OctoNest: Adaptive Cross-Device Execution through Stateful Control

Anonymous authors
Paper under double-blind review

Abstract

Computer use agents are expanding from single-device operation toward cross-device systems that coordinate tasks across heterogeneous environments. Execution conditions are often only partially known at planning time and revealed through interaction. Failures may require intra-device modality switching or inter-device reassignment; failing to distinguish these cases can lead to repeated failures or premature termination. However, existing systems primarily scale up single-device agents without sufficiently distinguishing device-level and modality-specific execution conditions. We propose OctoNest, which coordinates stateful cross-device orchestration and iterative device-local modality control. Device Agents refine subtasks and select modalities, while an Orchestrator uses execution feedback to revise plans and device assignments. We also introduce CAPEBench, comprising 158 instances from 23 cross-device seed tasks with controlled perturbations. OctoNest leads all three quality metrics, improving Perfect Pass over the strongest baseline by 18.35 percentage points and reducing token cost per perfect pass by 39.8%. Further analyses support the complementary roles of local refinement and global revision and demonstrate CAPEBench's ability to distinguish control limitations under changing execution conditions.

1 Introduction

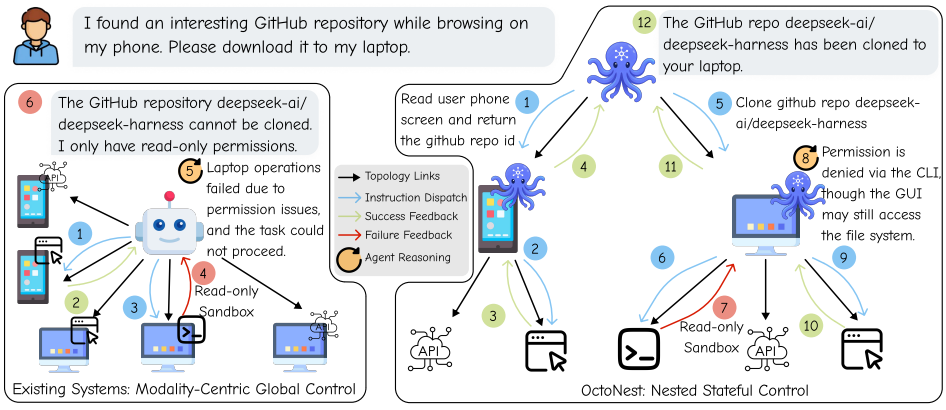


Figure 1: Modality-centric global control versus OctoNest's stateful cross-device execution, which distinguishes modality feasibility from device feasibility to guide intra-device and inter-device switching.

Recent advances in LLMs have extended them beyond conversational response generation toward agents that perceive and act in external environments (Yao et al., 2023; Schick et al., 2023). Among them, computer use agents, which operate digital applications and systems on users' behalf, have become a rapidly growing research direction (Zhou et al., 2024; Rawles et al., 2025; Xu et al., 2025a). They act through three primary execution modalities: 1) GUIs across web, desktop, and mobile (ByteDance Seed, 2025; Xu et al., 2026; Rawles et al., 2025); 2) structured APIs for direct application and service operations (Trivedi et al., 2024; Yan et al., 2025); and 3) CLIs for code,

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.

shell commands, and system operations (Yang et al., 2024; Merrill et al., 2026). Recent systems further combine multiple execution modalities, including hybrid API–GUI control and GUI interaction augmented with Python or Bash execution (Yan et al., 2025; Zhang et al., 2026; Song et al., 2026). Within a single device, these modalities provide complementary ways to accomplish a task.

In practice, many user requests span multiple devices because task-relevant information, applications, and execution capabilities are distributed across phones, personal computers, and remote machines (Xu et al., 2025b; Chen et al., 2026b; Li et al., 2026a). CRAB centralizes perception across environments, planning, and action selection in a primary agent (Xu et al., 2025b). TopoClaw similarly relies on a primary agent to coordinate multiple devices while making both planning and execution decisions (Huang et al., 2026). UFO^3 separates global graph orchestration from device execution, maintaining a mutable task graph and dispatching its nodes to device agents tied to device-specific execution modalities (Zhang et al., 2025c). Despite their different organizations, these systems coordinate endpoints bound to a modality, grounding cross-device decisions into the action spaces exposed by their modalities and returning observations to the global agent.

Cross-device execution depends on both global task dependencies and execution states local to each device and its modalities. However, existing systems largely leave the global agent to interpret modality-level observations for cross-device coordination, without sufficiently distinguishing between adaptation within a device and reassignment across devices. For example, a write operation blocked by a CLI's read-only sandbox may still be completed through the GUI on the same device, whereas a device-wide network disconnection may make a network-dependent subtask infeasible on that device. Access to modalities subject to different runtime constraints can enable continued execution when one modality becomes unavailable (Hu et al., 2026; Zhou et al., 2026), while failures that leave no feasible alternative on the current device require reassignment. The states governing these possibilities are only partially known at planning time and progressively revealed through interaction (Zhu et al., 2026; Bonatti et al., 2025), making execution a failure-prone process of state discovery. This motivates supporting multiple modalities and updating execution state from feedback to distinguish modality feasibility from device feasibility, guiding intra-device or inter-device switching while preserving task constraints.

We propose **OctoNest**, a stateful cross-device execution framework that coordinates intra-device and inter-device switching through nested control. The orchestrator maintains orchestration state to plan and assign device-level subtasks. Each device agent maintains device-local state to progressively refine its subtask, select API, CLI, or GUI modality executors, and switch modalities when needed. When no viable local continuation can be identified, it returns execution evidence to the orchestrator, which revises the remaining plan and device assignments. As illustrated in Figure 1, this design preserves progress through local adaptation when possible and enables reassignment when necessary, supporting continued execution under changing conditions.

We introduce **CAPEBench** (*Cross-device Adaptation to Perturbations in Execution*), comprising 158 instances derived from 23 seed tasks. CAPEBench covers unperturbed, modality-level, device-level, and mixed conditions, with modality-level perturbations testing intra-device switching, device-level perturbations testing inter-device switching, and mixed perturbations testing both. Compared with the strongest baseline, OctoNest improves Completion, Adherence, and Perfect Pass by 10.51, 11.36, and 18.35 percentage points, respectively, while reducing token cost per perfect pass by 39.8%. Further analysis shows that OctoNest selects capability-compatible modality sequences and achieves higher affected-step recovery under perturbations, supporting the complementary roles of device-local control and cross-device orchestration.

Our contributions are threefold:

- We propose **OctoNest**, a stateful cross-device execution framework that unifies intra-device and inter-device switching. By distinguishing modality feasibility from device feasibility, OctoNest enables agents to adapt at the appropriate level as state is revealed during execution.
- We distinguish device-local modality decisions from cross-device planning: device agents use execution feedback to refine subtasks, select and combine API, CLI, and GUI modalities, and perform intra-device switching, while the orchestrator uses device-level outcomes to revise the remaining plan and perform inter-device switching. This enables local recovery when an alternative modality remains and cross-device reassignment when local execution is no longer viable.

- We introduce **CAPEBench**, a cross-device execution benchmark with controlled perturbations that diagnoses whether agents recognize interaction-revealed changes in device and modality state and choose appropriately between intra-device and inter-device switching. On CAPEBench, OctoNest improves Perfect Pass by 18.35 percentage points and cuts the token cost per perfect pass by 39.8% over the strongest baseline.

2 METHODOLOGY

2.1 PROBLEM FORMULATION

We consider a natural-language instruction I to be completed over a set of heterogeneous devices $\mathcal{D}$, subject to task dependencies and device assignment constraints $\mathcal{C}_I$. Each device $d \in \mathcal{D}$ exposes a set of execution modalities $\mathcal{M}_d$, such as API, CLI, and GUI. At time t, its latent execution state is represented as

$$x_d^t = \left(x_{d,\mathrm{dev}}^t, \{x_{d,m}^t\}_{m \in \mathcal{M}_d}\right)$$

where $x_{d,\mathrm{dev}}^t$ captures conditions shared across modalities on device d, such as connectivity, while $x_{d,m}^t$ captures modality-specific conditions, such as access permissions and authentication.

For an unfinished subtask goal g, let $\mathcal{R}_d(g; x_d^t)$ denote the set of feasible executions that complete g on device d under the current state and applicable task constraints. Each execution may involve one or more modalities.

Within a device, the failure of one modality does not imply that $\mathcal{R}_d(g; x_d^t)$ is empty. If g remains feasible using an alternative modality, execution can continue through *intra-device switching*, which changes modalities while retaining the subtask's device assignment. Across devices, when

$$\mathcal{R}_d(g; x_d^t) = \varnothing, \qquad \exists d' \in \mathcal{D} \setminus \{d\} : \mathcal{R}_{d'}(g; x_{d'}^t) \neq \varnothing$$

completing the subtask requires *inter-device switching*, which reassigns the unfinished goal from d to a feasible device d'. Both forms of switching must respect $\mathcal{C}_I$.

These feasibility conditions depend on latent execution states that are only partially known at planning time (Chen et al., 2026a). Agents must therefore use evidence revealed through interaction to assess whether execution can continue within the current device or requires reassignment across devices. The objective is to complete I while satisfying $\mathcal{C}_I$, adapting execution through intra-device or inter-device switching as task-relevant state is revealed.

2.2 OCTONEST OVERVIEW

As shown in Figure 2, OctoNest maintains execution state at two levels: orchestration state and device-local state. The Orchestrator uses orchestration state to plan and assign device-level subtasks. Each Device Agent uses device-local state to refine its subtask into instructions for API, CLI, or GUI Modality Executors, incorporating their feedback to guide execution. When feedback indicates that another modality offers a viable continuation, it performs intra-device switching; when no viable local continuation can be identified, it returns execution evidence to the Orchestrator. Device-level outcomes update orchestration state, enabling the Orchestrator to propagate results, revise the remaining plan, and perform inter-device switching when reassignment is needed.

2.3 ORCHESTRATION STATE AND INTER-DEVICE SWITCHING

Cross-device coordination requires tracking subtask dependencies, propagating intermediate results, and accounting for device feasibility (Zhang et al., 2025c), while revising the remaining plan as execution reveals new evidence (Sun et al., 2023). The Orchestrator supports these requirements through orchestration state, represented at coordination step t as

$$s_t^O = (P_t, \mathcal{Y}_t, \mathcal{F}_t)$$

where P_t is the current global plan, $\mathcal{Y}_t$ contains completed subtask results, and $\mathcal{F}_t$ contains execution feedback and context available for assessing device feasibility and revising the remaining plan. The plan is represented as a dependency-ordered subtask chain

$$P_t = \langle q_1, \ldots, q_K \rangle, \qquad q_j = (g_j, d_j, c_j, z_j)$$

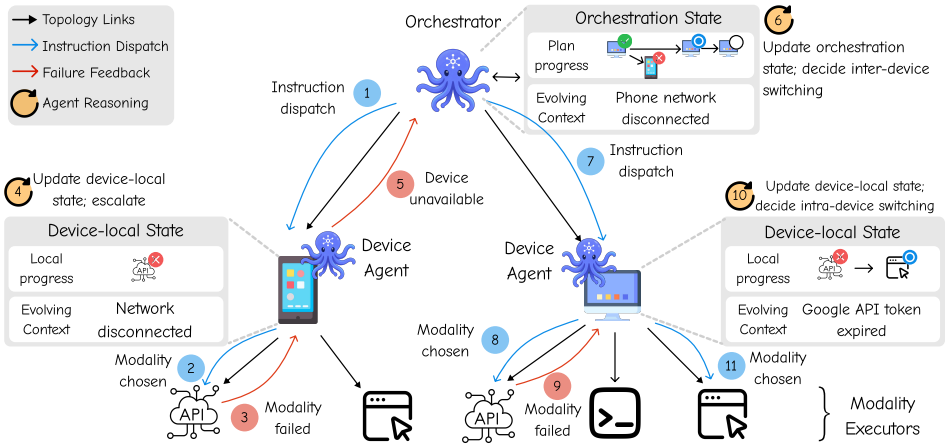


Figure 2: Overview of OctoNest. The Orchestrator maintains orchestration state for device-level coordination and inter-device switching, while each Device Agent maintains device-local state to progressively ground subtasks and coordinate API, CLI, and GUI Modality Executors.

where K is the number of subtasks in the current plan, g_j specifies the subtask objective, d_j is its assigned device, c_j contains execution context required by the subtask, and z_j denotes its execution status.

Given the instruction, task constraints, and available device profiles, the Orchestrator initializes P_0 and dispatches the next executable subtask to its assigned Device Agent. After local execution terminates, the Device Agent returns an outcome r_j, containing the execution status, available results, and execution context. This outcome updates the orchestration state:

$$s^O_{t+1} = \mathcal{U}_O\left(s^O_t, q_j, r_j\right)$$

where $\mathcal{U}_O$ denotes the orchestration state update. For a successful outcome, the Orchestrator records the result and propagates required information to dependent subtasks. On escalation, it uses the returned evidence to revise the unfinished plan, potentially refining subtasks or reassigning unfinished work to another device consistent. Reassigning unfinished work to a different device constitutes inter-device switching.

2.4 Device Agent: Device-Local State and Intra-Device Switching

Assigning a subtask to a device leaves open how it should be carried out through the available modalities. The next executable step depends on local progress and modality conditions revealed through interaction, so execution requires progressively refining the subtask and adapting modality choices as evidence accumulates (Prasad et al., 2024; Zhang et al., 2026). The Device Agent manages this process by maintaining device-local state and coordinating the available Modality Executors. For a subtask q_j assigned to device $d = d_j$, its state $s^d_{j,\ell}$ at local step ℓ retains prior execution attempts, their outcomes, and the interaction context used to assess local progress and modality feasibility.

At each local step, the Device Agent makes one of three decisions:

$$\pi_d\left(q_j, s^d_{j,\ell}, \mathcal{M}_d\right) \rightarrow \left\{\textsc{Execute}(m,u), \textsc{Done}(y_j), \textsc{Escalate}(e_j)\right\}$$

where π_d denotes the decision process of the Device Agent on device d. Execute issues instruction u through modality m, Done returns the completed subtask result y_j, and Escalate returns execution evidence e_j for orchestration-level replanning.

At local step ℓ, an Execute decision selects modality $m \in \mathcal{M}_d$ and issues instruction u. The selected Modality Executor returns execution feedback $\mathcal{E}_{d,m}(u)$, which updates the device-local state:

$$s^d_{j,\ell+1} = \mathcal{U}_D\big(s^d_{j,\ell}, m, u, \mathcal{E}_{d,m}(u)\big),$$

where $\mathcal{U}_D$ denotes the device-local state update.

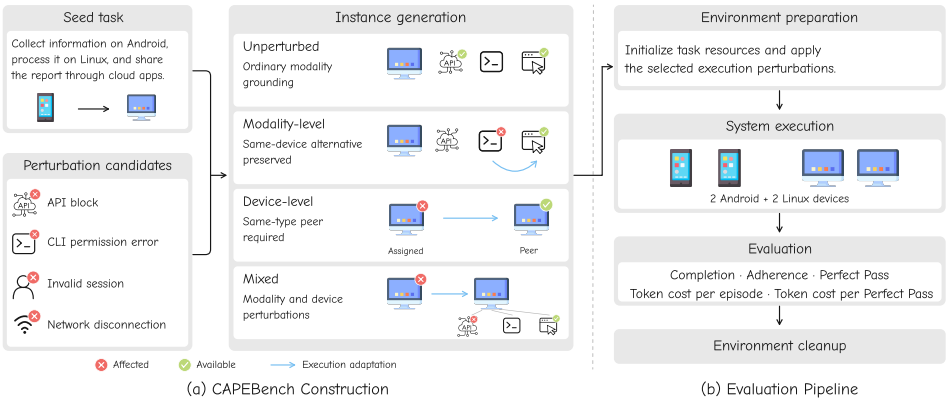


Figure 3: CAPEBench construction (left) and evaluation pipeline (right).

A successful modality step does not necessarily complete the device-level subtask; the Device Agent checks remaining requirements and generates another step when needed. After a failed step, the Device Agent uses execution feedback to revise the instruction or select an alternative modality; the latter constitutes intra-device switching. If no viable local continuation is identified or the local execution budget is exhausted, it escalates to the Orchestrator. A DONE or ESCALATE decision terminates local execution and produces the device-level outcome r_j, including execution context.

2.5 MODALITY EXECUTORS

API, CLI, and GUI execution involve distinct action spaces and observations (Trivedi et al., 2024; Rawles et al., 2025). OctoNest uses a dedicated Modality Executor for each modality to translate instructions into concrete actions and interpret their feedback (Erdogan et al., 2025; Yao et al., 2023), separating modality-specific execution from device-local coordination. Given an instruction u, the executor for modality m on device d performs one or more actions and returns

$$\mathcal{E}_{d,m}(u) \rightarrow (\sigma, y, o),$$

where σ indicates execution success or failure, y contains any available output, and o records execution observations, including failure details when applicable. This feedback is incorporated into device-local state to guide subsequent execution. The API Executor invokes structured application or service operations; the CLI Executor performs command-line, computation, and file-system operations; and the GUI Executor interacts with applications through visual observations and interface actions. Implementation details are provided in Appendix C.2.

3 EXPERIMENTS

3.1 CAPEBENCH AND EXPERIMENTAL SETUP

CAPEBench comprises 158 instances derived from 23 cross-device seed tasks, each instantiated under a specified runtime condition. It runs on two Linux and two Android devices with real application services and local file systems. Each device exposes a subset of API, CLI, and GUI modalities, with only partial service API coverage, requiring agents to ground task steps through the available modalities. As illustrated in Figure 3, CAPEBench covers unperturbed, modality-level, device-level, and mixed conditions (Zhu et al., 2026; Treviño et al., 2025; Jia et al., 2026). Unperturbed instances evaluate ordinary modality grounding. Modality-level perturbations preserve a same-device alternative to test intra-device switching; device-level perturbations prevent local completion while preserving a same-type peer to test inter-device switching; mixed perturbations test both. Perturbations are applied deterministically to the environment, with task-specific effects revealed through execution. Appendix B.3 details their taxonomy and injection procedure.

Following prior agent evaluations (Ma et al., 2024), we report three quality metrics. **Task Completion** (Comp.) measures the percentage of required service and file-state postconditions satisfied.

Table 1: Main results on CAPEBench. Bold and underlined values mark the best and second-best results. Brackets show pointwise 95% confidence intervals for changes from paired seed-task bootstrap.

System		Quality			Efficiency	
Method	**Exec.**	**Comp. (%) ↑**	**Adh. (%) ↑**	**PP (%) ↑**	**Tok./Ep. ↓**	**Tok./PP ↓**
CRAB	GUI	2.37	16.19	0.00	545,102	∞
CRAB	API	22.95	38.39	0.00	<u>513,463</u>	∞
TopoClaw	Hybrid	46.86	52.17	5.70	933,484	16,387,826
UFO^3	GUI	47.06	50.94	10.13	1,452,661	14,345,029
UFO^3	API	59.28	65.44	0.00	**426,608**	∞
UFO^3	Hybrid	<u>65.00</u>	<u>65.46</u>	<u>22.78</u>	702,893	<u>3,084,917</u>
OctoNest	Hybrid	**75.51**	**76.82**	**41.14**	764,552	**1,858,449**
Change vs. UFO^3-Hybrid		+10.51	+11.36	+18.35	+8.8%	−39.8%
95% CI of changes		[2.79, 18.22]	[5.03, 17.69]	[7.33, 28.75]	[−9.4, 31.9]%	[−58.8, −11.6]%

Table 2: Quality and efficiency across runtime conditions compared with UFO^3-Hybrid.

Condition	Method	Comp. (%) ↑	Adh. (%) ↑	PP (%) ↑	Tok./PP ↓
Unperturbed	UFO^3-Hybrid	73.19	70.97	30.43	2,146,184
	OctoNest	**81.81** (+11.8%)	**86.05** (+21.2%)	**52.17** (+71.4%)	**1,151,222** (−46.4%)
Modality	UFO^3-Hybrid	82.98	79.43	38.10	1,775,644
	OctoNest	**84.64** (+2.0%)	**85.94** (+8.2%)	**54.76** (+43.8%)	**1,342,100** (−24.4%)
Device	UFO^3-Hybrid	55.69	60.79	19.44	3,321,659
	OctoNest	**62.59** (+12.4%)	**68.41** (+12.5%)	**33.33** (+71.4%)	**2,644,824** (−20.4%)
Mixed	UFO^3-Hybrid	54.33	55.89	10.53	7,395,304
	OctoNest	**74.39** (+36.9%)	**71.68** (+28.3%)	**31.58** (+200.0%)	**2,465,462** (−66.7%)

Instruction Adherence (Adh.) measures the fraction of gold steps completed with the intended semantics on an allowed device; modality-level perturbations retain the original device constraint, whereas device-level perturbations permit a same-type peer. **Perfect Pass** (PP) measures the percentage of episodes in which every gold step is completed with the intended semantics on an allowed device. Efficiency metrics are average tokens per episode (**Tok./Ep.**) and tokens per perfect pass (**Tok./PP**), computed as Tok./Ep. divided by the perfect-pass rate.[1] See Appendix D.2 for details.

We compare three representative cross-device systems. CRAB pioneers coordinated interaction across desktop and mobile environments (Xu et al., 2025b). UFO^3 supports adaptive execution through explicit task dependencies and feedback-driven task-graph updates (Zhang et al., 2025c); we extend it with API, CLI, and GUI execution to form UFO^3-Hybrid. TopoClaw supports topology-aware coordination through a primary agent that accounts for device capabilities and user context (Huang et al., 2026). All methods use a 30-minute execution timeout. Unless otherwise stated, text-based agents use DeepSeek-V4-Pro and vision-based agents use Kimi K2.5; system-specific configurations are detailed in Appendix C.3. Main comparisons use **Full-158**; selected analyses use a fixed sample of 82 instances detailed in Appendix B.

3.2 RQ1: How effective and efficient is OctoNest?

Table 1 shows that OctoNest achieves the highest Completion, Adherence, and Perfect Pass among the evaluated systems. Compared with UFO^3-Hybrid, it improves all three quality metrics, includ-

[1] Tok./PP includes tokens spent on unsuccessful episodes and is reported as ∞ when no episode achieves Perfect Pass.

ing a gain of 18.35 percentage points in Perfect Pass, with the pointwise 95% confidence interval for each improvement excluding zero. Although mean token usage per episode is higher, token cost per perfect pass decreases by 39.8%. The improvement in efficiency therefore comes from a higher yield of successful executions, rather than lower token expenditure on each attempt. The results suggest that execution organization matters alongside modality availability. CRAB and TopoClaw rely on primary-agent control and achieve limited end-to-end success, even with TopoClaw's hybrid execution. UFO^3-Hybrid separates cross-device planning from device-level execution and achieves substantially higher Perfect Pass, consistent with the value of hierarchical task organization. OctoNest further improves Perfect Pass despite sharing API, CLI, and GUI access with UFO^3-Hybrid. This gap motivates examining how device-local modality decisions interact with cross-device orchestration, using UFO^3-Hybrid as the primary behavioral comparison.

Table 2 shows improvements in all three quality metrics and lower token cost per perfect pass across all four runtime conditions. Without perturbations, OctoNest achieves higher Perfect Pass, indicating that the benefits extend to ordinary task execution, where agents must still select suitable modalities and satisfy device constraints. Under modality-level perturbations, similar Completion but substantially different Perfect Pass reveals a gap between average final state satisfaction and fully adherent execution. OctoNest's advantage in this condition is particularly apparent in completing the entire required execution correctly, where intra-device switching can preserve progress after a modality becomes unavailable. Device-level and mixed perturbations additionally require agents to continue execution beyond the originally assigned device. OctoNest achieves higher Perfect Pass under device-level perturbations. Under mixed perturbations, which exercise both intra-device and inter-device switching, it improves both Completion and Perfect Pass. Higher success also reduces token cost per perfect pass by 66.7%, the largest reduction among the four conditions. Nevertheless, Perfect Pass remains below one-third in this setting, showing that reliable execution under combined modality and device constraints remains challenging.

3.3 RQ2: Which components contribute to OctoNest's performance?

We first examine the contributions of device-local and orchestration-level control on the 82-instance sample (Table 3). Removing the Device Agent substantially reduces Perfect Pass, showing the value of refining assigned subtasks before and during modality-specific execution. Without this layer, the Orchestrator must manage cross-device task progress while also issuing instructions directly to Modality Executors. A one-shot Device Agent retains the intermediate refinement step but reduces Perfect Pass from 43.90% to 14.63%: its initial modality choice and execution instruction cannot be revised in response to subsequent observations. Unifying execution across all modalities within a single Device Agent also substantially reduces Perfect Pass, supporting the separation of device-local coordination from modality-specific execution. Together, they support iterative device-local control that uses feedback to refine instructions and modality choices while delegating concrete actions to modality-specific executors.

Table 3: Architectural ablations. Quality metrics are percentages; Tok./PP is in millions.

Method	Comp.	Adh.	PP	Tok./PP
OctoNest	**75.55**	**77.49**	**43.90**	**1.631**
No Device Agent	56.59	58.02	18.29	2.237
One-shot Device Agent	57.52	56.21	14.63	3.778
Unified Device Agent	51.30	55.51	18.29	3.599
Reactive Orchestrator	62.70	64.47	32.93	1.931
Fixed orchestration plan	40.14	47.68	21.95	2.214
Failure-only device feedback	46.20	50.38	30.49	2.012

At the orchestration level, the Reactive Orchestrator retains execution history but chooses successive assignments without maintaining an explicit plan for the remaining task. Its lower performance supports the value of organizing future assignments and their dependencies, providing a persistent structure for coordinating progress across devices. The fixed-plan variant preserves this structure but cannot revise it when execution reveals that an assignment is no longer viable. Its substantial degradation shows why device-local adaptation alone is insufficient: when local execution escalates a failure, the fixed plan cannot introduce recovery assignments, and execution terminates. These results support an Orchestrator that combines an explicit future plan with execution-conditioned revision. Failure-only device feedback further shows that revision requires evidence about what went wrong. Audited traces reveal an approximately even split between transferring failed subtasks and retrying them on the original device, reflecting difficulty distinguishing device infeasibility from locally recoverable execution problems. Repeated attempts add overhead, and many tasks are aborted

after a second failure. Informative device feedback therefore helps the Orchestrator determine when to preserve an assignment and when inter-device switching is needed.

3.4 RQ3: How does OctoNest adapt during execution?

Progressive modality grounding. We examine how agents select and combine modalities to complete the parts of unperturbed CAPEBench instances that require execution on a single device. *Feasible* measures coverage of any sequence capable of satisfying the task requirements. *Preferred* measures coverage of the annotated sequence expected to fulfill these requirements straightforwardly and efficiently. *Composed* restricts preferred coverage to task parts requiring multiple modalities in the annotated preferred sequence. Coverage requires the reference modalities to appear in order within the observed execution sequence; unreached parts count as uncovered.

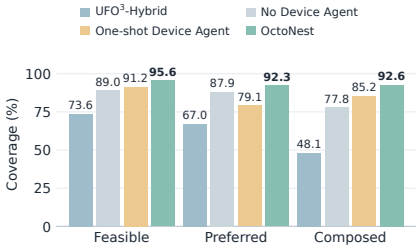


Figure 4: Modality-sequence coverage on unperturbed single-device task parts.

Figure 4 shows higher feasible and composed coverage for OctoNest than UFO^3-Hybrid, indicating better selection of capability-compatible sequences and composition of complementary modalities. Higher preferred coverage than the one-shot Device Agent supports progressively decomposing subtasks into finer steps with suitable modalities, rather than requiring one modality to complete the entire subtask. These gains arise without injected failures, showing that device-local refinement benefits ordinary execution as well as recovery.

Intra-device and inter-device recovery. We evaluate intra-device and inter-device recovery under single modality-level and device-level perturbations, respectively, restricting analysis to cases with a recovery destination where both systems encounter the failure. *Boundary match* measures whether recovery stays on the assigned device under modality-level perturbations or moves to the required peer under device-level perturbations. *Affected-step recovery* additionally requires every affected gold step to be completed with full adherence.

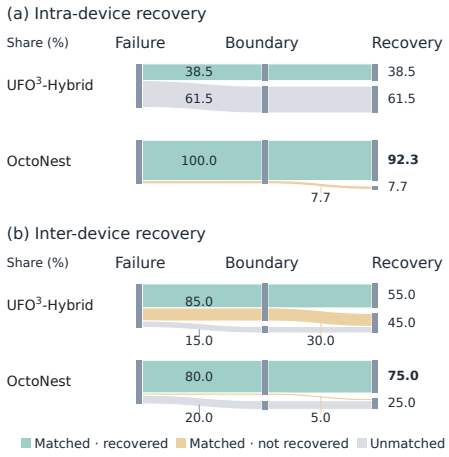


Figure 5: Intra-device (a) and inter-device (b) recovery on instances where both systems encounter the induced failure.

Under modality-level perturbations, OctoNest consistently keeps recovery on the assigned device and achieves higher affected-step recovery than UFO^3-Hybrid (Figure 5(a)). This shows that local alternatives can support continued execution without unnecessary device reassignment. Under device-level perturbations, OctoNest achieves higher affected-step recovery despite a slightly lower boundary match rate (Figure 5(b)). The orange branch shows that more baseline cases reach the required peer but still fail to recover, highlighting the importance of successful continuation after reassignment. In audited UFO^3-Hybrid traces, capability-mismatched modality choices and overly broad instructions prevent successful continuation, sometimes exhausting executor budgets. Corresponding OctoNest traces show the destination Device Agent breaking down the transferred subtask and selecting modalities suited to local execution conditions. Together with the orchestration ablations, these observations support complementary roles for the two control levels: the Orchestrator revises device assignments, while the Device Agent refines execution within the selected device.

3.5 RQ4: How does CAPEBench diagnose control limitations?

We pair perturbation families with architectural interventions to examine whether CAPEBench distinguishes limitations in different control capabilities.

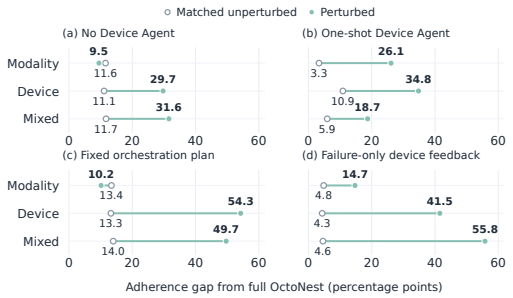


Figure 6: Adherence gaps from full OctoNest on matched unperturbed and perturbed tasks.

Diagnostic sensitivity. For each intervention, Figure 6 compares its Adherence gap from full OctoNest under perturbation with the corresponding gap on matched unperturbed tasks. The change between the two endpoints captures the additional penalty beyond the gap already present without perturbations. The interventions exhibit distinct responses. With a one-shot Device Agent, the gap widens substantially under modality-level perturbations, showing the importance of refining local execution when the initially selected modality becomes unavailable. With a fixed orchestration plan, the gap changes little under modality-level perturbations but widens by 41.0 and 35.7 percentage points under device-level and mixed perturbations, respectively. This contrast reflects the different control requirements: same-device alternatives can be handled locally, whereas device-level obstacles require a revised assignment. Failure-only device feedback exhibits its largest gap expansion under mixed perturbations, where deciding between local continuation and device reassignment requires informative execution evidence. Without such evidence, the Orchestrator has difficulty determining which control response is appropriate. Together, these patterns show that CAPEBench reveals how different control limitations respond to changing execution conditions, beyond the performance differences already visible on unperturbed tasks.

4 Related Work

LLM agents operate through heterogeneous execution modalities, including tools and APIs (Karpas et al., 2022; Patil et al., 2024; Qin et al., 2024; Ruan et al., 2024), web, desktop, and workplace environments (Koh et al., 2024; Drouin et al., 2024; Li et al., 2026b), mobile applications (Zhang et al., 2025d; Xu et al., 2026), and code or command-line interfaces (Yang et al., 2024; Merrill et al., 2026). Recent systems combine these modalities by comparing or integrating API and GUI control (Zhang et al., 2025a; Yan et al., 2025), augmenting GUI interaction with Python and Bash execution (Song et al., 2026), and unifying browser, shell, and extensible tool use (Steinberger & OpenClaw Contributors, 2026). As these capabilities diversify, multi-agent systems organize specialized agents through role-based interaction and structured workflows (Wu et al., 2024; Chen et al., 2024; Hong et al., 2024; Qian et al., 2024); systems such as Magentic-One, UFO, AgentOrchestra, and OWL further separate high-level planning and delegation from specialized execution (Fourney et al., 2024; Zhang et al., 2025b;e; Hu et al., 2025). Cross-device research extends this organization across distinct device environments: AppCopilot incorporates cross-app and cross-device orchestration on Android devices (Fan et al., 2025); CRAB supports and evaluates desktop–mobile tasks (Xu et al., 2025b); UFO^3 coordinates desktops, servers, mobile devices, and edge endpoints through mutable task constellations (Zhang et al., 2025c); and TopoClaw relies on a primary agent to coordinate planning and execution across devices while accounting for device capabilities (Huang et al., 2026). OctoNest distinguishes modality feasibility from device feasibility during execution.

5 Conclusion

We presented OctoNest, which coordinates device-local modality control and cross-device orchestration to support intra-device and inter-device switching as execution state is revealed through interaction. We introduced CAPEBench to evaluate these capabilities under controlled modality-level, device-level, and mixed perturbations. Experiments show improved execution quality and lower token cost per perfect pass. Our analyses highlight complementary requirements for reliable multi-device execution: device-local control must progressively decompose subtasks and select suitable modalities, while orchestration must maintain and revise the remaining plan using execution feedback. CAPEBench reveals distinct degradation patterns under architectural interventions, exposing limitations obscured by unperturbed evaluation. These findings highlight the importance of coordinating local refinement and global plan revision under partially known execution conditions.

REFERENCES

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 *International Conference on Machine Learning (ICML)*, 2025. URL https://proceedings.mlr.press/v267/bonatti25a.html.

ByteDance Seed. UI-TARS-2 Technical Report: Advancing GUI Agent with Multi-Turn Reinforcement Learning. *arXiv preprint arXiv:2509.02544*, 2025. URL https://arxiv.org/abs/2509.02544.

Chaoran Chen, Vy Nguyen, Ziji Zhang, Abhinav Gullapalli, Ziyi Wang, Yuxuan Lu, Dakuo Wang, Jing Huang, Zhou Yu, and Jin Lai. Retry, Switch, or Abstain? Learning Strategy-Aware Tool-Use Policies via Controlled Error Injection. *arXiv preprint arXiv:2608.11977*, 2026a. URL https://arxiv.org/abs/2608.11977.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. AgentVerse: Facilitating Multi-Agent Collaboration and Exploring Emergent Behaviors. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/578e65cdee35d00c708d4c64bce32971-Abstract-Conference.html.

Zixiang Chen, Yuheng Lu, Zihao Cheng, Zeming Liu, Jizeng Bai, Ziye Huang, Zhiyin Lin, Zihan Li, Yuhang Guo, Yunhong Wang, and Haifeng Wang. JarvisGUI: Towards Cross-Device GUI Agents with Dynamic Task Composition. *arXiv preprint arXiv:2609.10451*, 2026b. URL https://arxiv.org/abs/2609.10451.

Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H. Laradji, Manuel Del Verme, Tom Marty, David Vázquez, Nicolas Chapados, and Alexandre Lacoste. WorkArena: How Capable Are Web Agents at Solving Common Knowledge Work Tasks? In *International Conference on Machine Learning (ICML)*, 2024. URL https://proceedings.mlr.press/v235/drouin24a.html.

Lutfi Eren Erdogan, Nicholas Lee, Sehoon Kim, Suhong Moon, Hiroki Furuta, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. Plan-and-Act: Improving Planning of Agents for Long-Horizon Tasks. In *International Conference on Machine Learning (ICML)*, 2025. URL https://proceedings.mlr.press/v267/erdogan25a.html.

Jingru Fan, Yufan Dang, Jingyao Wu, Huatao Li, Runde Yang, Xiyuan Yang, Yuheng Wang, and Chen Qian. AppCopilot: Toward General, Accurate, Long-Horizon, and Efficient Mobile Agent. *arXiv preprint arXiv:2509.02444*, 2025. URL https://arxiv.org/abs/2509.02444.

Adam Fourney, Gagan Bansal, Hussein Mozannar, Cheng Tan, Eduardo Salinas, Erkang Zhu, Friederike Niedtner, Grace Proebsting, Griffin Bassman, Jack Gerrits, Jacob Alber, Peter Chang, Ricky Loynd, Robert West, Victor Dibia, Ahmed Awadallah, Ece Kamar, Rafah Hosn, and Saleema Amershi. Magentic-One: A Generalist Multi-Agent System for Solving Complex Tasks. *arXiv preprint arXiv:2411.04468*, 2024. URL https://arxiv.org/abs/2411.04468.

Xinzge Gao, Chuanrui Hu, Bin Chen, and Teng Li. Chain-of-Memory: Enhancing GUI Agents for Cross-Application Navigation. *arXiv preprint arXiv:2506.18158*, 2025. URL https://arxiv.org/abs/2506.18158.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/6507b115562bb0a305f1958ccc87355a-Abstract-Conference.html.

Mengkang Hu, Yuhang Zhou, Wendong Fan, Yuzhou Nie, Ziyu Ye, Bowei Xia, Tao Sun, Zhaoxuan Jin, Yingru Li, Zeyu Zhang, Yifeng Wang, Qianshuo Ye, Bernard Ghanem, Ping Luo, and Guohao Li. OWL: Optimized Workforce Learning for General Multi-Agent Assistance in Real-World Task Automation. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/48dcc43a534c5b582f9d0fdb778e9b84-Abstract-Conference.html.

Xuhao Hu, Xi Zhang, Haiyang Xu, Kyle Qiao, Jingyi Yang, Xuanjing Huang, Jing Shao, Ming Yan, and Jieping Ye. ToolCUA: Towards Optimal GUI-Tool Path Orchestration for Computer Use Agents. *arXiv preprint arXiv:2605.12481*, 2026. URL https://arxiv.org/abs/2605.12481.

Heyuan Huang, Yeyi Guan, Jihong Wang, Mingzhi Wang, Jiamu Zhou, Xiangmou Qu, Jiaxin Yin, Xin Liao, Xingyu Lou, and Jun Wang. TopoClaw: A Human-Centric and Topology-Aware Agent Operating System. *arXiv preprint arXiv:2605.15556*, 2026. URL https://arxiv.org/abs/2605.15556.

Jin Jia, Zhiling Deng, Zhuangbin Chen, Yingqi Wang, and Zibin Zheng. MAS-FIRE: Fault Injection and Reliability Evaluation for LLM-Based Multi-Agent Systems. *arXiv preprint arXiv:2602.19843*, 2026. URL https://arxiv.org/abs/2602.19843.

Ehud Karpas, Omri Abend, Yonatan Belinkov, Barak Lenz, Opher Lieber, Nir Ratner, Yoav Shoham, Hofit Bata, Yoav Levine, Kevin Leyton-Brown, Dor Muhlgay, Noam Rozen, Erez Schwartz, Gal Shachaf, Shai Shalev-Shwartz, Amnon Shashua, and Moshe Tennenholtz. MRKL Systems: A modular, neuro-symbolic architecture that combines large language models, external knowledge sources and discrete reasoning. *arXiv preprint arXiv:2205.00445*, 2022. URL https://arxiv.org/abs/2205.00445.

Kimi Team. Kimi K2.5: Visual Agentic Intelligence. *arXiv preprint arXiv:2602.02276*, 2026. URL https://arxiv.org/abs/2602.02276.

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 *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024. URL https://aclanthology.org/2024.acl-long.50/.

Huatao Li, Xinwei Geng, Yuheng Wang, Yutong Li, Runde Yang, Hantao Chen, Shu Yao, Jingru Fan, Xuhui Ren, Yuanyuan Zhao, Fei Huang, and Chen Qian. DevicesWorld: Benchmarking Cross-Device Agents in Heterogeneous Environments. *arXiv preprint arXiv:2607.13465*, 2026a. URL https://arxiv.org/abs/2607.13465.

Wanli Li, Bowen Zhou, Yunyao Yu, Zhou Xu, Yifan Yang, Dongsheng Li, and Caihua Shan. WeaveBench: A Long-Horizon, Real-World Benchmark for Computer-Use Agents with Hybrid Interfaces. *arXiv preprint arXiv:2606.09426*, 2026b. URL https://arxiv.org/abs/2606.09426.

Chang Ma, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. AgentBoard: An Analytical Evaluation Board of Multi-Turn LLM Agents. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/877b40688e330a0e2a3fc24084208dfa-Abstract-Datasets_and_Benchmarks_Track.html.

Mike Merrill, Alexander Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong (Ryan) Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Christopher Rytting, Ryan Marten, Yixin Wang, Jenia Jitsev, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/444a3737adaee10d86ad2ef5f74468e6-Abstract-Conference.html.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large Language Model Connected with Massive APIs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/e4c61f578ff07830f5c37378dd3ecb0d-Abstract-Conference.html.

Archiki Prasad, Alexander Koller, Mareike Hartmann, Peter Clark, Ashish Sabharwal, Mohit Bansal, and Tushar Khot. ADaPT: As-Needed Decomposition and Planning with Language Models. In *Findings of the Association for Computational Linguistics: NAACL*, 2024. URL https://aclanthology.org/2024.findings-naacl.264/.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative Agents for Software Development. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024. URL https://aclanthology.org/2024.acl-long.810/.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/28e50ee5b72e90b50e7196fde8ea260e-Abstract-Conference.html.

Christopher 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 (ICLR)*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/01a83bc2f2732a58e6aa731e659e7101-Abstract-Conference.html.

Yangjun Ruan, Honghua Dong, Andrew Wang, Silviu Pitis, Yongchao Zhou, Jimmy Ba, Yann Dubois, Chris Maddison, and Tatsunori Hashimoto. Identifying the Risks of LM Agents with an LM-Emulated Sandbox. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/7274ed909a312d4d869cc328ad1c5f04-Abstract-Conference.html.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language Models Can Teach Themselves to Use Tools. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. URL https://proceedings.neurips.cc/paper/2023/hash/d842425e4bf79ba039352da0f658a906-Abstract-Conference.html.

Linxin Song, Yutong Dai, Viraj Prabhu, Jieyu Zhang, Taiwei Shi, Li, Junnan Li, Silvio Savarese, Zeyuan Chen, Jieyu Zhao, Ran Xu, and Caiming Xiong. CoAct-1: Computer-Using Multi-Agent System with Coding Actions. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/ce65173b994cf7c925c71b482ee14a8d-Abstract-Conference.html.

Peter Steinberger and OpenClaw Contributors. OpenClaw, 2026. Open-source software. URL https://github.com/openclaw/openclaw.

Haotian Sun, Yuchen Zhuang, Lingkai Kong, Bo Dai, and Chao Zhang. AdaPlanner: Adaptive Planning from Feedback with Language Models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/b5c8c1c117618267944b2617add0a766-Abstract.html.

Eduardo Treviño, Hugo Contant, James Ngai, Graham Neubig, and Zora Zhiruo Wang. Benchmarking Failures in Tool-Augmented Language Models. In *Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL)*, 2025. URL https://aclanthology.org/2025.naacl-long.149/.

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 *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024. URL https://aclanthology.org/2024.acl-long.850/.

Peiyi Wang, Lei Li, Liang Chen, Zefan Cai, Dawei Zhu, Binghuai Lin, Yunbo Cao, Lingpeng Kong, Qi Liu, Tianyu Liu, and Zhifang Sui. Large Language Models Are Not Fair Evaluators. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024. URL https://aclanthology.org/2024.acl-long.511/.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversations. In *Conference on Language Modeling (COLM)*, 2024. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2023/08/LLM_agent.pdf.

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 (NeurIPS)*, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5d413e48f84dc61244b6be550f1cd8f5-Abstract-Datasets_and_Benchmarks_Track.html.

Frank (Fangzheng) Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, Mingyang Yang, Hao Yang Lu, Amaad Martin, Zhe Su, Leander Maben, Raj Mehta, Wayne Chi, Lawrence Jang, Yiqing Xie, Shuyan Zhou, and Graham Neubig. TheAgentCompany: Benchmarking LLM Agents on Consequential Real World Tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025a. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/0d744742f6fac4d1134c019b7cef3c8a-Abstract-Datasets_and_Benchmarks_Track.html.

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. *arXiv preprint arXiv:2602.16855*, 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*, 2025b. URL https://aclanthology.org/2025.findings-acl.1113/.

Yunhe Yan, Shihe Wang, Jiajun Du, Yexuan Yang, Yuxuan Shan, Qichen Qiu, Xianqing Jia, Xinge Wang, Xin Yuan, Xu Han, Mao Qin, Yinxiao Chen, Chen Peng, Shangguang Wang, and Mengwei Xu. MCPWorld: A Unified Benchmarking Testbed for API, GUI, and Hybrid Computer Use Agents. *arXiv preprint arXiv:2506.07672*, 2025. URL https://arxiv.org/abs/2506.07672.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations (ICLR)*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Chaoyun Zhang, Shilin He, Liqun Li, Si Qin, Yu Kang, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. API Agents vs. GUI Agents: Divergence and Convergence. In *ICML Workshop on Computer Use Agents (CUA)*, 2025a. URL https://openreview.net/pdf/ea6fc8767bab35e40f16578a6894d2e8fdcb26eb.pdf.

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 *Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL)*, 2025b. 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. UFO^3: Weaving the Digital Agent Galaxy. *arXiv preprint arXiv:2511.11332*, 2025c. URL https://arxiv.org/abs/2511.11332.

Chaoyun Zhang, He Huang, Chiming Ni, Jian Mu, Si Qin, Shilin He, Lu Wang, Fangkai Yang, Pu Zhao, Bo Qiao, Chao Du, Liqun Li, Yu Kang, Paul Jiang, Suzhen Zheng, Rujia Wang, Jiaxu Qian, Minghua Ma, Jian-Guang Lou, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. UFO2: The Desktop AgentOS. *Transactions on Machine Learning Research*, 2026. URL https://openreview.net/forum?id=iAuZVWCduc.

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 *CHI Conference on Human Factors in Computing Systems (CHI)*, 2025d. URL https://doi.org/10.1145/3706598.3713600.

Wentao Zhang, Liang Zeng, Yuzhen Xiao, Yongcong Li, Ce Cui, Yilei Zhao, Rui Hu, Yang Liu, Yahui Zhou, and Bo An. AgentOrchestra: Orchestrating Multi-Agent Intelligence with the Tool-Environment-Agent (TEA) Protocol. *arXiv preprint arXiv:2506.12508*, 2025e. URL https://arxiv.org/abs/2506.12508.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. URL https://papers.neurips.cc/paper_files/paper/2023/hash/91f18a1287b398d378ef22505bf41832-Abstract-Datasets_and_Benchmarks.html.

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 (ICLR)*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/4410c0711e9154a7a2d26f9b3816d1ef-Abstract-Conference.html.

Xiao Zhou, Siyue Zhang, Yilun Zhao, Jinbiao Wei, Tingyu Song, Arman Cohan, and Chen Zhao. GUI vs. CLI: Execution Bottlenecks in Screen-Only and Skill-Mediated Computer-Use Agents. *arXiv preprint arXiv:2606.24551*, 2026. URL https://arxiv.org/abs/2606.24551.

Dongsheng Zhu, Xuchen Ma, Yucheng Shen, Xiang Li, Yukun Zhao, Shuaiqiang Wang, Lingyong Yan, and Dawei Yin. When Tools Fail: Benchmarking Dynamic Replanning and Anomaly Recovery in LLM Agents. *arXiv preprint arXiv:2606.05806*, 2026. URL https://arxiv.org/abs/2606.05806.

A NOTATION

Tables 4 and 5 summarize the notation used in the method and evaluation, respectively. Latent execution states describe the environment, whereas orchestration and device-local states record information maintained by the agents.

Table 4: Notation for problem formulation and stateful control.

Symbol	Definition
Task, devices, and latent execution state	
I	Natural-language instruction to be completed.
$\mathcal{C}_I$	Task dependencies and device assignment constraints associated with I.
$\mathcal{D}$; d, d'	Set of available devices; the current device and an alternative device, respectively.
$\mathcal{M}_d$; m	Execution modalities exposed by device d; a selected modality, such as API, CLI, or GUI.
t	Time index for latent execution state; coordination-step index in orchestration equations.
x_d^t	Latent execution state of device d at time t, comprising device-level and modality-specific conditions.
$x_{d,\mathrm{dev}}^t$	Conditions shared across modalities on device d, such as connectivity.
$x_{d,m}^t$	Conditions specific to modality m on device d, such as permissions and authentication.
g	Unfinished subtask goal considered in the feasibility formulation.
$\mathcal{R}_d(g; x_d^t)$	Set of feasible executions that complete g on device d under its current state and applicable task constraints; an execution may use multiple modalities.
$\varnothing$	Empty set; $\mathcal{R}_d(g; x_d^t) = \varnothing$ means that the goal has no feasible execution on device d.
Orchestration state and inter-device control	
s_t^O	Orchestration state at coordination step t: $(P_t, \mathcal{Y}_t, \mathcal{F}_t)$.
P_t; P_0	Current dependency-ordered subtask chain $\langle q_1, \ldots, q_K \rangle$; its initial plan.
K; j	Number of subtasks in the current plan; subtask index in the method formulation.
q_j	Device-level subtask represented by (g_j, d_j, c_j, z_j).
g_j; d_j	Objective of subtask q_j; device assigned to execute it.
c_j; z_j	Execution context required by the subtask; execution status of q_j.
$\mathcal{Y}_t$	Completed subtask results maintained in orchestration state.
$\mathcal{F}_t$	Execution feedback and context available for assessing device feasibility and revising the remaining plan.
r_j	Outcome returned after local execution of q_j terminates, including status, available results, and execution context.
$\mathcal{U}_O$	Orchestration state update using the current state, dispatched subtask, and returned outcome.
Device-local control and modality execution	
ℓ	Device Agent decision-step index within an assigned subtask. One Executor invocation may contain multiple low-level actions.
$s_{j,\ell}^d$	Device-local state for q_j on device d at local step ℓ, recording progress, outputs, observations, and evidence about modality feasibility.
π_d	Decision process of the Device Agent on device d, selecting EXECUTE, DONE, or ESCALATE.
u	Concrete instruction issued to a selected Modality Executor.
EXECUTE(m, u)	Invoke modality m to carry out instruction u.
DONE(y_j)	Terminate local execution and return completed subtask result y_j.

Continued on the next page.

Table 4 continued.

Symbol	Definition
ESCALATE(e_j)	Terminate local execution and return execution evidence e_j for orchestration-level replanning.
$\mathcal{E}_{d,m}$	Modality Executor for modality m on device d; $\mathcal{E}_{d,m}(u)$ returns feedback (σ, y, o).
σ; y; o	Executor-level success status, resulting output, and execution observations, respectively.
$\mathcal{U}_D$	Device-local state update using the previous state, selected modality, issued instruction, and executor feedback.

Table 5: Notation and metric abbreviations used in evaluation.

Symbol / metric	Definition
CAPEBench quality and token efficiency	
N; i	Number of evaluated episodes; episode index. In the CRAB evaluation, these refer to evaluated tasks.
J_i; j	Number of gold steps in episode i; gold-step index in evaluation equations. Gold steps are distinct from system-generated subtasks q_j.
$a_{i,j} \in \{0, 1\}$	Whether gold step j in episode i has evidence-supported completion with the intended semantics on an allowed device.
A_i	Episode-level adherence score: $J_i^{-1} \sum_{j=1}^{J_i} a_{i,j}$.
$\mathbb{I}[\cdot]$	Indicator function, equal to one when the enclosed condition holds and zero otherwise.
Comp.	Task Completion: mean episode-level fraction of satisfied service and file-state postconditions, expressed as a percentage.
Adh.	Instruction Adherence: mean episode-level adherence, $100N^{-1} \sum_{i=1}^{N} A_i$.
PP	Perfect Pass: percentage of episodes with full adherence on every gold step, $100N^{-1} \sum_{i=1}^{N} \mathbb{I}[\forall j,\ a_{i,j} = 1]$.
T_i	Total input and output tokens used by the evaluated system's planning and execution components in episode i.
Tok./Ep.	Mean tokens per episode: $N^{-1} \sum_{i=1}^{N} T_i$, including unsuccessful episodes.
Tok./PP	Total tokens across all episodes divided by the number of perfect-pass episodes; equivalently, Tok./Ep./(PP/100). Reported as ∞ when there are no perfect passes.
Judge consistency	
MAE	Mean absolute difference between episode-level adherence scores from a judge and the reference judge, expressed in percentage points.
κ	Fleiss' kappa for agreement among judges on binary gold-step decisions.
Additional evaluation on CRAB	
$s_i \in \{0, 1\}$	Official evaluator's binary success decision for task i.
$c_i \in [0, 1]$	Official evaluator's completeness score for task i, including verified intermediate progress; distinct from subtask context c_j.
SR	Success Rate: percentage of tasks judged fully successful, $100N^{-1} \sum_{i=1}^{N} s_i$.
Mean Completeness	Mean official task completeness, $100N^{-1} \sum_{i=1}^{N} c_i$; distinct from CAPEBench's postcondition-based Completion.
$\mathcal{I}_{\text{tok}}$	Indices of CRAB tasks with positive token usage: $\{i : T_i > 0\}$.
Token CE	Token cost-effectiveness: mean per-task completeness per token, scaled by 100, $\frac{100}{\lvert\mathcal{I}_{\text{tok}}\rvert} \sum_{i \in \mathcal{I}_{\text{tok}}} c_i / T_i$.

B CAPEBENCH CONSTRUCTION

CAPEBench evaluates cross-device task execution under controlled changes in modality and device state. Starting from manually specified seed tasks, it generates unperturbed and perturbed instances with fixed task requirements and reproducible execution conditions. This section describes the execution environment, task specifications, perturbation mechanisms, and instance construction.

B.1 EXECUTION ENVIRONMENT AND AVAILABLE MODALITIES

CAPEBench runs on two Linux and two Android devices with real application services and local file systems. Each device exposes a subset of API, CLI, and GUI modalities, with only partial service API coverage, requiring agents to ground task steps through the available modalities. Linux devices support all three modalities, while Android devices support API and GUI execution.

The environment includes Gmail, Google Drive, GitHub, and Google Calendar, together with local file processing on Linux. Each device maintains its own authentication profile, application state, and local files. Tasks can therefore require both access to application content and execution on a particular device, such as creating a local report before using its contents in a subsequent service operation.

Table 6: Runtime versions used in formal evaluation.

Runtime	**System version**	**Display**
Linux	Ubuntu 22.04.5 LTS; kernel 6.8.0-111-generic; x86_64	X display `:1`; 1280×800
Android	Android 13; SDK 33; security patch 2024-03-01; model `sdk_gphone64_arm64`; ABI `arm64-v8a`	1080×2400; density 420

API execution remains subject to the assigned device's conditions. Structured service calls are mediated by a relay that applies the corresponding device's authentication state, network availability, and injected API restrictions. Operations involving local files also use the assigned device's file system. For example, uploading `~/Desktop/incident_brief.md` from a Linux device reads the file on that device. API access therefore does not bypass the device constraints imposed on GUI or CLI execution.

The standard API profile exposes selected service operations while leaving other operations to GUI or CLI execution. API-only baselines receive an expanded service-API profile, as detailed in Table 7. This expansion adds application operations but does not provide general shell execution or local filesystem manipulation. CLI execution supports these local operations on Linux, while GUI execution provides visual interaction with applications on both device types.

B.2 SEED TASKS AND GOLD-STEP SCHEMA

CAPEBench contains 23 manually written cross-device seed tasks. Each seed specifies a natural-language instruction and an ordered list of gold steps. The instructions combine service operations and local file processing, with intermediate outputs such as document contents, file paths, issue URLs, or event details needed by subsequent steps. Each seed involves at least two devices, includes at least two device transitions in its gold-step sequence, and contains at least one local CLI or filesystem step.

Each gold step records a step identifier, the intended device, the relevant application or local operation, and its semantic intent. These annotations specify what must be accomplished without prescribing the complete sequence of low-level actions. They also remain independent of the subtasks generated by an evaluated system: one generated subtask may cover multiple gold steps, and completing one gold step may involve multiple execution instructions.

Table 8 illustrates a seed task and its gold steps. The task requires agents to collect email evidence, process it on Linux, and carry the resulting information into GitHub, Calendar, and Gmail operations. The gold steps preserve the required device assignments and semantic dependencies while leaving concrete execution decisions to the agent.

Table 7: API tool profiles used by standard CAPEBench evaluation and API-only baselines.

Profile	Device	Service	Exposed API tools
limited	Linux	GitHub	github.list_issues, github.get_pull_request, github.list_pull_files
		Calendar	calendar.create_event, calendar.update_event, calendar.search_events, calendar.list_event_changes
		Drive	drive.upload_file, drive.list_folder, drive.get_permissions, drive.ensure_permission
limited	Android	Drive	drive.list_folder, drive.get_permissions, drive.ensure_permission
		Gmail	gmail.create_label, gmail.create_thread, gmail.search_threads, gmail.send
expanded	Linux	GitHub	github.upsert_issue, github.list_issues, github.create_comment, github.get_pull_request, github.list_pull_files
		Calendar	calendar.ensure_calendar, calendar.create_event, calendar.update_event, calendar.search_events, calendar.get_event, calendar.list_event_changes
		Drive	drive.create_folder, drive.upload_file, drive.upload_content, drive.list_folder, drive.read_text_file, drive.get_permissions, drive.ensure_permission
expanded	Android	Drive	drive.create_folder, drive.upload_file, drive.upload_content, drive.list_folder, drive.read_text_file, drive.get_permissions, drive.ensure_permission
		Gmail	gmail.create_label, gmail.create_thread, gmail.search_threads, gmail.get_thread, gmail.get_attachment, gmail.send

Table 8: Representative CAPEBench seed task and gold steps.

Field	Content
Seed instruction	Use android_device_1 to read the Gmail alert thread (label mdc_alerts, subject starts with [MDC-ALERT]) and collect the attachment; use linux_device_1 to upload the evidence file to Google Drive folder MDC Ops Workspace/gateway_a_incident_evidence and run CLI commands to generate ~/Desktop/incident_brief.md from the alert thread and attachment; use linux_device_2 to create an issue in example/mdc using the contents of ~/Desktop/incident_brief.md plus Google Drive evidence links, then create the Calendar triage event in calendar MDC Ops on May 8, 2026 from 10:00 to 10:30 (Asia/Shanghai) using the created issue URL; use android_device_2 to send the final acknowledgement email to ops-recipient@example.com with issue link and event details.
Gold steps	
Step	**Intended device and semantic intent**
S1	android_device_1; Gmail; read the MDC alert thread and collect the attachment evidence file.
S2	linux_device_1; Drive; upload the collected alert evidence file to the MDC Ops Workspace/gateway_a_incident_evidence Drive folder.
S3	linux_device_1; CLI; generate ~/Desktop/incident_brief.md from the alert thread and attachment content.
S4	linux_device_2; GitHub; create the tracking issue with ~/Desktop/incident_brief.md content and Google Drive evidence links.
S5	linux_device_2; Calendar; create the triage event in MDC Ops and include the issue URL.
S6	android_device_2; Gmail; send final acknowledgement email with issue link and event details.

The gold-step annotations are retained by the evaluation harness and are not provided to the evaluated agent. They identify the task requirements affected by each perturbation and support instruction-adherence evaluation and recovery analysis. The associated environment specification defines the initial service resources, authentication conditions, and local artifacts needed to instantiate the task.

B.3 Perturbation Taxonomy and Injection

Perturbations are classified by the scope of the execution alternatives they preserve. A modality-level perturbation restricts an execution modality while retaining an alternative on the assigned device. A device-level perturbation makes the affected task requirement infeasible on the assigned device while preserving a same-type peer that can support continued execution. This classification concerns task-relevant feasibility: a device-level perturbation need not disable every function of the device.

Table 9 summarizes the perturbation families and their persistence. API and CLI perturbations affect particular service calls or command invocations. Session perturbations remove access to an affected service on the assigned device, while network perturbations prevent network-dependent execution on that device.

Table 9: Perturbation families and injection points in CAPEBench. Counts refer to annotated injection points before mixed-instance generation.

Scope	Target	Fault subtypes	Persistence	Count
Modality-level	API	Rate limit, service busy, token expired, insufficient scope, permission denied, API disabled, unsupported operation	One matching call for rate limit and service busy; persistent otherwise	25
Modality-level	CLI	Tool unavailable, command not found, permission denied	Persistent	17
Device-level	Session	Invalid service session on the assigned device	Persistent	15
Device-level	Network	Network unavailable on the assigned device	Persistent	21

Each injection point specifies the target device and service or operation, the affected gold steps, the modality conditions, and the perturbation's persistence. The injection compiler translates this specification into environment controls and relay rules. Session perturbations change authentication profiles and enforce corresponding API restrictions. Network perturbations alter device connectivity and apply corresponding restrictions to relay-mediated API access. These controls prevent structured service calls from bypassing the perturbed device state.

API and CLI perturbations are enforced when a matching service call or command is issued. Persistent perturbations continue to affect matching operations throughout the episode. Rate-limit and service-busy perturbations affect only the first matching call, allowing execution to resume through a subsequent retry as well as through an alternative modality. Thus, modality-level instances test adaptation within the assigned device without requiring a modality change after every failure.

The current taxonomy does not include a standalone GUI-unavailable perturbation. GUI access is affected by session and network perturbations when the corresponding service becomes inaccessible on the assigned device. These conditions also restrict the relevant structured service access and are therefore classified as device-level perturbations.

Perturbations are installed as part of instance preparation, but their task-specific effects are encountered through execution. Agents are not given the selected perturbation specification or its expected recovery scope in advance.

B.4 Instance Generation and Benchmark Statistics

Instances are generated deterministically from the seed specifications. Each seed produces one unperturbed instance and one single-perturbation instance for every annotated injection point. Mixed instances combine one modality-level and one device-level injection point from the same seed. Can-

didate pairs are filtered using explicit incompatibility annotations in the seed specification; instance generation applies these annotations rather than inferring compatibility from system outcomes.

All instances derived from a seed retain its task instruction and gold-step requirements. The selected perturbations change execution conditions, while the evaluation protocol determines the allowed devices for affected steps, as described in Appendix D.2. This permits comparisons across runtime conditions without replacing the underlying task objective.

Table 10: CAPEBench instance construction and the execution behavior examined under each condition.

Condition	Construction	Execution behavior examined	Count
Unperturbed	One instance per seed, without injected perturbations	Modality selection and cross-device execution under nominal conditions	23
Modality-level	One modality-level injection point	Adaptation within the assigned device, including intra-device switching	42
Device-level	One device-level injection point	Inter-device switching and continued execution on a same-type peer	36
Mixed	One compatible modality-level and device-level pair	Adaptation involving both intra-device and inter-device switching	57
Total			**158**

Before evaluation, the pipeline renders the seed and environment templates into run-specific specifications, resolving resource identifiers and time-dependent fields. It then generates instances from these rendered specifications. The task instruction is supplied to the agent, while gold steps, selected injection points, and checking metadata remain with the evaluation harness.

The resulting benchmark contains 158 instances from 23 seeds and 78 injection points. The seeds contain 124 gold steps, with five to seven steps per seed. Table 11 reports the task statistics and application distribution. Gold-step counts are computed over the seed tasks, before expansion into perturbed instances.

Table 11: Seed-task statistics and gold-step distribution in CAPEBench.

Item	Value
Seed tasks	23
Gold steps	124
Average gold steps per seed	5.39
Gold steps per seed	5–7
Minimum devices per seed	2
Minimum device transitions per seed	2
Gmail steps	38
Drive steps	19
GitHub steps	24
Calendar steps	20
Local CLI/filesystem steps	23

Analysis sample. Main comparisons use Full-158. Architectural ablations, model-backbone comparisons, and oracle-planned execution use a fixed sample of 82 CAPEBench instances selected independently of system outcomes. It includes the unperturbed instance for each of the 23 seeds and up to one instance per seed for each perturbed condition: 22 modality-level, 19 device-level, and 18 mixed instances.

Table 12: Model identifiers and client inference settings for backbone comparisons and GUI execution. "Default" denotes the serving provider's default configuration; "N/A" denotes not applicable.

Model	API model ID	Temp.	Top-p	Thinking
DeepSeek-V4-Pro	`deepseek-v4-pro`	0.0	1.0	Off
Gemini-3-Flash	`gemini-3-flash-preview`	0.0	1.0	Default
GPT-5.4	`gpt-5.4-2026-03-05`	0.0	1.0	Off
Qwen3.8-Max	`qwen3.8-max`	0.0	1.0	Off
Kimi K2.5	`kimi-k2.5`	1.0	0.95	On
GPT-4o	`gpt-4o-2024-11-20`	Default	Default	N/A

C Implementation Details and Compared Systems

This section describes model configurations, OctoNest's implementation, baseline adaptations to CAPEBench, and architectural ablations. Available execution modalities and API tool profiles are detailed in Appendix B.

C.1 Model Versions and Inference Settings

Table 12 lists the API model identifiers and client-side inference settings used by each configuration. Parameters marked as "Default" were omitted from the requests and left to the serving endpoint. We use the DeepSeek-V4-Pro preview release of April 24, 2026,[2] and the original Qwen3.8-Max release of August 2, 2026.

C.2 OctoNest Implementation

Orchestration and plan updates. The Orchestrator receives the user instruction and device profiles describing device identifiers, operating systems, available capabilities, and controllable applications. It constructs an ordered plan whose subtasks specify an execution objective and an assigned device. Subtasks are dispatched sequentially, with each Device Agent receiving the assigned instruction together with any execution context accumulated for that subtask.

The runtime maintains subtask status, returned results, and the remaining plan. After successful execution, the Orchestrator identifies which remaining subtasks need information from the returned result and appends that information to their contexts. Such information can include retrieved content, resource identifiers, URLs, and local artifact locations. This propagation is not restricted to the immediately following subtask: a result may supply information needed by several later subtasks.

When device-local execution escalates a failure, the Orchestrator receives the recorded Device Agent decision rounds, including their inputs, decisions, and execution observations. The replanning input also includes the original user instruction, device profiles, completed-subtask results, and the remaining plan. Based on this information, the Orchestrator can revise execution instructions, insert recovery subtasks, or change device assignments. Replanning replaces the remaining portion of the plan while retaining earlier execution records.

The Orchestrator also assesses whether an unsuccessful subtask produced reusable partial progress and can carry that information into subsequent assignments. Execution ends when the current plan is exhausted, the Orchestrator declares that it cannot continue, or an execution budget is reached. Runtime completion decisions are distinct from benchmark success, which is determined by the independent evaluation protocol.

Device-local control and escalation. For each assigned subtask, the Device Agent receives the instruction, propagated context, device capabilities, and available modalities. It maintains a local interaction history containing its decisions and the feedback returned by Modality Executors. Each decision selects one of three actions: EXECUTE, which specifies a modality and a self-contained

[2] `https://huggingface.co/deepseek-ai/DeepSeek-V4-Pro`

instruction; DONE, which returns the local result; or ESCALATE, which transfers recovery responsibility to the Orchestrator.

Executor success does not necessarily complete the assigned subtask. The Device Agent can use the returned information to issue another instruction, either through the same modality or through a different one. Similarly, a failed invocation can lead to a revised instruction, an alternative modality, or escalation. This allows local execution to combine modalities across finer steps rather than committing the entire subtask to a single initial choice.

The default local execution chain permits up to four Executor invocations, with a separate budget of three failed-attempt or instruction-correction events. Successful invocations count toward the chain limit without consuming this failure budget. Once the chain limit is reached, the Device Agent must conclude with DONE or ESCALATE; exhaustion of the failure budget also ends local execution. These limits apply to Executor invocations, each of which may itself contain multiple low-level actions.

Modality Executors. Each Executor receives a concrete instruction and returns execution status, output, and observations to the Device Agent. Executors perform modality-specific interaction; subsequent device-local decomposition and modality selection remain with the Device Agent.

The **API Executor** invokes structured service functions through the relay under the active API profile. It operates on both Linux and Android devices and uses the authentication, connectivity, and perturbation conditions associated with the assigned device. Its default interaction budget is 12 steps per invocation.

The **CLI Executor** performs Linux-local shell and filesystem operations, including file reading and writing, search, patching, and scripting. Its default interaction budget is also 12 steps per invocation.

The **GUI Executor** interacts with applications through screenshots and device actions on Linux and Android. It builds on an OSWorld-style GUI-agent implementation using Kimi K2.5 (Xie et al., 2024; Kimi Team, 2026). Its memory combines recent screenshot–action history, longer textual interaction history, and selected earlier screenshots with descriptions (Gao et al., 2025). Android actions are translated into operations such as tapping, swiping, text entry, and navigation; Linux actions operate the assigned desktop environment. The default GUI interaction budget is 30 steps per invocation.

C.3 BASELINE ADAPTATION

All compared systems use the same CAPEBench instances, device environments, perturbation mechanisms, 30-minute episode timeout, and evaluation pipeline. Adapters connect each system to the benchmark devices and collect execution records for the common checkers. GUI and hybrid configurations use the standard API profile where applicable, while API-only configurations receive the expanded profile described in Table 7.

CRAB. We use CRAB's single-agent configuration (Xu et al., 2025b) and extend its device-addressed action space from one Linux and one Android device to CAPEBench's two devices of each type. CRAB-GUI retains the single-agent interaction loop and directs GUI actions to the named benchmark device. Its actions include desktop and Android interaction primitives, together with waiting and task completion. CRAB-API uses the same control organization with structured service API tools. We do not construct a CRAB-Hybrid variant, as jointly integrating API, CLI, and GUI execution would require substantial changes to its original execution architecture beyond the adaptation considered here.

CRAB-GUI uses GPT-4o. In pilot runs with Kimi K2.5, the model frequently referenced visual labels outside the GroundingDINO/EasyOCR annotations, preventing reliable progress through CRAB's action interface. CRAB-API uses DeepSeek-V4-Pro.

UFO^3. The UFO^3 adaptation uses task-graph creation, dependency-based scheduling, and graph updates driven by execution results (Zhang et al., 2025c). To support the CAPEBench environment, graph nodes are connected to the corresponding API or GUI Executors used by OctoNest.

UFO^3-Hybrid extends this setup to API, CLI, and GUI execution. Each available device–modality pair is exposed as a separate execution target to the task-graph planner. Selecting a target therefore determines both the physical device and the Executor modality. The Executor returns its result to the graph-level controller, which can update subsequent assignments. This configuration provides access to all supported modalities without adding OctoNest's iterative device-local control process. Text planning and API/CLI execution use DeepSeek-V4-Pro, while GUI execution uses Kimi K2.5.

TopoClaw. The TopoClaw adaptation supplies the original task instruction to its primary agent and connects its tools to the CAPEBench environment (Huang et al., 2026). We preserve its native agent loop and orchestration mechanisms, including task-graph execution when initiated by the system.

TopoClaw accesses service APIs and Linux CLI/GUI operations through a benchmark gateway. Android GUI execution uses its native MobileGUIAgent, with a bridge translating screenshot and action requests to the assigned Android device. API and CLI operations pass through the benchmark's perturbation controls, while GUI execution observes the same prepared device state as the other systems. The primary agent and native auxiliary agents use Kimi K2.5. This configuration is reported as TopoClaw-Hybrid. Since TopoClaw natively supports hybrid execution and requires only limited extensions to provide modality coverage comparable to OctoNest, we evaluate this configuration without introducing API-only or GUI-only variants.

C.4 Architectural Ablations

All architectural ablations use the same fixed 82-instance sample and evaluation protocol. Each intervention changes the specified control mechanism while retaining the remaining components where applicable.

No Device Agent. We remove device-local planning and expose each available device–modality Executor directly to the Orchestrator. The Orchestrator chooses both the device and modality and supplies the execution instruction. Each assignment invokes the selected Executor once; failure returns directly to the Orchestrator without an intervening device-local revision.

One-shot Device Agent. We retain the Device Agent for a single initial decision using the assigned instruction, device capabilities, and available modalities. It selects one modality and issues an instruction intended to complete the assigned subtask in one Executor invocation. The Executor retains its normal multi-step interaction loop, but its subsequent observations cannot trigger another Device Agent decision. An unsuccessful invocation is escalated directly to the Orchestrator.

Unified Device Agent. We retain the Orchestrator but replace the Device Agent and separate Modality Executors with one model session per assigned subtask. This session directly selects and executes API, CLI, and GUI operations on Linux, or API and GUI operations on Android. Deterministic tool adapters perform the requested operations without additional Executor-model calls. The unified session uses Kimi K2.5 to process both textual tool feedback and screenshots,[3] while the Orchestrator continues to use DeepSeek-V4-Pro. This configuration combines device-local coordination and concrete execution within a single agent.

Reactive Orchestrator. We remove explicit planning of the remaining task. At each orchestration decision, the controller receives the original instruction, device capabilities, and accumulated execution history, then selects one next subtask or terminates. Device-local refinement, modality-specific execution, and detailed failure feedback remain enabled.

Fixed orchestration plan. We retain initial plan generation and device-local control but disable orchestration-level replanning. Successful results can still be propagated to subsequent subtasks. If a Device Agent escalates a failure, however, the system terminates without revising the remaining plan or assigning recovery work to another device.

[3] A single model handles all execution modalities in this configuration and must therefore support both textual tool use and screenshot-based GUI interaction.

Failure-only device feedback. We retain the planning and execution hierarchy but replace the detailed evidence supplied after device failure with a binary failure signal. Device-local decision history, execution observations, and diagnostic error details are withheld from the replanning input. The Orchestrator still receives the original request, device profiles, completed results, and remaining task descriptions, and must decide how to continue without the detailed evidence explaining the failed execution.

D Evaluation Protocol

We evaluate task outcomes, instruction adherence, token usage, and execution behavior using a common protocol. Main system comparisons use Full-158, while selected analyses use the fixed 82-instance sample described in Appendix B.

D.1 Episode Execution and Evaluation

Each episode follows a prepare–execute–check–cleanup pipeline. Preparation instantiates the task resources, configures device and service state, and installs the selected perturbations. The evaluated system then receives the rendered task instruction and interacts with the available devices. After execution, the checkers inspect the resulting environment and execution records. Cleanup removes episode-specific resources and restores the controlled device state.

The device names in each task instruction specify the intended assignments under ordinary execution. The prompts given to all compared systems explicitly state that unfinished work may be moved to another available device when the assigned device cannot complete it. Device capabilities and available actions guide the choice of replacement; for the evaluated device-level perturbations, the corresponding same-type peer is the intended capable alternative. The adherence checker applies this peer-device rule to affected gold steps, while unaffected steps retain their original device constraints. The allowed-device labels and gold-step annotations remain internal to evaluation.

Table 13: Formal evaluation pipeline and artifact boundaries.

Stage	Pipeline action	Artifact or boundary
Prepare	Render run-local seed and environment specifications, expand instances, prepare device and service state, install selected perturbations, and run preflight checks.	Instances are generated from run-local rendered files; selected fault metadata stays in the harness.
Execute	Dispatch the rendered natural-language instruction to the evaluated system and allow interaction through its permitted execution modalities.	The system sees the task instruction and execution modalities; logs, subtask traces, and token journals are written during execution in real time.
Check	Run the common result checker, adherence checker, and token aggregation logic.	Completion, adherence, and token metrics are computed from the shared episode record and observable final state. All other metrics are derived from these three values.
Cleanup	Remove relay rules, restore authentication and network state, clean local temporary artifacts, and reset service-side test resources.	Each episode starts from a controlled state; reported results use artifacts produced by this pipeline.

All systems receive a 30-minute execution budget per episode. Unsuccessful and timed-out executions remain in the evaluation set and receive credit for requirements supported by the available evidence. A timeout does not automatically erase verified partial progress. Infrastructure-invalid attempts are handled separately and rerun; induced benchmark failures are valid execution outcomes and are not excluded.

Each reported instance contributes one selected execution record per system. When an invalid attempt is replaced, its replacement is used instead of counting both attempts. Task quality and token usage are computed from the same selected episode record.

D.2 Quality and Efficiency Metrics

Task Completion. Task Completion measures the fraction of required service and file-state postconditions satisfied in the final environment. These checks verify outcomes such as created issues, uploaded files, calendar events, sent messages, and local artifacts. We first compute the fraction of satisfied postconditions within each episode and then average these episode scores. Consequently, episodes with more postconditions do not receive greater weight.

Instruction Adherence. Instruction Adherence evaluates execution against the fixed gold steps rather than against system-generated subtasks. A generated subtask can cover several gold steps, and a gold step can draw evidence from several execution traces. The checker therefore aligns traces with the gold-step specification and scores each gold step independently.

For each step, the checker determines whether its completion is supported by evidence, whether the executed behavior satisfies its semantic intent, and whether completion occurred on an allowed device. Direct environment checks take precedence when available. Otherwise, matched execution records or downstream results that specifically depend on the step can provide completion evidence. A system's own task-status label is not sufficient: a failed subtask may contain a completed gold step, while a reported successful subtask may lack evidence for one.

Let $a_{i,j} \in \{0, 1\}$ indicate whether gold step j in episode i satisfies all three requirements. For an episode containing J_i gold steps, its adherence score is

$$A_i = \frac{1}{J_i} \sum_{j=1}^{J_i} a_{i,j}$$

Failed, skipped, and unverified steps remain in the denominator. The reported Adherence is the mean episode score, expressed as a percentage.

Table 14: Gold-step adherence scoring fields.

Field	Evidence	Scoring rule
Gold-step completion	Direct result-check evidence when available; otherwise matched traces and downstream dependent traces.	Pass only when the gold-step result is proven. Trace self-reports do not override failed direct checks.
Intent match	Gold-step intent, matched trace instruction, matched trace result, and observed text.	Pass only when the executed behavior satisfies the gold step's semantic intent.
Device match	Expected device, allowed-device set, and devices observed in matched traces.	Pass only when a trace providing completion evidence executes on an allowed device.

The allowed device is determined from the instance specification. Unperturbed steps and steps affected only by modality-level perturbations retain their original device constraint. Steps designated for device-level recovery require the corresponding same-type peer. For device-wide network perturbations, this rule extends to gold steps assigned to the affected device. Mixed instances apply the device-level rule wherever it is applicable; other steps retain their original assignment.

Device matching is tied to completion evidence. An unsuccessful attempt on an allowed device cannot be combined with successful execution on a disallowed device to obtain adherence credit. Gold-step annotations and evaluation-side device constraints are not supplied to the agent.

Perfect Pass. Perfect Pass measures the fraction of episodes in which every gold step receives full adherence:

$$\mathrm{PP} = \frac{100}{N} \sum_{i=1}^{N} \mathbb{I}[\forall j,\ a_{i,j} = 1]$$

where N is the number of evaluated episodes. It therefore captures fully adherent execution, whereas Completion and Adherence retain credit for partial progress.

Table 15: Model identifiers and request settings used by the judge-consistency runner. "Default" denotes the serving provider's default configuration.

Judge	API model ID	Temp.	Top-p	Thinking
DeepSeek-V4-Pro	`deepseek-v4-pro`	0.0	1.0	Default
Qwen3.7-Plus	`qwen3.7-plus`	0.0	1.0	Default
GPT-5.6-Terra	`gpt-5.6-terra`	Default	Default	Low
Gemini-3.7-Flash	`gemini-3.7-flash`	Default	Default	Default

Token efficiency. Let T_i denote the total input and output tokens used by the evaluated system's planning and execution components in episode i. We report

$$\text{Tok./Ep.} = \frac{\sum_{i=1}^{N} T_i}{N}, \qquad \text{Tok./PP} = \frac{\sum_{i=1}^{N} T_i}{\sum_{i=1}^{N} \mathbb{I}[\forall j,\ a_{i,j} = 1]}$$

The latter is equivalent to Tok./Ep. divided by the perfect-pass rate expressed as a fraction. It includes tokens consumed by unsuccessful episodes and is reported as ∞ when no episode achieves Perfect Pass. Token counts measure model usage rather than monetary cost.

D.3 Statistical Analysis

Aggregate quality scores are episode-weighted means. Runtime-condition results use the same definitions restricted to the corresponding instances. We compute comparisons from the underlying stored scores before rounding the displayed values.

For the main comparison between OctoNest and UFO^3-Hybrid, we use paired bootstrap resampling at the seed-task level. Each resample draws 23 seeds with replacement and includes all instances belonging to each selected seed. The same sampled seeds and instance multiplicities are used for both systems, preserving pairing and dependence among instances derived from the same seed.

We generate 100,000 resamples with a fixed random seed and report the 2.5th and 97.5th percentiles as pointwise 95% confidence intervals. Quality comparisons use percentage-point differences. For efficiency, each resample recomputes Tok./Ep. and Tok./PP for both systems before calculating the relative percentage change. No resamples have zero perfect-pass counts in this comparison.

These intervals characterize variation under resampling of the observed seed tasks. They do not estimate repeated-execution variability or provide simultaneous coverage across all reported metrics.

D.4 Adherence Checker Consistency

Adherence scoring requires semantic alignment between gold steps and execution traces and may therefore depend on the evaluator model (Zheng et al., 2023; Wang et al., 2024). We assess cross-judge consistency on the same 44 CAPEBench instances for OctoNest, UFO^3-GUI, UFO^3-Hybrid, TopoClaw-Hybrid, and CRAB-GUI. The sample contains seven unperturbed, 12 modality-level, eight device-level, and 17 mixed instances, comprising 242 gold steps per system.

We compare DeepSeek-V4-Pro, the reference judge used for the reported benchmark scores, with Qwen3.7-Plus, GPT-5.6-Terra, and Gemini-3.7-Flash. Table 15 lists their API model identifiers and inference settings. For each system–instance pair, the judges evaluate the same fixed execution record and gold-step specification under the same scoring rules. This analysis reuses existing executions rather than generating new agent trajectories.

Table 16 reports mean episode-level adherence, gold-step agreement with DeepSeek, and episode-level mean absolute error (MAE). Step agreement is the percentage of binary gold-step decisions matching the reference judge. MAE averages the absolute difference between episode adherence scores and is expressed in percentage points.

All held-out judges reproduce the OctoNest gold-step decisions made by DeepSeek. Across the two UFO^3 configurations and TopoClaw-Hybrid, agreement ranges from 93.80% to 99.17%, with episode-level MAE at most 4.70 points. Agreement is lower for CRAB-GUI. Although UFO^3-GUI

Table 16: Adherence checker consistency across judge models. Mean adherence and step agreement are percentages; MAE is in percentage points.

System	Judge	Mean adh.	Step agree.	Ep. MAE
OctoNest	DeepSeek-V4-Pro	78.89	Reference	Reference
	Qwen3.7-Plus	78.89	100.00	0.00
	GPT-5.6-Terra	78.89	100.00	0.00
	Gemini-3.7-Flash	78.89	100.00	0.00
UFO^3-GUI	DeepSeek-V4-Pro	47.36	Reference	Reference
	Qwen3.7-Plus	44.40	95.04	4.62
	GPT-5.6-Terra	46.60	98.35	1.52
	Gemini-3.7-Flash	46.98	98.76	1.14
UFO^3-Hybrid	DeepSeek-V4-Pro	60.55	Reference	Reference
	Qwen3.7-Plus	60.42	98.35	1.54
	GPT-5.6-Terra	59.59	98.35	1.61
	Gemini-3.7-Flash	60.42	99.17	0.78
TopoClaw-Hybrid	DeepSeek-V4-Pro	48.01	Reference	Reference
	Qwen3.7-Plus	46.11	97.11	2.80
	GPT-5.6-Terra	45.13	93.80	4.70
	Gemini-3.7-Flash	46.11	95.45	4.47
CRAB-GUI	DeepSeek-V4-Pro	15.88	Reference	Reference
	Qwen3.7-Plus	7.39	91.32	8.48
	GPT-5.6-Terra	4.49	88.43	11.39
	Gemini-3.7-Flash	10.48	90.50	9.23

Table 17: Human verification on 36 sampled pairs of system and instance comprising 196 gold steps. All values are percentages.

System	Judge Adh.	Human Adh.	Step agree.
OctoNest	86.35	87.94	97.96
UFO^3-Hybrid	64.60	60.16	91.84
TopoClaw-Hybrid	49.31	36.72	83.67
CRAB-GUI	11.11	12.33	89.80
Overall	52.84	49.29	90.82

and TopoClaw-Hybrid exchange order across judges, every judge ranks OctoNest first and UFO^3-Hybrid second.

Across the 1,210 system-specific gold-step items, the four judges achieve Fleiss' $\kappa = 0.930$. This indicates high cross-judge consistency under the shared scoring protocol; it does not constitute an independent human validation of scoring accuracy.

Human verification. We conduct human verification on 36 sampled pairs of system and instance, covering nine task variants across four systems and 196 gold-step decisions, sampled from the existing judge-consistency subset. An experienced computer science student reviews execution evidence and assigns each step label.

Table 17 shows high agreement for OctoNest and the primary baseline, UFO^3-Hybrid, at 97.96% and 91.84%, respectively. Overall step agreement reaches 90.82% (Cohen's $\kappa = 0.816$), with the system ranking preserved within the reviewed sample. Disagreements mainly concern evidence of content reading and representation-sensitive checkpoint matching. These results support the stability of the sampled comparison between OctoNest and the evaluated baselines.

E Additional Experimental Results

We further examine whether OctoNest's performance advantage persists across text-model backbones and on an external cross-device benchmark, and evaluate oracle-planned execution to assess the role of planning and recovery decisions and the adequacy of the execution budget.

E.1 Evaluation across Model Backbones

We compare OctoNest with UFO^3-Hybrid on the 82-instance sample using DeepSeek-V4-Pro, Gemini-3-Flash, GPT-5.4, and Qwen3.8-Max. Within each comparison, both systems use the same text backbone for planning and API/CLI execution, while Kimi K2.5 remains fixed for GUI execution. The task subset, available modalities, and evaluation protocol remain unchanged.

Table 18: Results across model backbones on the 82-instance sample, with Kimi K2.5 fixed for GUI execution. Bold indicates the better result for each backbone.

Model backbone	Framework	Comp. ↑	Adh. ↑	PP ↑	Tok./Ep. ↓	Tok./PP ↓
DeepSeek-V4-Pro	UFO^3-Hybrid	67.24	67.67	26.83	**667,210**	2,486,873
	OctoNest	**75.55**	**77.49**	**43.90**	716,035	**1,630,969**
Gemini-3-Flash	UFO^3-Hybrid	64.82	67.66	21.95	892,575	4,066,173
	OctoNest	**72.17**	**76.23**	**41.46**	**668,161**	**1,611,447**
GPT-5.4	UFO^3-Hybrid	29.67	30.33	18.29	**310,157**	**1,695,525**
	OctoNest	**76.79**	**74.78**	**39.02**	832,696	2,133,783
Qwen3.8-Max	UFO^3-Hybrid	66.95	68.14	21.95	**557,648**	2,540,396
	OctoNest	**76.50**	**75.09**	**31.71**	596,351	**1,880,799**

Table 18 shows that OctoNest achieves higher Completion, Adherence, and Perfect Pass than UFO^3-Hybrid under all four backbones. Because the GUI model and available execution modalities are held fixed, these comparisons examine how each framework operates with different text models for planning and API/CLI execution. The consistent advantage suggests that OctoNest's benefits are not specific to DeepSeek-V4-Pro. Across the tested model families, its separation of cross-device orchestration, device-local refinement, and modality-specific execution supports more effective use of the available capabilities.

The variation across backbones further distinguishes average execution quality from complete task success. OctoNest maintains relatively similar Completion and Adherence scores, whereas UFO^3-Hybrid exhibits a much larger decline with GPT-5.4. However, OctoNest's Perfect Pass still varies across models. GPT-5.4, for example, achieves OctoNest's highest Completion but a lower Perfect Pass than DeepSeek-V4-Pro and Gemini-3-Flash. Satisfying more final-state requirements on average therefore does not necessarily produce more fully adherent episodes: residual errors in individual required steps can still prevent complete success. The framework's quality advantage persists, but it does not eliminate the influence of the backbone on end-to-end reliability.

The efficiency results reveal a separate interaction between execution quality and token expenditure. With DeepSeek-V4-Pro and Qwen3.8-Max, OctoNest uses more tokens per episode, but the higher Perfect Pass rate compensates for this increase and lowers Tok./PP. Gemini-3-Flash improves both sides of the trade-off, achieving higher quality with fewer tokens per episode. GPT-5.4 behaves differently: OctoNest substantially improves execution quality, yet the additional token expenditure exceeds the gain in successful episodes, making Tok./PP 25.8% higher. Thus, the backbone with lower token usage per attempt need not provide better execution quality, and a framework's quality advantage need not imply lower cost per success. Across these comparisons, OctoNest consistently improves quality, while its efficiency depends on the balance between additional execution effort and the resulting success rate.

E.2 Evaluation on CRAB Benchmark

Task selection and execution setup. As of September 24, 2026, we identified no external cross-device benchmark other than CRAB with publicly available task definitions, environment setup

instructions, and evaluation code.[4] We therefore evaluate OctoNest and the baseline systems on the public CRAB Benchmark-v0 release v0.1.2 (Xu et al., 2025b) to assess whether the observed benefits extend beyond CAPEBench.[5] We include all 10 JSON tasks in the release's `cross` directory and all five hand-authored tasks involving both Ubuntu and Android. This gives a fixed evaluation set of 15 tasks spanning one Linux device and one Android device.

The task inventory in the pinned release differs from the collection described in the CRAB paper. The released definitions contain 75 JSON tasks and 11 hand-authored tasks, while the paper reports 120 tasks, including 18 cross-platform tasks. We therefore base task selection on the executable cross-device definitions in the pinned release and compare all systems on that same set, rather than directly comparing against the original paper's aggregate results.

The selected tasks involve information retrieval, cross-device content transfer, and document or application operations. Examples include transferring Linux text into Android Keep Notes, combining Android Maps lookup with Ubuntu document creation, and sending information from Ubuntu Slack through Android Messages. These tasks require intermediate information to be carried across applications and devices while preserving the requested content and destination. No additional CAPEBench perturbations are injected.

We compare OctoNest with CRAB, UFO^3, and TopoClaw. Each task–system pair contributes one valid execution under a 30-minute budget. Infrastructure-invalid attempts are replaced, while timed-out executions remain in the evaluation. OctoNest and UFO^3 use Linux CLI/GUI and Android GUI execution, without service API access. They use DeepSeek-V4-Pro for text planning and CLI execution and Kimi K2.5 for GUI execution. CRAB uses GPT-4o, while TopoClaw uses Kimi K2.5.

Evaluation metrics. We use CRAB's official `GraphEvaluator`, which represents task progress through intermediate checkpoints and their dependencies. Success Rate (SR) measures the fraction of tasks judged fully successful. Mean Completeness averages the official per-task completeness values, retaining credit for verified intermediate progress on unsuccessful tasks. This completeness metric is distinct from CAPEBench's service- and file-postcondition Completion metric.

For N tasks, let $s_i \in \{0, 1\}$ denote the evaluator's success decision and $c_i \in [0, 1]$ its completeness score. We compute

$$\mathrm{SR} = \frac{100}{N} \sum_{i=1}^{N} s_i, \qquad \text{Mean Completeness} = \frac{100}{N} \sum_{i=1}^{N} c_i. \tag{1}$$

Each task receives equal weight, so tasks with larger evaluator graphs do not contribute disproportionately. Both successful and unsuccessful executions are included.

We additionally report token cost-effectiveness (Token CE):

$$\text{Token CE} = \frac{100}{|\mathcal{I}_{\mathrm{tok}}|} \sum_{i \in \mathcal{I}_{\mathrm{tok}}} \frac{c_i}{T_i}, \tag{2}$$

where T_i is the token count and $\mathcal{I}_{\mathrm{tok}} = \{i : T_i > 0\}$. This metric averages per-task completeness per token and gives credit for partial progress. It is distinct from CAPEBench's Tok./PP, which measures aggregate token expenditure per fully adherent episode.

Results and analysis. Table 19 shows that OctoNest achieves the highest observed Success Rate, Mean Completeness, and Token CE among the compared systems. It completes three tasks, compared with two for UFO^3, and attains a Mean Completeness of 40.02%. The agreement between the success and completeness results indicates that its advantage appears both in fully completed tasks and in average evaluator-verified progress.

[4] At the time of checking, the official repositories of DevicesWorld (Li et al., 2026a) and JarvisGUI (Chen et al., 2026b) indicated that their benchmark resources were forthcoming. NebulaBench provides task descriptions in the UFO^3 paper (Zhang et al., 2025c), but we could not locate a public benchmark package containing the corresponding environment preparation and evaluation procedures.

[5] Official repository: `https://github.com/camel-ai/crab/tree/v0.1.2`; pinned commit: `762e0b392a78657c3c477aa6736460a37459d544`.

Table 19: Additional evaluation on 15 tasks from the official CRAB Benchmark-v0 release v0.1.2. SR and mean completeness are percentages. Token CE is the mean per-task completeness-to-token ratio over all tasks with token usage, multiplied by 100. Bold indicates the largest reported value.

Method	Successful tasks	SR ↑	Mean completeness ↑	Token CE ↑
CRAB	0/15	0.00	27.25	3.47×10^{-4}
TopoClaw	0/15	0.00	20.37	8.71×10^{-5}
UFO^3	2/15	13.33	32.68	5.31×10^{-4}
OctoNest	**3/15**	**20.00**	**40.02**	$\mathbf{1.38 \times 10^{-3}}$

Completeness also reveals differences hidden by binary success. CRAB and TopoClaw complete no task end to end, but their nonzero completeness scores reflect successful intermediate operations. The additional progress achieved by OctoNest therefore matters even when execution does not satisfy every requirement of a task.

Task-level results illustrate both the gains and their boundaries. OctoNest and UFO^3 both complete the postal-code lookup and Linux-text-to-Android-Keep tasks. OctoNest additionally completes the task that reads the first incomplete item in Android Tasks and follows its instruction. Among unsuccessful executions, OctoNest makes more verified progress on the map-to-university and photo-transfer tasks, whereas UFO^3 progresses further on the Slack-to-Messages task.

These results provide complementary evidence on an independently defined task set using the benchmark's own evaluator. The observed advantage is therefore not confined to CAPEBench's injected perturbations or scoring rules, although the limited task set does not establish broad cross-benchmark generalization.

E.3 Execution with Oracle Planning

To examine execution performance when device assignments, modality choices, and recovery decisions are provided in advance, we evaluate an oracle-planned configuration on the 82-instance sample. Using task specifications and perturbation metadata, the oracle predefines task stages, their dependencies, and the device and modality used at each stage, including feedback-conditioned intra-device and inter-device switching. The plan remains fixed during execution. Modality Executors still perform the actual API, CLI, and GUI operations, using DeepSeek-V4-Pro for API and CLI execution and Kimi K2.5 for GUI execution. Stage completion and failure detection rely on actual execution feedback. This configuration supplies privileged planning information while retaining the need to carry out concrete actions in the environment.

Table 20 shows that perturbed instances approach unperturbed performance under oracle planning, with Perfect Pass differing by only 1.33 percentage points. Completion and Adherence likewise remain close across the two groups. This suggests that much of the recovery challenge can be addressed by providing suitable modality choices and device assignments. The comparison with online systems further clarifies this pattern. Without perturbations, OctoNest matches the oracle's Perfect Pass rate and closely approaches its Completion and Adherence. Under perturbations, OctoNest improves Perfect Pass over UFO^3-Hybrid from 25.42% to 40.68%, while oracle planning reaches 50.85%. These results support the value of nested control while identifying further opportunities to improve execution-conditioned planning, particularly under device-level perturbations, where the gap between OctoNest and the oracle is largest.

Oracle-planned execution consumes substantially fewer tokens, and this advantage is not explained solely by the absence of online planning-model calls. On this sample, OctoNest's Orchestrator and Device Agents together account for 14.97% of its total token usage. As a post-hoc accounting comparison, we add their token usage on each matched instance to the oracle's execution cost while retaining the oracle's outcomes. Even after this adjustment, the oracle requires 46.3% fewer tokens per episode and 54.0% fewer tokens per perfect pass than OctoNest. The cost advantage therefore extends beyond direct planning overhead to the execution induced by those decisions, supporting the potential of better planning and recovery decisions to reduce downstream execution expenditure.

Table 20: Comparison with oracle-planned execution on the 82-instance sample. Quality metrics are percentages; time is the mean execution-stage duration in minutes. Perturbed pools modality-level, device-level, and mixed instances. Token counts include all runtime model calls; oracle planning requires no online planning-model calls.

Condition	System	Comp. ↑	Adh. ↑	PP ↑	Tok./Ep.	Tok./PP	Time (min)
Unperturbed	UFO^3-Hybrid	73.19	70.97	30.43	653,186	2,146,184	14.21
	OctoNest	81.81	86.05	52.17	600,638	1,151,222	11.31
	Oracle planning	82.32	87.20	52.17	248,538	476,365	5.51
Modality-level	UFO^3-Hybrid	82.88	77.23	36.36	659,140	1,812,636	14.88
	OctoNest	82.58	82.60	45.45	768,847	1,691,464	15.16
	Oracle planning	80.45	84.50	50.00	280,134	560,269	6.40
Device-level	UFO^3-Hybrid	53.95	60.38	26.32	691,714	2,628,514	10.70
	OctoNest	64.21	71.35	42.11	869,347	2,064,699	16.40
	Oracle planning	87.19	90.38	63.16	280,517	444,152	5.96
Mixed	UFO^3-Hybrid	54.54	59.47	11.11	669,126	6,022,133	13.31
	OctoNest	70.93	66.80	33.33	637,110	1,911,331	12.67
	Oracle planning	69.26	73.55	38.89	306,323	787,689	6.48
Perturbed	UFO^3-Hybrid	64.92	66.38	25.42	672,677	2,645,862	13.06
	OctoNest	73.11	74.16	40.68	761,021	1,870,842	14.80
	Oracle planning	79.21	83.05	50.85	288,247	566,887	6.28
Overall	UFO^3-Hybrid	67.24	67.67	26.83	667,210	2,486,873	13.38
	OctoNest	75.55	77.49	43.90	716,035	1,630,969	13.82
	Oracle planning	80.08	84.22	51.22	277,109	541,023	6.07

The experiment also supports the choice of a 30-minute execution budget. All 82 oracle-guided execution attempts terminated within this limit, averaging 6.07 minutes, with 95.12% terminating within 10 minutes. The budget therefore accommodates oracle-guided execution across the evaluated instances, including the additional operations required for recovery.

Taken together, these results reveal substantial room to improve online coordination with the current executor and model configuration. OctoNest's token cost per perfect pass is approximately 3.01 times that of oracle-planned execution, and the gap remains substantial after accounting for online planning costs. Alongside the quality gains, this suggests that better device assignments, modality choices, and recovery decisions can improve both task success and the efficiency of downstream execution.

Oracle planning nevertheless leaves concrete execution challenges unresolved. Of the 82 episodes, 40 do not achieve Perfect Pass, with 66 unmet final-state postconditions. Audited failures include omitted source information, incomplete generated artifacts, GUI operations reported as successful without the required changes taking effect, and failed stages that prevent dependent operations from being reached. These errors motivate improvements in information extraction, artifact validation, and verification of action outcomes. CAPEBench therefore offers room for progress both in coordinating existing execution capabilities and in strengthening those capabilities themselves.

F Case Studies: Recovery Behavior and Remaining Limitations

We follow four execution traces to illustrate how observations lead to local execution changes or revisions of the remaining plan. The first two traces concern the same approval task under different perturbations. The third follows both forms of switching within one episode, while the fourth shows how requirements can be lost during successive recovery attempts.

F.1 SUCCESSFUL INTRA-DEVICE SWITCHING

The user asks the agent to inspect an approval review event on Linux device 1, prepare an approval note on Linux device 2, update a GitHub issue, upload the note to Drive, and send an email from Android device 1.

1. **Attempt the Calendar inspection.** The Device Agent on Linux device 1 selects API execution to retrieve the approval event from the `MDC Ops` calendar. The API Executor attempts the search but returns:

   > HTTP 400 Bad Request: this calendar operation is not supported by the API endpoint.

   The required event details have not yet been obtained.
2. **Change the modality while retaining the device.** The feedback identifies an unsupported API operation. The Device Agent therefore selects GUI execution on Linux device 1 and instructs the GUI Executor to open Google Calendar, locate the approval review event, and return its details.
3. **Recover the source information.** The GUI Executor retrieves the event titled `MDC Change Approval Review`, including its scheduled time, description, and change identifier. The Calendar inspection is now complete without changing its device assignment.
4. **Continue with the recovered information.** Linux device 2 uses the event details to generate the approval note, update the GitHub issue, and upload the file. Android device 1 then sends the notification. All five gold steps and all final-state checks pass.

The decisive observation is the API-specific error: it leads to a different execution modality on the same device, preserving the source information needed by subsequent steps.

F.2 SUCCESSFUL INTER-DEVICE SWITCHING

Now consider the same approval task when Linux device 1 has a device-level network failure.

1. **Encounter a different failure during the same operation.** The Device Agent again begins with the Calendar API. This time, the returned observation states:

   > calendar is unreachable because the device network is unavailable.

   No approval event details have been retrieved.
2. **Escalate the device-level blocker.** The Device Agent determines that changing modalities would not restore access over the unavailable connection. It reports the failure to the Orchestrator without attempting local GUI or CLI recovery.
3. **Revise the assignment and its downstream dependency.** The Orchestrator assigns the Calendar inspection to Linux device 2. It also updates the subsequent file-generation instruction to use the event details returned by this recovery operation.
4. **Resume execution on the peer device.** The destination Device Agent selects API execution and retrieves the approval event successfully. The note, GitHub update, Drive upload, and final email then complete. All five gold steps and all final-state checks pass.

Although the initial operation is identical to the preceding case, the observed network failure calls for inter-device switching. The revised plan also connects the recovered information to the remaining work.

F.3 COMBINING INTRA-DEVICE AND INTER-DEVICE SWITCHING

The user asks the agent to read a configuration-drift alert, inspect the associated Calendar event, generate and upload a report, create a GitHub issue, and send a final notification. In this instance, Android device 1 has no network access, while the Gmail API is disabled on Android device 2.

1. **Discover that the initial device cannot read the alert.** The GUI Executor on Android device 1 encounters the message "Connect to the internet for more results." An API attempt

also reports network unavailability. The Device Agent escalates because neither attempt retrieves the alert.

2. **Move the reading operation to another device.** The Orchestrator reassigns the operation to Android device 2. Its GUI Executor opens the alert and returns the complete message, including the affected configuration identifiers. The Linux devices then inspect the change event, generate the report, upload it, and create the tracking issue.
3. **Encounter a modality-specific failure later in the task.** When Android device 2 attempts to send the final notification, its Gmail API returns HTTP 403, reporting that the API is not enabled. This failure occurs during notification delivery, after the earlier alert-reading operation has already succeeded.
4. **Complete the notification locally.** The Device Agent switches from API to GUI execution on Android device 2. The GUI Executor sends the email with the Drive and GitHub links and confirms it in the Sent folder. All six gold steps and all final-state checks pass.

The episode first requires inter-device switching to obtain the source information, then intra-device switching to complete a later operation. A replacement device can therefore restore progress while still requiring local adaptation to its available modalities.

F.4 Task-Requirement Drift During Replanning

The user asks the agent to read Calendar and Gmail information, create `change_summary.md`, update a GitHub issue with its contents, and send a notification containing the issue link.

1. **Collect the information, then encounter an execution blocker.** The Calendar and Gmail reading steps succeed. During file generation on Linux device 2, however, the injected device fault causes the CLI attempts to return errors. The Device Agent reports that it cannot complete the operation locally.
2. **Reassign execution, but change the required output.** The Orchestrator transfers the operation to Linux device 1. Its revised instruction now states:

   > Do not write to a file; return the complete markdown text directly.

   Execution can proceed, but the revised task no longer requires creating the original file.
3. **Discover that the returned result is insufficient.** The executor reports that the Markdown has been generated, but returns only a description of the result. The next Device Agent cannot obtain the actual Markdown body and therefore cannot update the GitHub issue.
4. **Repair the missing content, but omit an unfinished operation.** The Orchestrator introduces a step to regenerate the full Markdown. The revised remaining plan then proceeds directly to sending the email, omitting the GitHub update. Its email instruction still requires the issue link from that unsuccessful operation.
5. **Encounter the unresolved dependency.** Regeneration returns the full text, but the email step fails because the issue link is unavailable. The Orchestrator subsequently restores the GitHub update to the plan, yet execution reaches the time limit before the remaining operations complete. The final checks confirm neither the required file, the issue update, nor the notification. Completion is 0%, and Adherence is 40%, with only the two initial reading steps satisfied.

This sequence illustrates a remaining limitation of iterative replanning. Each revision responds to an immediate blocker, but the revisions do not consistently preserve the original output requirements and unfinished dependencies. Maintaining execution state supports further recovery attempts; it does not by itself ensure that the evolving plan continues to fulfill the complete task.

G Limitations

Method. OctoNest maintains execution state and revises decisions as feedback becomes available, but successive revisions do not guarantee that every original task requirement and dependency is preserved. As illustrated in Appendix F, recovery can change required outputs or omit unfinished

operations, leaving the overall task incomplete despite progress on individual subtasks. Feasibility judgments also depend on the completeness and accuracy of execution feedback: ambiguous failures or insufficient result content can lead to inappropriate continuation, reassignment, or termination. The current design focuses on adaptation within an episode and does not explicitly address memory reuse across episodes. Such reuse requires deciding what to retain and when past observations remain reliable: an expired login session may later be restored by the user, whereas an expired API token may remain unresolved. This creates an exploration and exploitation trade-off between revalidating previously unavailable modalities and relying on alternatives that succeeded before. Furthermore, the current implementation uses a sequential task chain to simplify planning and recovery. This limits its ability to exploit independent operations across devices; parallel scheduling and concurrent state updates under a DAG-based plan remain outside the present evaluation.

Benchmark. CAPEBench uses real application services and introduces a broad range of execution perturbations to approximate the difficulties agents may encounter in practical multi-device tasks. Nevertheless, the evaluated services cover only a subset of real-world applications, and the constructed perturbations cannot encompass the full diversity and complexity of deployment conditions. Extending this coverage requires additional environment integration and systematic modeling of how failures arise and interact in practice. Such efforts lie beyond the scope of the present diagnostic benchmark, and we leave broader evaluation in more varied real-world environments to future work.

Evaluation. Cross-system comparisons reflect the evaluated implementations, adaptations, and model configurations. Although the benchmark protocol is standardized, differences in controller models and native execution mechanisms prevent complete isolation of architectural effects from implementation and model effects. Architectural ablations and trace analyses provide complementary evidence but do not eliminate all such dependencies.

H DISCUSSION

Multi-device execution requires agents to act on feasibility that is only partially known at planning time. As permissions, authentication, and connectivity become apparent through interaction, a failed operation may require either intra-device switching to another modality or inter-device switching to another device. Responding at the wrong level can lead to repeated failures, unnecessary reassignment, or premature termination. Our results support organizing execution around this distinction: device-local state informs whether execution can continue on the current device, while orchestration state guides revisions to device assignments and the remaining plan. OctoNest thus provides a way to connect interaction-revealed state with adaptation at the appropriate control level, addressing a central challenge as computer use agents expand from single-device operation to cross-device tasks.

Several limitations remain. Although OctoNest maintains execution state, agents may still overlook an original output requirement or an unfinished step when revising the plan. Successful local recovery therefore does not always ensure completion of the full task. Incomplete or ambiguous execution feedback can also make it difficult to determine whether a failure requires another modality on the current device or reassignment to another device. Future work could investigate explicit tracking of outstanding requirements and targeted verification when feedback is insufficient, helping agents preserve task requirements during recovery and make better-informed switching decisions.

CAPEBench uses real application services and diverse execution perturbations to approximate practical multi-device challenges. Nevertheless, the services and runtime conditions covered remain a subset of those encountered in real-world environments. Extending this coverage involves integrating additional services and reproducing more varied execution conditions, which we leave for future benchmark development.

Finally, the present design focuses on adaptation within an episode. Reusing experience across episodes requires deciding what to retain and when previous observations remain reliable. An expired login session may later be restored by the user, whereas an expired API token may remain unresolved. This creates an exploration and exploitation trade-off between revalidating previously unavailable modalities and relying on alternatives that succeeded before. Future work could investigate memory mechanisms that retain the context and timing of observations and refresh or

invalidate them as conditions change, allowing accumulated experience to guide execution without treating historical feasibility as current fact.

I DISCLOSURE OF AI ASSISTANCE

We used generative AI tools to improve the clarity, grammar, conciseness, and presentation of author-written text, and to assist with literature retrieval and discovery, including identifying potentially relevant prior work. All retrieved references were independently verified against the original sources by the authors. We reviewed all AI-assisted edits and take full responsibility for the final content of this work.