux desktops (D): 2M, 2D, M+D, M+2D, 2M+D, and 2M+2D. Within each combination, a seeded generator selects one supported application or file type for each device. The sampling pools contain 13 Android and 14 Linux options, including calendar events, contacts, notes, and recipes on Android, and structured files, documents, browser pages, and code projects on Linux. With probability 0.25, the configuration requests plausible non-target records or files in the initial state. These distractors must be distinguishable using visible selection criteria. The resulting device, application, and initial-state configuration is passed to task design, where the task goal and information flow are determined.

Stage 2: Task design

Task design expands the sampled configuration into a specification of what each device contributes and how the required results are derived. The prompt asks the model to define the visible sources, cross-device dependencies, outputs, and expected results before writing the user instruction. A visible source records its device, application or file type, location, and initial content or content requirements. A required output records its destination and completion condition. Sources and outputs have distinct identifiers even when a task updates the same object in place, so the initial state remains separate from the required result.

The `information_flow` field gives a textual summary of the dependencies from sources to outputs, while `why_cross_device` explains why the participating devices are needed. Dependencies can involve transferring information, combining facts and rules from different devices, resolving conflicting records, or using a result to determine a later operation. At the level of individual expected results, `value_trace` records the source evidence and derivation rules used to specify the required values. These provenance records are created during task construction.

Expected results are organized in `expected_data` under the corresponding `output_id`. Each trace has three fields: `expected_path` points to an entry in that structure; `expected_value` gives its exact value or a concise group/rule summary; `visible_source` identifies the evidence and derivation rule. A source-based trace begins with a real `source_id`; a value or format explicitly prescribed by the user instruction can instead cite `instruction:`. When a result depends on several sources, the trace cites all of them and states the relevant selection key, matching relation, conflict-resolution rule, or calculation. A group or rule trace can cover repeated rows or fields. Each required output must have at least one trace covering the decisions relevant to evaluation.

The task-design template includes a three-device example. An SMS on `android_1`, identified as `proposal_request_sms`, names a proposal PDF, specifies five fields to extract, and gives the requested note title. The PDF on `linux_0`, identified as `proposal_pdf`, supplies the proposal ID, project title, budget, approval status, and reviewer. The required output is a Markor note on `android_0`. The following fields are reproduced from that template example.

Stage 2: Information flow and value trace (example excerpt)


```
{
  "information_flow":
    "Android SMS request + Linux PDF -> Android Markor note",
  "expected_data": {
    "proposal_summary_note": {
      "title": "Proposal PROP-2026-77 Summary",
      "content_contains": [
        "Proposal ID: PROP-2026-77",
        "Project Title: Harbor Sensor Refresh",
        "Budget: 48,750 USD",
        "Approval status: approved",
        "Reviewer: Mina Hart"
      ]
    }
  },
  "value_trace": [
    {
      "expected_path": "proposal_summary_note.title",
      "expected_value": "Proposal PROP-2026-77 Summary",
      "visible_source":
        "proposal_request_sms requested note title"
    },
    {
      "expected_path": "proposal_summary_note.content_contains",
      "expected_value": "the complete five-item content_contains list",
      "visible_source":
        "proposal_request_sms field list + proposal_pdf text anchors"
    }
  ]
}
```


The two traces distinguish the sources' contributions. The SMS determines the title and which fields to extract; the PDF supplies values such as the budget of 48,750 USD. The PDF alone does not specify the requested field list or note title, while the SMS alone does not provide the proposal facts. The second trace is a group record covering all five entries in `content_contains`, so it cites both sources. The third device is the required destination for the resulting note.

Each trace must lead back to an initial visible source or an explicit instruction rule. For a result produced by a later action, the trace identifies the input and rule that determine it; the new output cannot serve as its own evidence. Review returns hidden requirements for revision. Resource construction must preserve the cited facts and rules, while evaluator construction uses the corresponding expected data and traces. Changes to a source, output, or rule require coordinated updates to the affected design fields before instantiation.

The user instruction is written after these relationships are established. It identifies the source locations, overall goal, and output destination. Detailed facts and rules already present in the sources remain there for the agent to obtain during execution.

Stage 2 | Task design prompt (excerpt)

System

Design a multi-device task from the sample, supported capabilities, and pattern catalog. Define the user instruction, visible sources, required outputs, information flow, deterministic expected data, and their value traces. Do not generate runnable task configuration or implementation calls. Return only JSON.

User prompt: inputs

`{{ sample_json }}`, `{{ surface_capability_context }}`, `{{ pattern_catalog }}`, `{{ extra_constraints }}`

User prompt: requirements

1. Construct the sources, dependencies, outputs, expected data, and value traces before writing the instruction.
2. Use only the devices and surfaces allowed by the sample. Every sampled device must participate as a visible source or a required output. The task must contain a necessary cross-device dependency; unrelated actions on separate devices do not qualify.
3. Visible sources describe inputs or initial states available at the start. Do not prefill newly requested outputs. For an update, distinguish the initial source from the updated output.
4. Summarize the source-to-target dependency in `information_flow` and explain the necessary cross-device relationship in `why_cross_device`. These descriptions must agree with the actual sources and outputs.
5. Organize `expected_data` by the exact `output_id` values. Each trace includes `expected_path`, `expected_value` (exact value or group/rule summary), and `visible_source`. The path must exist in the expected data, and every required output must have at least one trace.
6. Start `visible_source` with a real `source_id`, followed by the field, row, rule, or calculation supporting the expected result. Cite all sources used jointly and retain the selection key, matching relation, source-of-truth rule, or formula. Repeated values may share a group or rule trace.
7. Trace expected results to initial visible sources or explicitly stated `instruction:` rules. Do not cite an output created during execution as its own evidence, or use a path or "computed from source" in place of a source ID.
8. Write the task title and instruction in English. Identify the source and target devices, applications, objects, or paths. Keep facts and rules already supplied by the sources in those sources; do not copy the answers into the instruction.
9. For `setup=distractor`, include a plausible non-target candidate in the visible initial state and provide a visible selection key, matching key, or source-of-truth rule.
10. Return `status=unsupported` when the sampled capabilities cannot support a task that can be completed, observed, and deterministically checked. Do not produce setup actions, evaluators, cleanup actions, or helper/API calls at this stage.

Output contract (key fields)

`status` (`accepted` or `unsupported`), `instruction`, `detailed_description`, `information_flow`, `why_cross_device`, `visible_sources`, `required_outputs`, `expected_data`, and `value_trace`.

A separate review pass checks whether the visible information supports the expected results and whether the required operations and outputs are supported by the sampled capabilities. It examines each device's source or output role and asks whether a source claimed to be necessary could be omitted while still completing the main goal. The review returns `accepted`, `revise`, or `rejected`, with blocking issues and required fixes when the design is not accepted.

Stage 2 | Design review prompt (excerpt)

System

Review whether the task design is supported by visible information and current capabilities and has stable completion evidence. You may optimize the user instruction without changing the structured design. Require revision or rejection for substantive design problems. Return only JSON.

User prompt: inputs

`{{ sample_json }}`, `{{ surface_capability_context }}`, `{{ task_design_json }}`

User prompt: requirements

1. Judge whether the task can be completed using the instruction and sources visible on the devices. The internal `detailed_description` cannot substitute for information visible to the agent.
2. Check that each sampled device has a necessary source or output role. Verify that `information_flow` and `why_cross_device` agree with the actual dependencies; ask whether a source claimed to be necessary could be omitted while still completing the main goal. Distinguish initial states from completed outputs.
3. Check that each required output has expected data and a value trace grounded in visible information. If a required field or rule appears only in internal descriptions or expected data, return `revise`; do not hide the missing requirement inside an optimized instruction.
4. Check the sampled capabilities for supported operations and stable evidence of completion. Complexity or a large number of steps is not itself a defect.
5. Optimize the instruction only when the task has no blocking structural problem. Preserve the source and target locations and the user's goal. Do not change the sources, outputs, expected data, or value traces in this review.
6. Return `accepted` when there is no blocking issue, `revise` when the design can be repaired within the sample, and `rejected` when its core intent cannot be retained. For each blocking issue, specify its location, the problem, and the smallest required repair.

Output contract (key fields)

`decision`, `optimized_instruction`, `instruction_revision_notes`, `blocking_issues` (each with `area`, `issue`, and `required_fix`), and `optional_improvements`.

The review may refine the instruction while preserving the structured design. Structural problems are passed to the task-design repair prompt, which returns a complete revised specification with the affected expected data, value traces, and information-flow description updated together. The accepted specification is then passed to resource planning and construction of the execution components.

Stage 2 | Task-design repair prompt (excerpt)

System

Repair an existing task design by making the smallest coordinated changes needed to address the review's blocking issues. Stay within the sampled devices and capabilities and preserve the core task intent. Return the complete task-design JSON, not a patch, diff, or repair proposal. Do not generate execution components.

User prompt: inputs

`{{ sample_json }}`, `{{ surface_capability_context }}`, `{{ task_design_json }}`, `{{ review_json }}`

User prompt: requirements

Stage 2 | Task-design repair prompt (excerpt) – continued

1. Address every item in `review.blocking_issues` and document the actual change in `repair_notes`. Preserve unaffected fields, IDs, object names, and paths. Apply optional improvements only when needed to keep the repair consistent.
2. Keep the device topology, allowed surfaces, and feasibility type unchanged. Preserve the necessary cross-device dependency and any required distractor with its visible selection rule.
3. When a source, output, or rule changes, update the instruction, detailed description, completion conditions, expected data, value traces, `information_flow`, and `why_cross_device` wherever affected. Remove stale references.
4. For a hidden requirement, place the missing user requirement or rule in an appropriate visible source. If the expected value must be derived, provide the inputs and derivation rule rather than the final answer. Remove an unsupported extra requirement that is unrelated to the task.
5. Every required output must have expected data and at least one trace to an initial visible source or explicit instruction rule. Keep initial states separate from completed outputs.
6. Return `unsupported` if a core output cannot be repaired within the sample while preserving the task intent. Do not replace it with an unrelated report or weaken a content requirement to file existence.

Output contract (key fields)

Return the complete task-design fields, together with `repair_notes`. For an accepted repair, `unsupported_reason` is null and `repair_notes` is nonempty; for an unsupported repair, give a specific reason.

Stage 3: Executable instantiation

Executable instantiation converts the reviewed design into resources and device state. Short text and compact application records can be specified directly, while larger tables, documents, archives, media, and code projects use structured resource requirements. These requirements describe the content, organization, and key values needed by the task, preserving the visible facts and rules identified in the reviewed design and its value traces. File builders then write the resources, and format-specific checks inspect properties such as table headers and values, document text, archive members, or media validity.

Device-specific setup operations place these resources in the environment and initialize the application records. Setup also removes old target outputs that could interfere with evaluation and checks whether the intended sources are visible in the relevant applications. An object to be updated is installed in its initial state; a newly requested output is absent. The resource-planning and setup prompts below specify the requirements for these components. Resource checks and their repair prompt are described under validation and repair.

Stage 3 | Resource planning prompt (excerpt)

System

Turn visible-source specifications into concrete asset plans. A code materializer writes the actual files. Do not generate setup, evaluation, cleanup, oracle steps, or the final task configuration.

User prompt: inputs

```
{{ task_design_json }}
```

User prompt: requirements

1. Plan assets for visible sources that need file generation. Required outputs must not be materialized as initial assets. For an object to be updated in place, represent only its initial source state.

Stage 3 | Resource planning prompt (excerpt) – continued

2. Describe source content and structure in an asset plan; the code materializer writes the actual files. Include `asset_kind`, `materializer`, `quality_checks`, and `expected_anchor_values` for each asset.
3. Cover every source with `content_mode=requirements` in the asset manifest. Inline application records may be initialized directly during setup. Do not generate setup, evaluation, or cleanup actions here.

Output contract (key fields)

`status`, `asset_manifest`, and `external_files`.

Stage 3 | Environment setup prompt (excerpt)

System

Write only setup actions and setup visibility checks. Materialize the reviewed design's visible sources and the generated assets faithfully. Do not write evaluation, cleanup, metadata, oracle steps, or the final task configuration.

User prompt: inputs

`{{ sample_json }}`, `{{ task_design_json }}`, `{{ asset_plan_json }}`, `{{ surface_capability_context }}`

User prompt: requirements

1. Use only the sampled devices. Place every visible source where the agent can access it, and remove old target states that could affect evaluation. Do not introduce the expected answer as an additional source.
2. Reference only files supplied by the asset plan or its materializer. For application records, establish the actual application state and its visibility; uploading a file alone may not make that state visible.
3. Do not initialize completed outputs. When the task updates an existing object, keep it in the initial state so that completion checks can verify the requested change.

Output contract (key fields)

`status`, `setup`, and `setup_visibility_checks`.

Completion checks and cleanup are constructed from the same design. The evaluation prompt requires a scored check on the resulting content or state of each required output. Separate checks that protect an input or verify an unchanged source can be required without contributing to the partial score. Expected values are taken from the reviewed design's `expected_data`, with `value_trace` identifying their source evidence and derivation rules. The evaluation prompt also specifies how to handle selection boundaries and text: it asks for coverage of required and excluded records, and reserves exact matching of reply text for cases with an explicit template. Cleanup covers both initialized resources and the files or application records that execution may create.

Stage 3 | Evaluation and cleanup prompt (excerpt)

System

Construct evaluation, cleanup, metadata, and limits. Expected values must come from the design's expected data and value traces. Do not change the instruction or sources, or generate setup or oracle steps.

User prompt: inputs

`{{ sample_json }}`, `{{ task_design_json }}`, `{{ setup_plan_json }}`, `{{ surface_capability_context }}`

Stage 3 | Evaluation and cleanup prompt (excerpt) – continued

User prompt: requirements

1. Derive expected values from the task design's `expected_data` and `value_trace`. Do not change the instruction, sources, or setup.
2. Give every required output at least one score-enabled goal evaluator. Source sanity checks and guards may be included with `enable_score_calc=false`. Do not score a source condition that already holds after setup as task progress.
3. For structured outputs, check the required fields or content rather than only a completion flag or a nonempty file. For selection tasks, include the correct records and exclude non-target records. Requirements such as "all" or "for each" must be checked across the requested set.
4. Every checked date, field, filename, or formatting rule must follow from the instruction or a visible source. For messages and other natural-language replies, use exact body matching only when a source explicitly provides a verbatim template; otherwise check the required information or content anchors.
5. Check that placeholders in a template have been replaced when required by the task. A source guard must not require an old state that conflicts with the requested update, deletion, or move.
6. Compare cleanup against every visible source and required output. Remove the initialized resources and possible outputs, including relevant application records and distractors.

Output contract (key fields)

`status`, `evaluation`, `cleanup`, `metadata`, and `limits`.

Together, these resources and execution components form an executable candidate task. The assembler combines the instruction, sampled devices, resource references, setup, evaluation, cleanup, and limits into its task configuration.

Stage 4: Validation and repair

Validation checks intermediate resources and the assembled task, returning failures to the affected component for repair and rechecking. Resource checks run during instantiation: the materializer writes the planned files and returns a quality report. Schema errors or a failed quality report trigger the resource-repair prompt, which receives the invalid plan and diagnostics. The repaired plan is materialized and checked again before setup construction continues.

Stage 4 | Resource repair prompt (excerpt)

System

Repair the schema, external files, and quality-check problems in an existing asset plan. Preserve task semantics and the instruction. Do not generate setup, evaluation, cleanup, oracle steps, or the final task configuration. Return one complete repaired asset-plan JSON object.

User prompt: inputs

`{{ task_design_json }}`, `{{ asset_plan_json }}`, `{{ asset_errors_json }}`, `{{ asset_quality_report_json }}`

User prompt: requirements

1. Cover each source with `content_mode=requirements` in the asset manifest and external files. Keep inline sources inline. Remove any asset that creates a required output before execution.
2. Use supported materializer and quality-check names. External assets must have matching file entries and real content or a valid generation specification, rather than a summary or placeholder.
3. Use the actual target extensions for documents, archives, images, and media. Represent multiple image or media files as separate assets where required; do not replace them with a JSON listing.

Stage 4 | Resource repair prompt (excerpt) – continued

4. When a document anchor is missing, put the value into text, table rows, or worksheet cells that the agent can read. Adding the value only to an internal anchor list does not repair the source.
5. For an in-place update, keep the resource in its initial state. Do not expose generator manifests, intermediate specifications, or benchmark identifiers as task resources. Return `unsupported` if repair requires changing task semantics.

Output contract (key fields)

One complete asset-plan object with `status`, `unsupported_reason`, `asset_manifest`, and `external_files`. Return `unsupported` with a reason if the plan cannot be repaired without changing task semantics.

For the assembled task, static checks verify that the configuration can be loaded, its referenced resources exist, and at least one scored completion check is present. Evaluator tests examine whether completion checks distinguish correct results from incomplete or incorrect ones. The test-planning prompt specifies positive states that satisfy the goal and negative states that leave it unfinished or introduce a specific error, such as an incorrect table entry or a missing field. Separate test scripts construct these states and inspect the evaluator response.

Stage 4 | Evaluator test planning prompt (excerpt)

System

Plan positive and negative oracle steps for testing. Do not change the task design or evaluator, and do not generate the final task configuration.

User prompt: inputs

`{{ task_design_json }}`, `{{ eval_cleanup_metadata_plan_json }}`

User prompt: requirements

1. Plan positive and negative test states for score-enabled goal checks. The positive state should satisfy evaluation; a negative or unchanged state should fail. Reuse the device IDs already present in the task.
2. Prefer changing one important goal condition in the negative case, such as a key value or required field. Leave setup sources in place and create or modify the required outputs directly. These are evaluator test fixtures, not instructions or answers supplied to the agent.

Output contract (key fields)

`positive_oracle_steps`, `negative_oracle_steps`, and `smoke_expectations`.

Targeted device tests exercise setup, evaluation, and cleanup. One setup test found that adding a Retro Music playlist erased earlier playlists. Changing the helper to add or update only the named playlist resolved the failure. Such tests check the combined effect of initialization operations, including whether later operations overwrite source records.

Manual execution testing. We conducted two rounds of manual execution testing on DevicesWorld-Lite. These tests complemented the automated checks through hands-on task execution, with identified issues returned to the relevant task components for revision and rechecking.

B.3 DevicesWorld-Lite Selection and Composition

DevicesWorld-Lite is a fixed subset of 200 tasks selected from DevicesWorld for the main experiments. Task selection considered device combinations, application scenarios, task requirements, and application and capability coverage, together with instruction quality, initial resources, and completion checks. All baselines use the same task set, initialization configurations, and outcome verification mechanism. Table B.2 and Figure B.3 compare the device composition and application coverage of DevicesWorld-Lite with the full benchmark.

Table B.2: Device composition of DevicesWorld and DevicesWorld-Lite. Each entry gives a task count and its percentage within the corresponding set.

Category	DevicesWorld $N = 5{,}894$	DevicesWorld-Lite $N = 200$
Task scope		
Single-device	280 (4.75%)	10 (5.00%)
Cross-device	5,614 (95.25%)	190 (95.00%)
Environment-type combination		
Android only	372 (6.31%)	20 (10.00%)
Linux only	371 (6.29%)	18 (9.00%)
SmartHome only	280 (4.75%)	10 (5.00%)
Android + Linux	1,876 (31.83%)	67 (33.50%)
Android + SmartHome	1,015 (17.22%)	25 (12.50%)
Linux + SmartHome	1,010 (17.14%)	30 (15.00%)
Android + Linux + SmartHome	970 (16.46%)	30 (15.00%)
Device instances per task		
1	280 (4.75%)	10 (5.00%)
2	3,103 (52.65%)	95 (47.50%)
3	1,853 (31.44%)	66 (33.00%)
4	658 (11.16%)	29 (14.50%)

Each block is a separate distribution over the same tasks; rows across blocks should not be added together. "Android only" and "Linux only" specify an environment type, not a single device: these tasks can involve multiple independent instances. One SmartHome environment counts as one instance, regardless of the number of appliances it contains. Percentages may not sum to exactly 100% because of rounding.

As shown in Table B.2, DevicesWorld-Lite covers all seven environment-type combinations, with cross-device tasks accounting for 95.0% of the subset, compared with 95.25% of the full benchmark. Tasks involving three or four device instances account for 47.5% of Lite, compared with 42.6% of the full benchmark.

Figure B.3 reports task-level application coverage. Each task is counted at most once for each application, even when the application appears on multiple devices. A task may involve multiple applications, so coverage percentages can sum to more than 100%. HTML resources refer to pages used as task sources or interactive interfaces and are counted separately from applications.

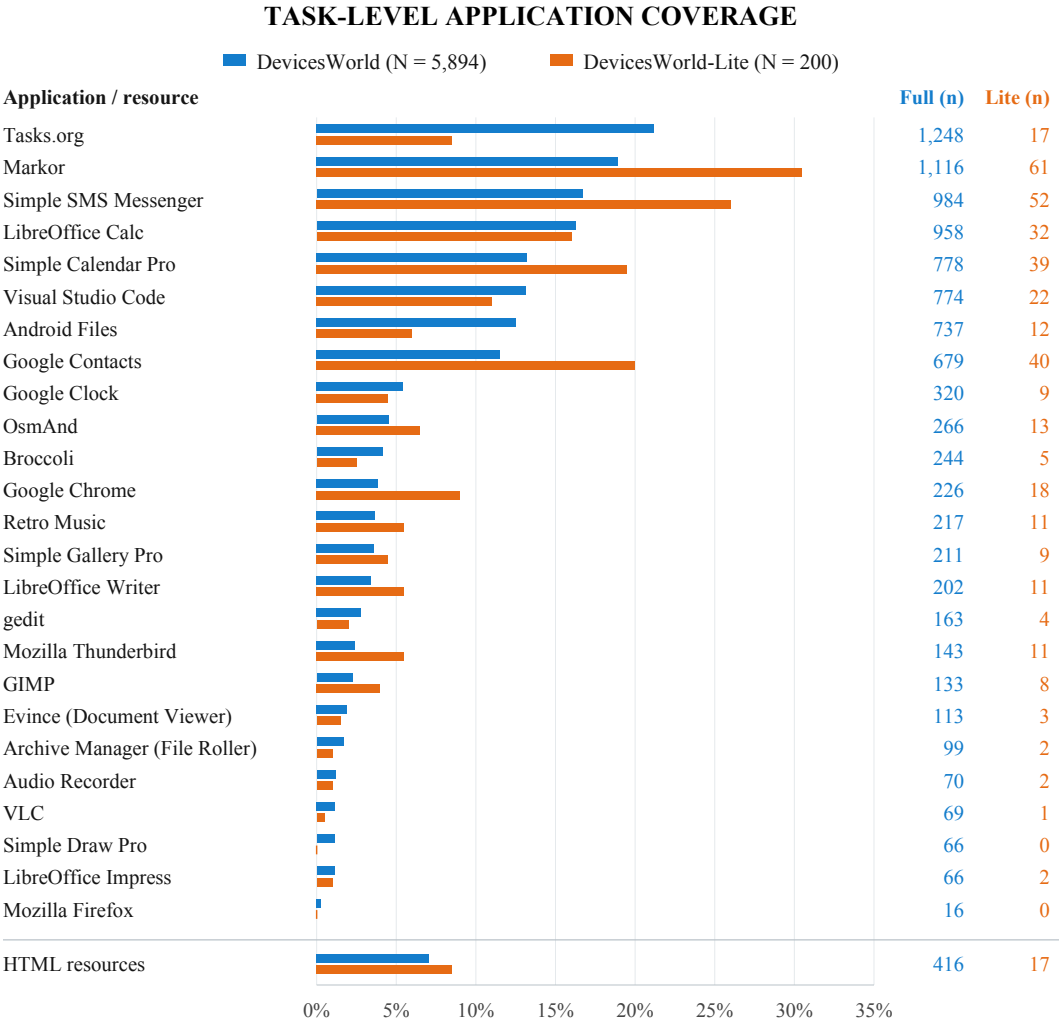


Figure B.3: Application coverage in DevicesWorld and DevicesWorld-Lite, using the same 25 application categories as main-text Figure 2. Bar lengths show the percentage of tasks involving each application, and the right-hand columns give task counts. Each task is counted at most once per application. HTML resources are reported separately and are not counted as an application.

DevicesWorld-Lite covers 23 of the 25 application categories shown in Figure B.3, spanning mobile note-taking, messaging, calendar and contact management, as well as desktop document, spreadsheet, code, and media tasks. HTML resources occur in 8.5% of Lite tasks, compared with 7.1% of the full benchmark.

C Representative Failure Cases

This appendix presents four failed runs from the main experiments, illustrating incorrect information matching, searches on the wrong device, omitted control actions, and stalled calendar navigation. Screenshots show selected steps, numbered from 0 (Step 0); rule, workflow, and device-state panels reproduce the execution records.

C.1 Information Association and Target-Device Localization

C.1.1 Confusing an Ingredient with a Recipe Name

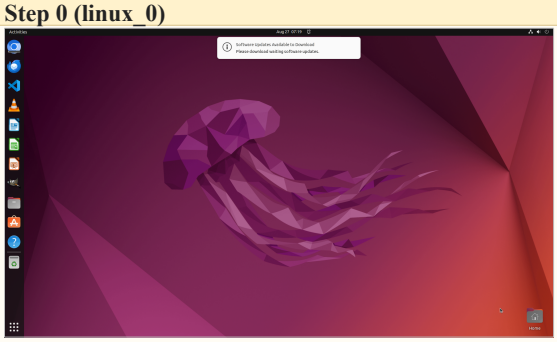
GPT-5.5 executes `linux_android_smarthome_696`, which requires combining Linux rules with an Android recipe, scheduling a SmartHome scene, and updating an existing task on the phone.

Task instruction

Match the recipe code in the existing `Dinner scene request` Task with the Broccoli recipe and `/tmp/home_ops/service-notes/source/index.csv`. Schedule the mapped SmartHome cooking scene, then complete that original Task—do not create a duplicate—and update its note with the matched recipe, the scheduled scene time, and that the scene has been scheduled.

The existing `Dinner scene request` to-do item contains the note `Recipe code: R-5`. On Linux, `index.csv` maps this code to the ingredient condition `required_ingredient=tofu` and time 18:05. In Broccoli, `Stir Fry` lists `tofu, vegetables, oil`; both its title and ingredients are visible in the initial screen. The matching recipe name is therefore Stir Fry, not tofu.

Step 0 (linux_0)



Action:

`cli.execute`

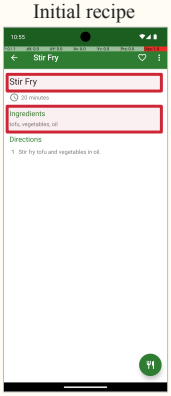
Python script reads this CSV:

`/tmp/home_ops/service-notes/source/index.csv`

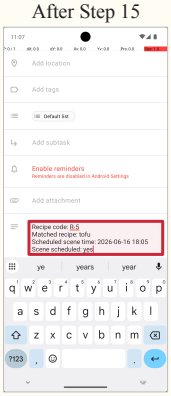
Explanation: The CSV returns R-5, tofu and 18:05. The ingredient must be matched to a phone recipe name.

Initial recipe and Step 15 (android_0)

Initial recipe



After Step 15



Action:

```
android.json_action
input_text(clear_text=true)

text:
Recipe code: R-5
Matched recipe: tofu
Scheduled scene time: 2026-06-16 18:05
Scene scheduled: yes
```

Explanation: Broccoli shows Stir Fry with tofu as an ingredient. The input uses the ingredient as the matched recipe name.

Figure C.1: (a) Dinner preparation. The Linux ingredient condition and the phone recipe are shown alongside the note entered at Step 15.

The agent reads the CSV at Step 0 and the R-5 row again at Step 12. At Step 15, however, it enters `Matched recipe: tofu` in the note, using the ingredient condition as the recipe name instead of matching it to `Stir Fry` in Broccoli. It marks the original to-do item as completed at Step 30, leaving the incorrect recipe name in the saved record (Figure C.1(a)).

Step 30 (android_0)



Action:

`android.json_action`
`click(x=74, y=348)`

Explanation: Clicks the completion checkbox. The call returns ok=true, but the completed task retains "Matched recipe: tofu".

Figure C.1: (a), continued. The original to-do item after Step 30, still containing the incorrect recipe name.

C.1.2 Searching the Wrong Phone after Scheduling Humidification

Claude Opus 4.8 executes `linux_android_smarthome_439` across Linux, two Android phones, and SmartHome. The first phone is `android_0`; the second is `android_1`.

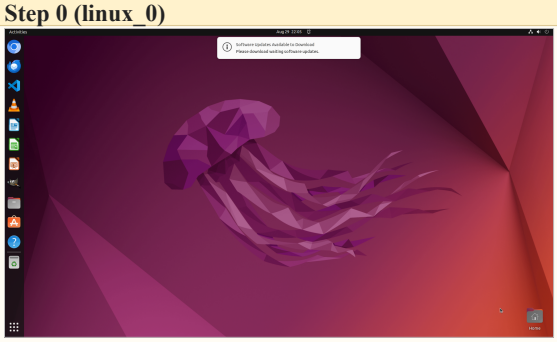
Task instruction (abridged)

The guest bedroom gets too dry overnight. Use the `Guest Wake` alarm on the first phone as the humidity-recovery time, confirm the bedroom mapping in `/tmp/home_ops/plan-revisions/source/clock_rule.csv`, and open `guest_wake_humidity.txt` in Downloads on the second phone. Schedule a Home workflow that starts the bedroom humidifier at medium when the alarm next occurs. Update and complete the existing `Guest Wake follow-up` task. Do not create a duplicate.

The displayed instruction omits one sentence requiring the note to state the 09:35 scheduled time, the bedroom humidifier's medium setting, and that the workflow was scheduled; this sentence was present in the agent's input.

The first phone contains the 09:35 alarm, the Linux file maps `Guest Wake` to the bedroom, and the second phone's file specifies the medium setting. The existing `Guest Wake follow-up` item is visible in the initial Tasks screen on `android_1`, although the instruction does not identify which phone holds it. The failure concerns locating this item, not deriving the scheduling parameters already given in the original instruction.

Step 0 (linux_0)



Action:

`cli.execute`

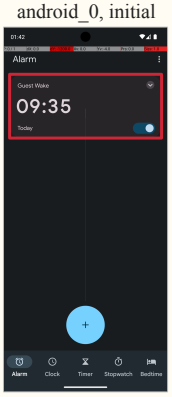
Command: cat. Target file:

`/tmp/home_ops/plan-revisions/source/clock_rule.csv`

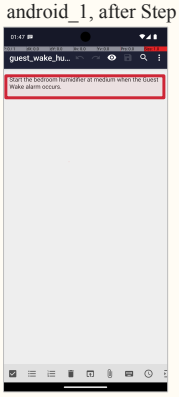
Explanation: Reads the alarm-to-room mapping. The returned row maps Guest Wake to the bedroom.

Phone 1 alarm and phone 2 file

android_0, initial



android_1, after Step 19



Action at Step 19:

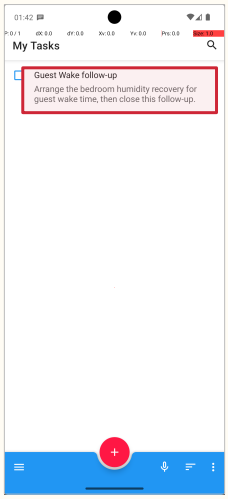
```
android.json_action
click(x=432, y=911)
```

Explanation: Opens the instruction file on android_1. It requests medium bedroom humidification when Guest Wake occurs.

Reference input: The initial alarm on android_0 shows 09:35.

Figure C.1: (b) Humidification. The Linux mapping, first-phone alarm, and setting file on the second phone.

Initial target task (android_1, second phone)



Initial state: The task is still incomplete.

Explanation: The existing Guest Wake follow-up task is on android_1, the second phone.

Required result: Update and complete this task after scheduling the humidifier.

Figure C.1: (b), continued. The original follow-up item on `android_1` before execution.

At Step 20, the agent creates a workflow for `2026-06-17T09:35:00`, with `turn_on` and `set_level(level="medium")` for the bedroom humidifier. The scheduling check passes (Figure C.1(b)).

Step 20 (home_0)

Workflow control steps

```
bedroom_humidifier_1

1  command: turn_on
   args: {}
2  command: set_level
   args: {"level": "medium"}
```

Action:

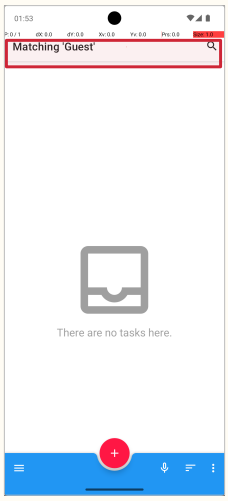
```
smarthome.schedule_workflow
workflow_id:
  guest_wake_humidity_0935
run_at: 2026-06-17T09:35:00
```

Explanation: Schedules both controls at 09:35. The call returns ok=true and creates the workflow.

Figure C.1: (b), continued. Step 20 creates the bedroom humidification workflow for 09:35.

After scheduling the workflow, the agent opens Tasks on `android_0` at Step 21. It searches for `Guest Wake follow-up`, then `Guest Wake`, and finally `Guest`, without finding the item. It does not switch to `android_1`, where the original follow-up item remains unchanged.

Step 43 (android_0, first phone)



Action:

```
android.json_action
keyboard_enter()
```

Explanation: Submits the search for "Guest" on android_0. The call succeeds, but the first phone has no matching task.

Later actions: Searches remain on the first phone. The agent never returns to the second phone to update the task.

Figure C.1: (b), continued. The search on `android_0` returns no matching item after Step 43.

C.2 A Completion Record without the Required Control Actions

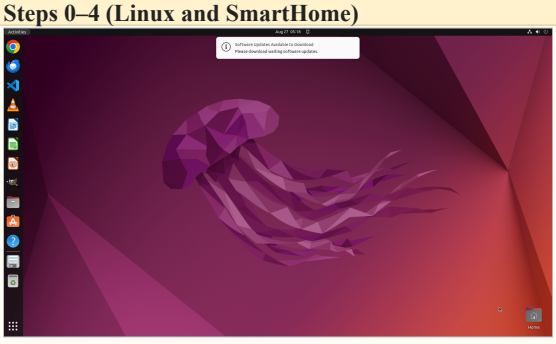
Gemini 3.1 Pro Preview executes `linux_smarthome_656`, which requires applying Linux climate rules to SmartHome devices and saving a handoff record.

Task instruction

Read `/tmp/climate/fallback_matrix.md`, check the listed rooms and live SmartHome capabilities, and apply the best permitted climate fallback for each room. Save the compact room-by-room handoff to `/tmp/climate/fallback_result.json` using the visible schema in the matrix, including the reason for any unavailable room.

The rules require turning on the office air conditioner in cool mode at 24 °C, and turning on the study heater at 22 °C. The guest room has no climate device and should be recorded as unavailable. Initially, the office air conditioner is off, in auto mode at 26 °C, and the study heater is off at 22 °C. Both require control actions to meet the requested state.

Steps 0–4 (Linux and SmartHome)



Actions at Steps 0–2:

```
Step 0  cli.execute
cat /tmp/climate/fallback_matrix.md

Steps 1–2  smarthome.
get_device_capabilities
```
device_id: office_air_conditioner_1; study_heater_1

Explanation: Reads the rule and queries capabilities. The desktop is shown after Step 3.

Figure C.2: Omitted climate control. Steps 0–2 read the rules and query device capabilities.

The agent reads the rules at Step 0 and queries the two devices' capabilities at Steps 1 and 2. Both responses list the available controls and show that the devices are off. It then writes the JSON at Step 3 and submits `done` at Step 4, without issuing a climate-control command (Figure C.2).

Step 3 write payload (linux_0)

fallback_result.json

```
{ "rooms": {
  "office": {"status": "applied",
    "device": "air conditioner",
    "target_c": 24},
  "study": {"status": "applied",
    "device": "heater",
    "target_c": 22},
  "guest room": {
    "status": "unavailable",
    "reason": "no climate device"}
} }
```

Action at Step 3:

`cli.execute`

Command: shell heredoc writes the JSON shown at left to

`/tmp/climate/fallback_result.json`

Explanation: The file is written (exit_code=0). Both rooms are marked applied, but the control actions were skipped.

Figure C.2: Continued. The JSON written at Step 3.

Step 4 (done) and final device states

Climate devices

Office air conditioner
OFF / AUTO / 26°C

Study heater
OFF / 22°C

Action at Step 4:

```
{"type": "done"}
```

Explanation: Ends without control actions. Office should be on/cool/24°C and study on/22°C. Both devices remain off.

Figure C.2: Continued. Device states when the agent submits `done` at Step 4.

The JSON marks both rooms as `applied`, although neither device has received a control command. The saved record therefore reports changes that were not made.

C.3 Failure to Adapt the Cross-Device Execution Strategy

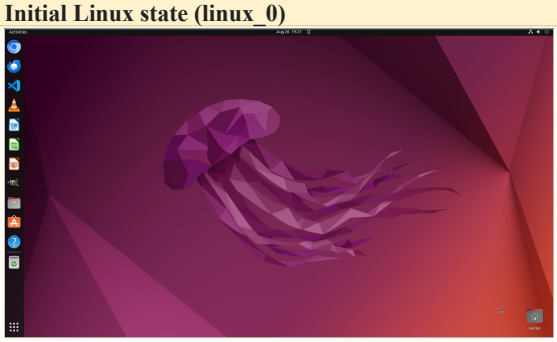
UFO^3 (GPT-5.5) executes `al_calendar_schedule_conflict`, which requires synchronizing Android meetings to a Linux schedule and recording the changes.

Task instruction

This week's meeting schedule has conflicts: Android Simple Calendar Pro and Linux `/tmp/schedule/week.csv` disagree on details for meetings with the same names. Follow `/tmp/schedule/rule.txt`: synchronize each matching meeting's date, start, end, and location from Android, leave non-matching meetings unchanged, then write `/tmp/schedule/log.json` with every field that changed and its old and new values.

The task matches meetings by name. On Linux, `week.csv` contains the title `Vendor planning`, while `rule.txt` specifies Android as the source of truth for the synchronized fields.

Initial Linux state (linux_0)



Action: None on Linux (0 actions).

Available input: /tmp/schedule/week.csv

```
Vendor planning
2027-02-13
```

Explanation: The source contains useful lookup clues, but no Linux read retrieves them.

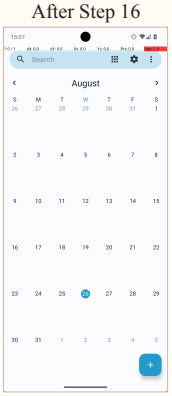
Figure C.3: Stalled calendar synchronization. The initial Linux state and an available meeting title.

UFO^3's own plan contains three subtasks: extract Android meetings, read the Linux schedule and rules, and synchronize the files with a change log. Execution starts with the Android subtask.

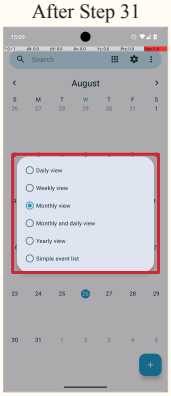
The agent changes calendar views throughout the run but does not retrieve the meeting fields needed for synchronization. The selected screens show a month view after Step 16, the view-selection dialog after Step 31, and another month view after Step 48 (Figure C.3). All 49 device interactions are on Android. Together with one initial planning step, they exhaust UFO^3's recorded 50-step budget; no revision of the overall task plan is recorded.

Calendar navigation (android_0)

After Step 16



After Step 31



Actions:

```
android.json_action
Step 16: click(index=5)
Step 31: click(index=2)
```

Explanation: The assigned subtask is to extract this week's meetings. Both clicks return ok=true, yet navigation yields no meeting records and no switch to Linux.

Figure C.3: Continued. Calendar screens after Steps 16 and 31.

Reading `week.csv` would have provided the meeting title `Vendor planning` as a search cue. The agent never reads this file. Its initial plan includes the Linux work, but execution never progresses beyond the first subtask.

Final Step 48 (android_0)



Action:

```
android.json_action
click(index=5)
```

Explanation: The screen remains a month view. No meeting data is returned and no replanning occurs.

Interaction steps: Android 49; Linux 0.
Replanning steps: 0.
Stop: 50-step limit.

Figure C.3: Continued. The final Android screen after Step 48 and the recorded interaction counts.

D Task Decomposition Study: Protocol and Additional Results

We supplement the main-text task decomposition study with its task selection, initialization, execution, and validation protocols, a guest-preparation case, and additional results. The study selects 60 cross-device tasks from the 200-task DevicesWorld-Lite set and decomposes them into 237 single-device subtasks. Each independent test supplies the prerequisites for a local goal, which the agent must then complete on the target device.

D.1 Task Selection and Decomposition Strategy

We manually select 60 cross-device tasks from DevicesWorld-Lite to cover different environment combinations, numbers of device instances, dependency structures, applications, and interfaces. Selection is based on task definitions, initialization resources, and intended outcomes, without using model execution results. To limit repetition among structurally similar tasks, at most two tasks are selected from each near-duplicate group. Table D.1 summarizes the selected tasks and their subtasks.

Table D.1: **Composition of the task decomposition study.** The selected set contains 60 original tasks and 237 single-device subtasks. Dependency-pattern counts refer to original tasks and may overlap.

Dimension	Category / statistic	Value
Original tasks ($N = 60$)		
Environment composition	Same-type multiple devices	12
	Two environment types	36
	Three environment types	12
Device instances per task	Two	28
	Three	20
	Four	12
Subtasks per original task	Range	2–7
	Mean	3.95
Dependency patterns	Chain	6
	Fan-in	54
	Fan-out	31
	Return dependency	6
Single-device subtasks ($N = 237$)		
Target environment	Android	95
	Linux	94
	SmartHome	48
Subtask objective	Information acquisition and reporting	149
	Files, application records, or device states	88

We decompose each task according to its execution topology, represented as a directed acyclic graph (DAG) of local goals. A directed edge indicates that a later local goal requires information, results, or device states produced by an earlier goal. Goals that become executable at the same dependency stage and have no ordering constraint between them are grouped in the same layer. This permits parallel execution in the dependency graph; it does not mean the independent tests run simultaneously. Subtask boundaries are determined jointly by execution layer and device instance: goals on different devices are separated even within the same layer, and goals on the same device are separated when they belong to different layers. A return dependency occurs when a later stage returns to a previously used device.

Each subtask comprises the reading, information processing, and device operations needed to complete its local goal. Individual actions, such as opening an application, clicking, entering text, and saving, are not treated as separate subtasks. Different Linux or Android instances are distinct devices; appliances within one SmartHome instance belong to the same execution environment.

Figure D.1 illustrates the decomposition by execution layer and device instance. S1 on `linux1` provides the prerequisites for S2 on `android1` and S3 on `android0`. S4 returns to `linux1` after both phone subtasks.

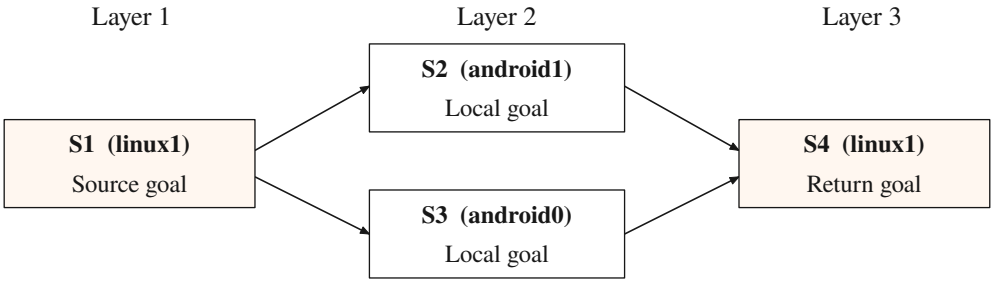


Figure D.1: **Decomposition by execution layer and device instance.** S2 and S3 share a layer but run on different phones; S1 and S4 use the same desktop at different stages. Edges indicate prerequisite dependencies. Independent tests are initialized separately.

D.2 Independent Initialization and a Recorded Case

Each independent test first restores the target device's task state and then supplies the information or state required from preceding subtasks. These prerequisites are constructed from source files, preloaded application data, and reference device states, rather than actual model outputs from predecessor tests. Consequently, a predecessor's observed success or failure does not change the input to a later independent test.

Prerequisites include source information in persistent context, completed-operation facts needed for later work, and native intermediate files or application states on the target device. Initialization may open the target application, locate preloaded records, or clear the output workspace. It supplies the current subtask's starting conditions, while the agent remains responsible for its own reading, computation, and target operations.

For example, in task `a2l2_training_media_deck_email`, independent testing of S06 initializes a JPEG from fixed task resources at `/sdcard/Pictures/diagnostic60_training_capture.jpg`. The agent must locate this image in Simple Gallery Pro and rename it to `training_setup_photo.jpg`. The initialized file represents the intermediate photo required from S04 and is not taken from an actual S04 test.

D.2.1 Guest-preparation task and its dependency structure

Task `linux_android_smarthome_338` involves one Linux desktop, two Android phones, and a SmartHome environment. For GPT-5.5, all seven independently tested subtasks pass, but the corresponding E2E run fails. The original instruction is:

> Prepare the bedroom for tonight's guest using `/tmp/home_ops/status-reports/source/coordination_note.txt`, the existing `Evening Prep Snack` recipe in Broccoli on the first phone, and the incomplete `Bedroom lighting` item in Tasks on the second phone. Rename the saved recipe to `Evening Guest Snack` and add the guest's requested tea accompaniment without changing its directions or preparation time. Set the bedroom light to the requested level, then complete the existing `Bedroom lighting` task with a brief note stating that the bedroom light was set to 55%. Do not create duplicate recipes or tasks.

The coordination note specifies lemon as the addition to the existing tea and crackers, and requests bedroom lighting at 55%. The recipe initially contains `tea; crackers`, the directions `Prepare while the room is set up.`, and a preparation time of `20 minutes`. The existing task is incomplete, and the bedroom dimmable light is off with a brightness setting of 10%. The required outputs are an in-place recipe update, the requested light state, and an updated and completed task on the other phone.

Figure D.2 separates the seven local goals by dependency layer and device. The first layer contains S01 (Linux note), S02 (recipe on `android_0`), S03 (task on `android_1`), and S04 (SmartHome state and capabilities). S05 updates the recipe using S01 and S02; S07 controls the light using S01 and S04. S06 updates and completes the task using S01, S03, S04, and the completed-light prerequisite from S07. S06 follows S07 because completing the lighting task requires the light-setting operation to have finished. Subtask identifiers follow the recorded task definitions rather than execution order. The recipe and lighting branches do not need to wait for one another.

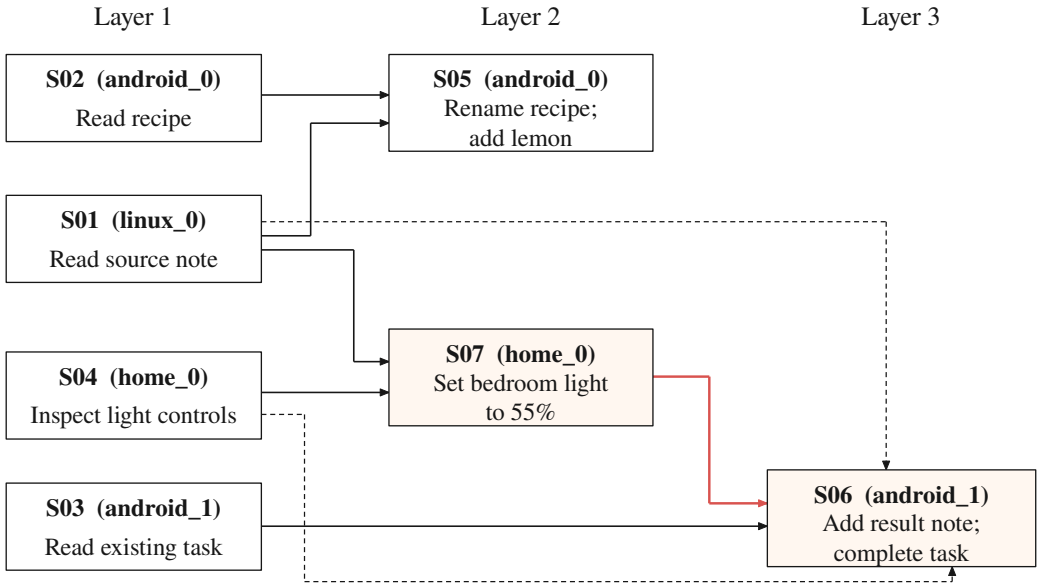


Figure D.2: **Seven local goals across four device instances.** Solid arrows show stage dependencies; the dashed arrows denote additional direct S01 and S04 source-context references to S06. The red arrow marks the completed-light prerequisite from S07 to S06. Independent tests receive reference predecessor information or states.

For S05, initialization supplies the note and original recipe fields as reference context and leaves the existing recipe unchanged. S07 starts with the light off at 10%. S06 receives the reference fact that the light-setting operation has completed, while the task on its target phone remains unmodified and incomplete.

D.2.2 Independent Subtask Executions

All seven subtasks pass independent testing, using 35 steps in total (Figure D.3). S01–S04 obtain the source note, recipe fields, existing task details, and light capabilities. In S05, the agent edits the existing recipe. After parsing failures and invalid-index attempts, it uses coordinates to enter the ingredients and saves the update. The assigned checks confirm the new title and ingredients, unchanged directions and preparation time, and preservation of the original record without duplication.

S07 changes the bedroom light from off at 10% to on at 55%. In the separately initialized S06 test, the agent receives the reference fact that the light-setting operation has completed, adds the required note, and marks the existing task complete. The assigned checks confirm the note, completion state, and absence of a duplicate task.

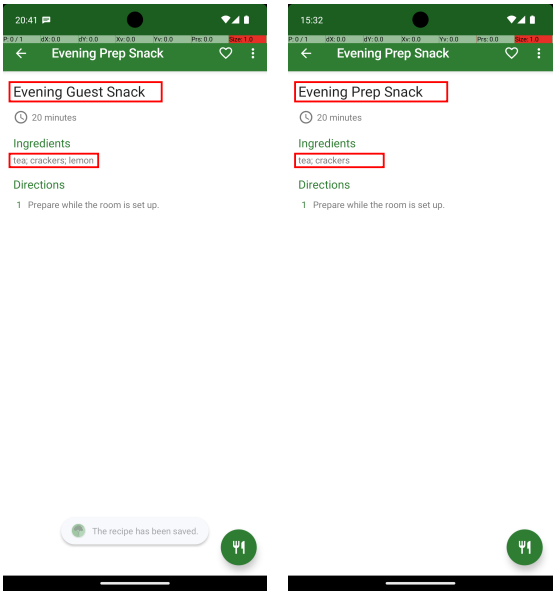
Run	Device	Execution summary	Steps	Result
S01	linux_0	Read the source note and report requirements	2	PASS
S02	android_0	Read the recipe and report its current fields	1	PASS
S03	android_1	Inspect the existing task and report its state	2	PASS
S04	home_0	Query light controls and report the current state	2	PASS
S05	android_0	Update and save the recipe after input recovery	12	PASS
S07	home_0	Set the bedroom light to 55%	2	PASS
S06	android_1	Add the light-setting note and complete the task	14	PASS

Figure D.3: **Execution summaries of the independent subtasks.** Each row summarizes one independently initialized run. All seven tests pass in 35 steps. Rows follow dependency layers, with S07 before S06.

D.2.3 Corresponding E2E execution and paired evidence

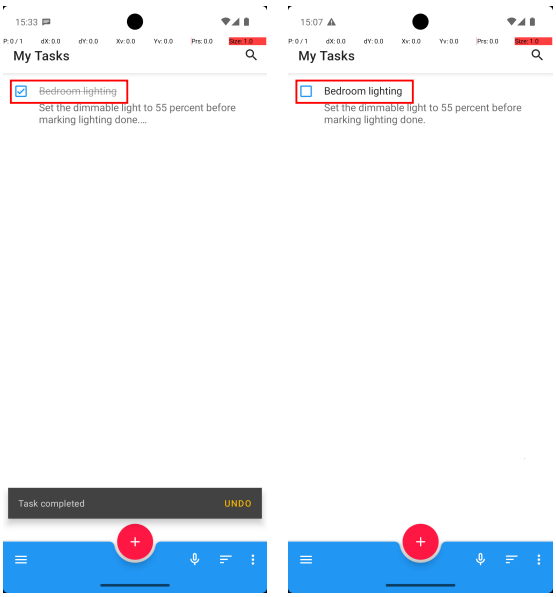
The corresponding E2E run terminates at the time limit after 49 steps. It reads the coordination note and sets the bedroom light to 55% in its first two steps. Of the 49 steps, 44 target `android_0`, four target Linux, one targets SmartHome, and none target `android_1`.

The agent repeatedly revisits the recipe details, menus, and sharing interfaces, and rereads the source note. It reaches the recipe editor and attempts to change the title, but the required update is not saved. The final recipe retains `Evening Prep Snack` and `tea; crackers`. The task on `android_1` also remains incomplete, although the lighting prerequisite has been satisfied. Final evaluation passes the light check but fails the recipe and task checks, yielding task success 0 and a score of $1/3$ (Figures D.4 and D.5).



(a) Independent S05: saved edit **(b) E2E: original recipe remains**

Figure D.4: **Recipe states in the independent S05 test and the corresponding E2E run.** Red boxes mark the recipe title and ingredients. The independent test saves `Evening Guest Snack` with lemon added, whereas the E2E run retains the original title and ingredients. In the independent screenshot, the toolbar still shows the old title; the recipe body and native record checks confirm the saved update.



(a) Independent S06: task completed **(b) E2E: second-phone task unfinished**

Figure D.5: **Task states on the second phone in the independent S06 test and the corresponding E2E run.** Red boxes highlight the existing `Bedroom lighting` task and its completion indicator. The independent test passes the checks for task completion and the required note; the E2E run leaves the task incomplete and contains no step on this phone.

D.3 Execution and Validation Protocol

Each model tests all 237 subtasks independently. The observation and action-space configurations of each subtask follow those of its original task for the same model and target device. Each independent test provides the local instruction, required predecessor information, and observations from the target device, and restricts execution to that device. Observations and actual interaction histories from other devices are not supplied. Budgets and local histories are reset for each subtask, as specified in Table D.2. Environment interfaces are detailed in Appendix A; single-device system prompts are adapted from the corresponding baselines.

Table D.2: **Configuration of independent subtask tests.** Budgets and interaction histories are reset separately for each subtask.

Setting	Independent-test configuration
Step budget	At most 30 steps and 600 seconds per subtask
Timing	Starts after initialization and predecessor-state preparation; includes model requests and device interaction. Initialization, evaluation, and cleanup are timed separately.
Observations	Same observation configuration as the original task for the target device
Action space	Same action space as the original task for the target device
History	Empty initial history; the most recent 10 local steps of textual history
Persistent context	The subtask instruction and predecessor information remain available at each step
Model requests	Temperature 0; at most 8,192 output tokens; 180-second request timeout
Device access	Only the assigned device instance

E2E outcomes reuse the corresponding model's complete-task results on the same 60 tasks from the main experiment, matched by task identifier. The main experiments use a 50-step, 1,800-second limit per complete task, whereas the independent tests allocate 30 steps and 600 seconds to each subtask separately. For the GPT-5.5 and Claude Opus 4.8 E2E runs, timing begins before environment initialization. Budgets are allocated to each independent subtask, not shared across a full decomposition.

Validation follows the subtask objective. For the 149 information-acquisition subtasks, a fixed GPT-5.5 semantic evaluator compares the agent's final information report against reference information derived from task sources. It assesses correctness, sufficiency for downstream requirements, and materially misleading content, while allowing semantically equivalent wording and additional true information. The evaluator uses temperature 0 and a maximum of 2,048 output tokens. Its input contains the original instruction, current subgoal, information-handoff requirements, reference information, and submitted report, excluding E2E outcomes and other subtasks' actual outputs. A small number of ambiguous automatic judgments are manually reviewed to determine the final result.

All evaluation checks from each original task are assigned to the corresponding subtasks. For the 88 output-producing subtasks, success requires all checks assigned to the current local goal to pass. These checks are stored in the subtask configuration and inspect files, application records, schedules, and device states. All reported rates use final adjudicated results, with every task or subtask retained in its corresponding denominator.

D.4 Detailed Results

Let $e_{m,t} \in \{0,1\}$ denote the E2E outcome of model m on original task t, and let $p_{m,t,j} \in \{0,1\}$ denote the independent outcome of its jth subtask. The three metrics are

$$\text{E2E}_m = \frac{1}{60}\sum_{t=1}^{60} e_{m,t}, \qquad \text{All-Subtask}_m = \frac{1}{60}\sum_{t=1}^{60}\prod_{j=1}^{k_t} p_{m,t,j},$$

$$\text{Subtask}_m = \frac{1}{237}\sum_{t=1}^{60}\sum_{j=1}^{k_t} p_{m,t,j}. \tag{10}$$

E2E and All-Subtask use the same 60 original tasks; Subtask uses all 237 independent subtasks. All-Subtask requires every constituent subtask to pass its separate test.

Under independent testing with explicit local goals and satisfied prerequisites, the models complete 79.75%–90.30% of subtasks and all constituent subtasks of 45.00%–70.00% of the original tasks (Table D.3). Yet many of these same tasks fail end to end. Among tasks whose subtasks all pass independently, E2E fails for 21/27 Gemini tasks (77.78%), 29/41 GPT tasks (70.73%), and 28/42 Claude tasks (66.67%). Only three or four tasks per model show the reverse outcome: E2E passes although at least one independent subtask fails.

Table D.3: **Exact success counts and paired outcomes.** Panel (a) reports counts and percentages for the three metrics. Panel (b) reports original-task counts; its four categories sum to 60 for each model.

Measure	Gemini 3.1 Pro Preview	GPT-5.5	Claude Opus 4.8
(a) Success metrics			
E2E	10/60 (16.67%)	16/60 (26.67%)	17/60 (28.33%)
All-Subtask	27/60 (45.00%)	41/60 (68.33%)	42/60 (70.00%)
Subtask	189/237 (79.75%)	213/237 (89.87%)	214/237 (90.30%)
(b) Paired outcomes on the same original tasks			
Both pass	6	12	14
All-Subtask only	21	29	28
E2E only	4	4	3
Both fail	29	15	15

D.4.1 Environment and predecessor-context breakdowns

Figure D.6 and Table D.4 break down local success by environment, with the table further separating subtasks by predecessor context. Here, predecessor context means information, completed-operation facts, or postconditions supplied from preceding work; it does not include source files, contacts, messages, or initial device states already present in the original task. Of the 237 subtasks, 92 (38.82%) receive predecessor context and 145 (61.18%) do not. Every original task contains both types. Among the 92, 91 receive source information and 22 receive completed-operation facts or states, with an overlap of 21; three also require native intermediate files on the target device.

Subtasks without predecessor context achieve success rates of 86.21%–93.10%, compared with 69.57%–85.87% for those with supplied context. The former group contains 144 information-acquisition subtasks and one output-producing subtask; the latter contains five and 87, respectively. This breakdown describes different roles in the execution topology rather than the effect of providing context to the same subtasks.

Local success exceeds 85% in every model–environment pair except Gemini on Android (51.58%).

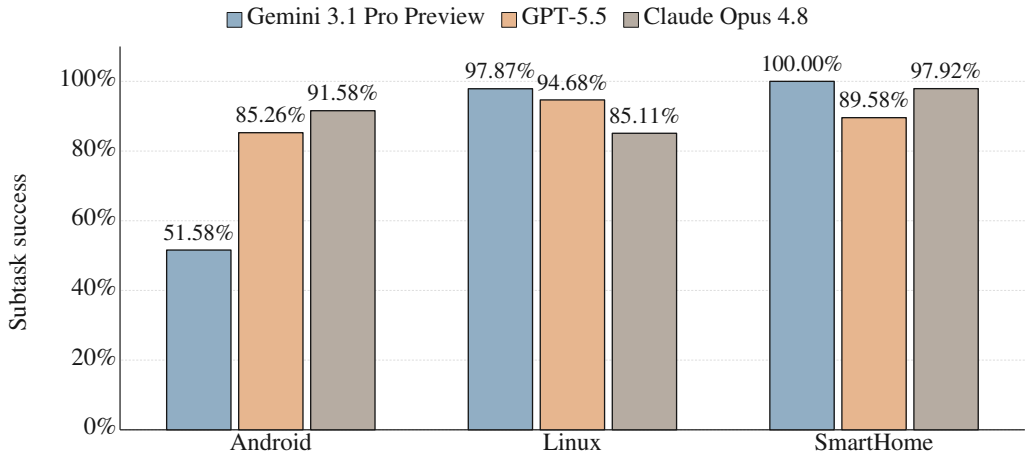


Figure D.6: **Local subtask success across the three execution environments.** Each percentage uses the environment-specific denominator: 95 Android, 94 Linux, or 48 SmartHome subtasks. The chart summarizes the overall environment rows in Table D.4.

Table D.4: **Subtask performance by environment and predecessor context.** Each result is the passed count followed by the percentage; N is the denominator for that row. "None" and "Provided" refer to predecessor context, not the availability of original task resources.

Environment	Predecessor context	N	Gemini 3.1 Pro Preview	GPT-5.5	Claude Opus 4.8
All	All	237	189 (79.75%)	213 (89.87%)	214 (90.30%)
	None	145	125 (86.21%)	134 (92.41%)	135 (93.10%)
	Provided	92	64 (69.57%)	79 (85.87%)	79 (85.87%)
Android	All	95	49 (51.58%)	81 (85.26%)	87 (91.58%)
	None	61	41 (67.21%)	55 (90.16%)	60 (98.36%)
	Provided	34	8 (23.53%)	26 (76.47%)	27 (79.41%)
Linux	All	94	92 (97.87%)	89 (94.68%)	80 (85.11%)
	None	62	62 (100.00%)	58 (93.55%)	54 (87.10%)
	Provided	32	30 (93.75%)	31 (96.88%)	26 (81.25%)
SmartHome	All	48	48 (100.00%)	43 (89.58%)	47 (97.92%)
	None	22	22 (100.00%)	21 (95.45%)	21 (95.45%)
	Provided	26	26 (100.00%)	22 (84.62%)	26 (100.00%)

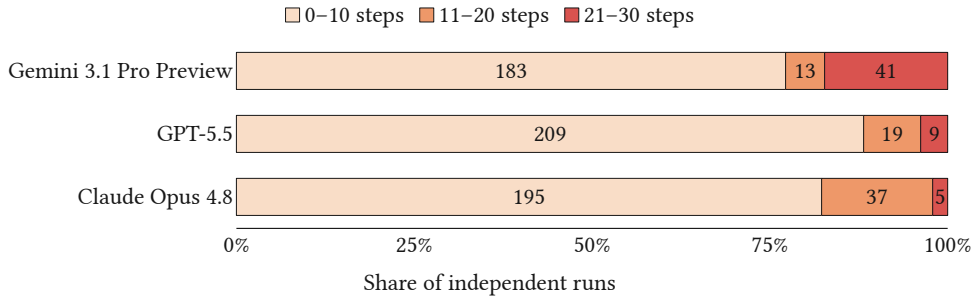
D.5 Recorded Steps and the 50-Step Reference

We report steps recorded during execution, including failed or invalid actions and the final termination step, but excluding initialization and evaluation. For each original task, S is the sum of the steps used by its independently tested subtasks. Let $s_{m,t,j}$ denote the steps of subtask j for model m on task t:

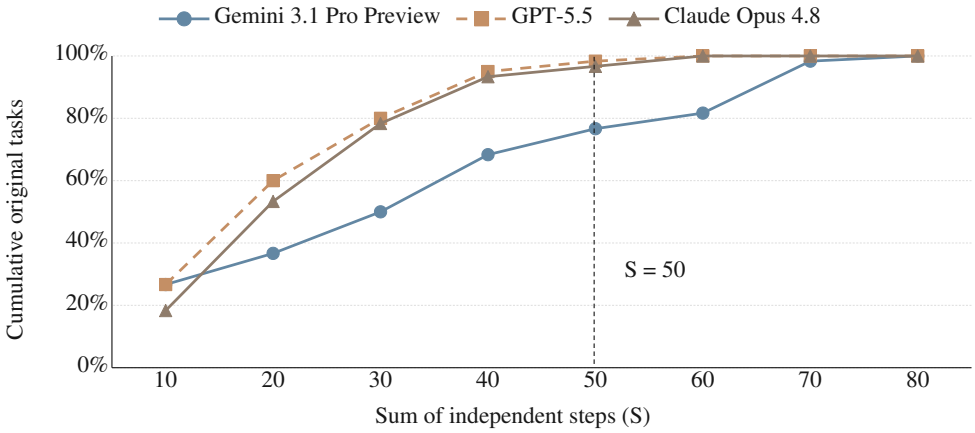
$$S_{m,t} = \sum_{j=1}^{k_t} s_{m,t,j}. \tag{11}$$

This quantity sums separate independent runs associated with one original task; it is not the measured length of a sequential execution. All selected runs are included, giving 237 subtask-level and 60 original-task-level observations per model.

Most independent runs use 0–10 recorded steps: 77.22% for Gemini, 88.19% for GPT, and 82.28% for Claude (Figure D.7(a)). These counts include both passing and failing runs. The corresponding mean lengths are 8.07, 4.86, and 5.48 steps. Summing the separate subtask runs for each original task gives $S \leq 50$ for 46/60 Gemini tasks, 59/60 GPT tasks, and 58/60 Claude tasks (Figure D.7(b)). Table D.5 gives the ranges, means, and medians. Gemini has a longer upper tail of independent subtask runs even though its median individual run is short.



(a) Independent subtask runs



(b) Sum of independent steps per original task

Figure D.7: **Recorded step profiles.** (a) Segment widths show the shares of 237 independent runs per model in the 0–10, 11–20, and 21–30 step bins, and labels give counts. (b) Cumulative percentages over 60 original tasks per model are calculated at the reported step-bin endpoints. The dashed line marks $S = 50$. Both panels include passed and failed runs.

Table D.5: **Summary of recorded steps.** Each model contributes 237 individual subtask runs and 60 sums over the subtasks of an original task. The statistics include passed and failed runs.

Measure	Gemini 3.1 Pro Preview	GPT-5.5	Claude Opus 4.8
(a) Individual independent subtask runs			
Range	1–30	0–29	1–27
Mean	8.07	4.86	5.48
Median	2	3	3
(b) Sum of independent subtask steps per original task			
Range	4–75	4–53	5–53
Mean	31.88	19.22	21.63
Median	31	17.5	20

The gap between independent subtask completion and E2E success remains among tasks whose successful independent runs sum to no more than 50 steps (Table D.6). Within this group, E2E still fails in 21/27 Gemini tasks (77.78%), 28/40 GPT tasks (70.00%), and 27/40 Claude tasks (67.50%). No Gemini task, one GPT task, and two Claude tasks both have $S > 50$ and pass all independent subtasks. This is a descriptive grouping of independent tests, not a rerun under a shared 50-step budget.

Table D.6: **Original-task counts using 50 steps as a reference.** S is the sum of independently recorded subtask steps. The final row divides the preceding failure count by the number of All-Subtask successes with $S \leq 50$.

Measure	Gemini 3.1 Pro Preview	GPT-5.5	Claude Opus 4.8
$S > 50$	14	1	2
$S > 50$, All-Subtask pass	0	1	2
$S \leq 50$, All-Subtask pass	27	40	40
$S \leq 50$, All-Subtask pass, E2E fail	21	28	27
Conditional E2E failure	21/27 (77.78%)	28/40 (70.00%)	27/40 (67.50%)