OS-NATIVE AGENTS: MANAGING USER TASKS IN THE DYNAMIC MOBILE WORLD

Anonymous authors
Paper under double-blind review

ABSTRACT

Mobile agents can perform an increasingly broad range of tasks on behalf of users. Yet the mobile world continues to change after an agent has acted: application and system states evolve, new information arrives, and users revise their plans. In conventional mobile interaction, the operating system (OS) delivers relevant changes and governs access to device resources, while users decide which ongoing tasks require attention and how to respond across applications. Delegating this coordination to an agent requires OS support across interactions, including access to relevant state changes, cross-application capabilities, and execution under user authorization. We therefore introduce OS-native agents, which are recognized by the operating system and integrated into its native state, capability, and trust mechanisms. The OS provides system-level interfaces through which the agent observes runtime application and system state, discovers and composes structured capabilities exposed by applications and system services, and executes actions across apps and services through an OS contract for identity, authorization, and audit. We realize OS-native agents on Android and introduce DynamicWorld, a replayable benchmark for managing multiple ongoing tasks in a shared Android environment, with controlled changes in applications, system state, and user requests.

1 INTRODUCTION

The mobile world continues to change after an agent has acted. A flight delay can disrupt a booked trip, a device change can invalidate a purchase choice, and users can revise their plans. Recent mobile agents can execute increasingly complex instructions (Rawles et al., 2025; Li et al., 2026; Cai et al., 2026). Maintaining their results as circumstances change requires ongoing coordination across applications and interactions.

Pursuing user goals over time requires task coordination across interactions. Users conventionally perform that coordination: the OS delivers application and system updates, and users determine which tasks need attention and what should happen next. An agent managing those tasks must recognize relevant changes, reconsider existing arrangements, and respond within the user's authorization. Users can then specify goals and constraints without having to interpret every update and issue another instruction.

Transferring task coordination to agents requires OS support beyond individual user interactions. Relevant changes must reach the agent while current application and system state remains available for subsequent decisions. Follow-up actions require capabilities across applications, governed by identity and authorization at execution time. The OS already mediates application activity, service access, and permissions across the device, providing a foundation for agents to carry out these responsibilities.

We introduce *OS-native agents*, which are recognized by the operating system and integrated into its native state, capability, and trust mechanisms. The OS provides interfaces for observing runtime state, discovering and composing structured capabilities, and executing actions under identity, authorization, and audit controls. We realize the framework on Android, combining native OS mechanisms with an agent harness that retains context and resumes reasoning when relevant events or user inputs arrive.

To evaluate how agents manage tasks as the mobile world changes, we introduce *DynamicWorld*. Dynamic and asynchronous benchmarks study responses to evolving environments (Froger et al.,

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

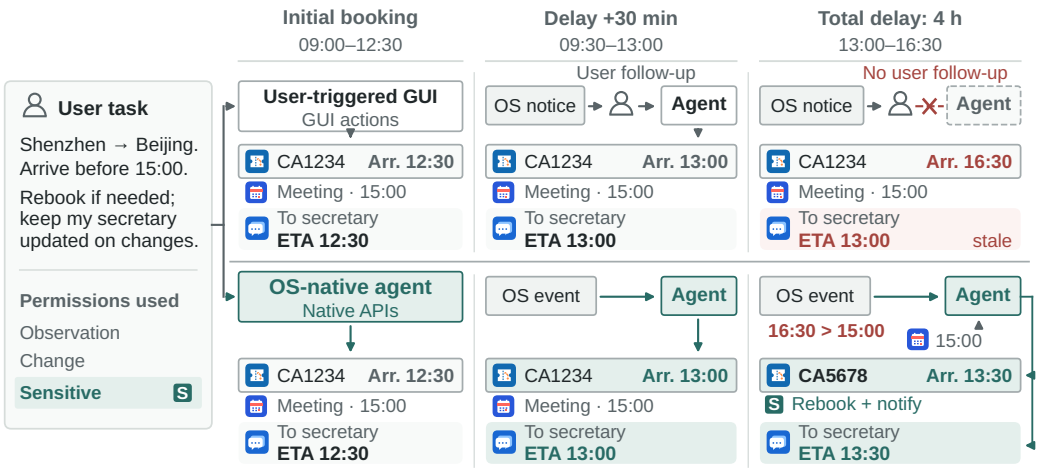

Figure 1: **Native OS support for managing user goals.** The upper workflow resumes on user follow-up; the lower workflow links the delegation to native events, state queries, and authorized actions. A user requests arrival before 15:00, authorizes rebooking if needed, and asks for updates to their secretary. A delay makes the booked flight miss the deadline. Native monitoring and state queries support rebooking and updating the secretary within the user's delegation. The S marker links the sensitive permission to the illustrated action.

2026; Nathani et al., 2026). DynamicWorld evaluates multiple ongoing tasks in a shared Android environment, with replayable changes in applications, system state, and user requests. It checks whether agents produce the required results on time and respect restrictions on their actions. An unauthorized share or an intervention after cancellation can violate a user goal even when the final state appears correct.

Our contributions are:

- We introduce OS-native agents, a framework that integrates agent activity with native state, capability, and trust mechanisms. Our Android realization combines these mechanisms with a persistent harness to resume ongoing tasks as the mobile world changes and execute cross-application actions under user authorization.
- We introduce DynamicWorld, a replayable benchmark for managing multiple ongoing tasks in a shared Android environment, with 88 tasks across 16 scenarios. Controlled application, system, and user changes exercise coordination over time.
- We evaluate complete agent systems with GPT-5.4 and GPT-5.4-mini. With GPT-5.4, our Android realization achieves 76.1% task success, compared with 40.9% for MobileClaw and 23.9% for MobileWorld, while using 58.0% fewer model calls than MobileClaw. Our analyses distinguish initial delivery from subsequent fulfillment and examine the model judgments required for continued task management under the same native support.

2 From User-Managed to Agent-Managed Tasks

Consider a user who asks an agent to book a flight arriving before 15:00, rebook if necessary, and keep their secretary informed (Figure 1). A later delay makes the booked flight miss the arrival deadline. The agent must receive the update, check current flight and calendar state, and revise the booking and message within the user's authorization. The goal persists across model turns; the relevant state and operations remain distributed across the phone.

OS integration connects the agent to the services that maintain this information and govern its use. Native event services retain and deliver changes while reasoning is paused; application and system queries supply current facts when it resumes. Applications describe their exposed operations through metadata, allowing the agent to discover the provider that owns each affected record. Actions enter

Table 1: **Agent integration and evaluation scope.** The comparison separates how agents access and act on their environment from the settings and outcomes used to evaluate them.

System / Benchmark	Agent–Environment Integration			Evaluation Scope	
	State Access	Capability	Trust Controls	Setting	Verification
AndroidWorld / M3A (Rawles et al., 2025)	Android GUI controller	GUI actions	—	Android task execution	System state
PhoneWorld (Tang et al., 2026)	Android GUI controller	GUI actions	—	Static mock-app tasks	Final answer / DB state
PhoneHarness / Bench (Li et al., 2026)	Device harness / host services	GUI / CLI / structured tools	Trace logging / safety checks	Mixed Android workflows	Side effects / execution traces
PalmClaw (Cai et al., 2026)	On-device Android application	Registered device tools	Android permissions / confirmation / tool boundaries	Mobile / assistant task execution	Task success / boundary cases
Gaia2 / ARE (Froger et al., 2026; 2025)	Asynchronous simulation runtime	Environment tools	—	Dynamic simulated environments	Action-level verification
PARE / Pare-Bench (Nathani et al., 2026)	Active-user simulation runtime	Environment tools	User confirmation	Active-user simulation	Proactive assistance outcomes
AOHP (Zhao et al., 2026)	AOSP framework / OS services	Service discovery / composition	Information-flow controls	Android service composition	Task completion / security policies
OS-native agent / DynamicWorld	Android services / persistent harness	App / system capability discovery	Permissions / confirmation / audit	Ongoing tasks / shared phone state	Required outcomes / authorization windows

execution paths with caller identities and applicable authorization checks, including when later events prompt another operation. These responsibilities motivate native state, capability, and trust integration (Section 3.2). The harness carries the user's goal across interactions and connects these services to model reasoning.

Existing work provides on-device tools, mixed-action harnesses, and OS service composition, alongside benchmarks for dynamic and proactive assistance (Cai et al., 2026; Li et al., 2026; Zhao et al., 2026; Froger et al., 2026; Nathani et al., 2026). Table 1 compares them along the state, capability, and trust dimensions of Section 3.2, together with their evaluation settings. Our framework connects these forms of OS support to persistent agent context and selective resumption (Section 3). DynamicWorld evaluates the resulting task management through required outcomes and action constraints as the phone changes (Section 4).

3 OS-Native Agents

3.1 Overview

An *OS-native agent* is an agent recognized by the operating system and integrated into its native state, capability, and trust mechanisms. The OS provides system-level interfaces through which the agent observes runtime application and system state, discovers and composes structured capabilities exposed by applications and system services, and executes actions across apps and services through an OS contract for identity, authorization, and audit.

Our OS-native agent integrates native state, capabilities, and execution controls with a persistent harness (Figure 2). The harness connects these interfaces to model reasoning, task context, memory, and skills across interactions.

3.2 Native OS Integration

State: observing the mobile world. OS and application interfaces expose both structured state and unstructured content. Structured observations include device time, battery and network status, the foreground application, and queried records such as orders and calendar entries. Notification text, on-screen content, and screenshots provide language and visual observations. A notification combines both forms, carrying its source application and timestamp alongside a message. System state arrives as snapshots and incremental updates; application details and GUI observations are retrieved on demand within available access permissions.

The agent relates these observations to user goals. A message asking to move a pickup to the east entrance half an hour later, for example, requires interpreting the text against the existing arrangement.

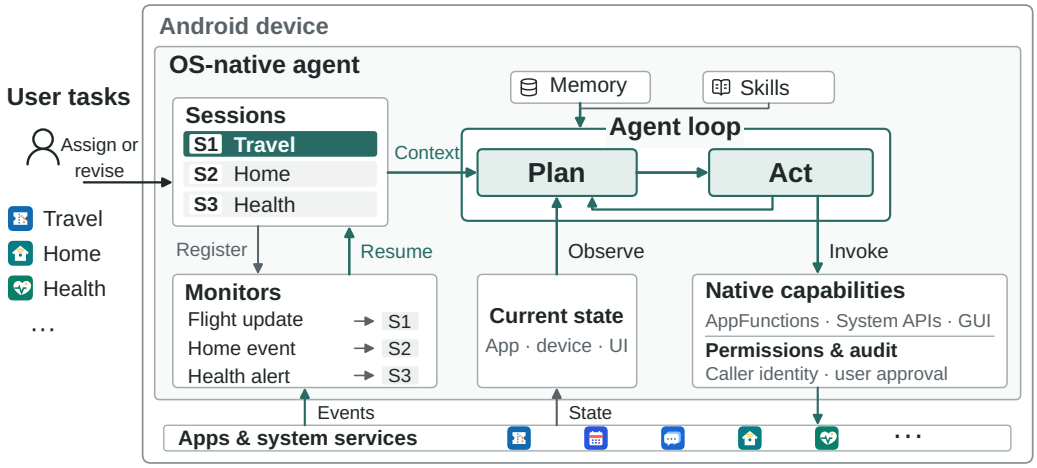

Figure 2: **OS-native agent framework.** Native state, capabilities, and execution controls connect the agent to applications and system services. The harness combines task and session context, memory, and skills with reasoning and execution feedback. Incoming events are assessed against ongoing tasks before execution resumes. Task and event labels are illustrative.

The agent can query current calendar and travel records to determine the necessary changes. Native interfaces supply observations with source context; model reasoning connects their meaning to the delegated task.

Between model turns, native services maintain monitors registered with an event source, filter, and lifetime. Matching notifications, system events, and deadlines provide change cues for delivery. Current-state queries then supply the facts needed to decide how to respond.

Capability: discovering and composing operations. A capability catalog describes available system operations and application functions through their providers, arguments, return formats, and availability. It updates when applications or their metadata change. The runtime exposes these descriptions as model-callable tools through capability search and progressive disclosure. Loading a skill supplies workflow instructions and exposes its declared application functions that are present in the catalog.

The agent can compose application functions, system operations, intents, and GUI actions within one workflow (Li et al., 2026; Zhao et al., 2026). Applications own the records affected by these operations. Returned object references let subsequent queries and actions address the same order or calendar entry, linking observations to concrete changes across apps.

Trust: acting under user control. The OS and application providers check interface access under the caller's execution identity. Supported sensitive operations obtain confirmation through trusted user interfaces. For supported profile fields, the model supplies a field reference and the device fills the destination locally after confirmation, keeping the value out of model context. The agent requests user authorization through explicit question and confirmation interfaces when a proposed action exceeds the delegation. Execution results and confirmation decisions are recorded for review.

3.3 OS-Native Agent Harness

The harness combines native state, capabilities, and execution controls in a persistent model–tool loop. Task context connects successive observations and actions to the same user delegation.

Grounding tasks in application state. Cross-application work requires a stable link between user goals and the records they concern. The agent queries current application state and retains object references, user constraints, and progress in task records. Session history and memory support continuation, while fresh queries establish the current facts. New requests and revisions are routed using this task context.

Responding to changes. An event warrants action when it matters to an ongoing delegation. Native monitors maintain observation between model turns; the harness associates incoming events with tasks and suppresses already handled or obsolete inputs. For relevant notifications, model assessment combines their content with task context to decide whether to retain an observation or resume execution. A flight delay can thus prompt a review of the existing pickup arrangement.

Executing under user control. Resumed work must reflect both current facts and the user's authority. The agent queries affected records and composes available application functions, system operations, and GUI actions. Execution paths apply access checks and require user confirmation for supported sensitive operations. Questions about intent or further authorization return through the user interaction channel. Tool results and subsequent state queries let the agent verify effects before updating task progress.

Maintaining and closing delegations. Waiting preserves responsibility, while revision and cancellation change what work remains. The agent saves progress and registers the subscriptions and monitors needed for continuation. Later inputs restore the relevant context and guide updates to application records. On completion or cancellation, the agent performs required application operations and closes the task; the runtime removes monitors no open task still needs. Appendix C details context retention and event handling.

3.4 Android Realization

We implement the OS-native agent on Android API 37 using native Android components and an extended DeepSeek Harness (DeepSeek AI, 2026). Native notification listeners and alarms support monitoring, content providers support queries, and intents and Accessibility support actions and GUI observations. Local applications expose Android AppFunctions whose metadata supplies the application capability catalog. These functions operate on the same records as the GUI (Section 4.2). Skills adapt public workflow and interface definitions to these functions.

Native integration resides in Android services and application interfaces. The harness coordinates reasoning and execution through these interfaces, using remote model inference. We extend DeepSeek Harness with task records, Android state and capability integration, and event-based continuation. An authenticated connection carries observations and tool calls between the Android components and the harness. Native tools use the agent application's identity; cross-application operations use AppFunctions, with runtime invocation checks and application validation. Appendix C describes available tools and execution controls.

DynamicWorld evaluates whether this system maintains multiple delegations correctly as shared application state and user instructions change.

4 DynamicWorld Benchmark

DynamicWorld evaluates how complete agent systems maintain multiple user goals as a shared phone changes. Tasks combine initial execution with later decisions about revision, notification, authorization, and closure.

4.1 Design Principles

Three principles connect the environment to ongoing task management. Shared application state links goals through common orders, messages, calendars, and resources. Earlier actions therefore shape later choices. Replayable changes specify event timing and dependencies, so each system encounters the same evolving situation. Temporal outcome evaluation verifies results during the periods in which each requirement applies, including revisions and cancellation, building on state-based and temporal verification (Rawles et al., 2025; Trivedi et al., 2024; Lu et al., 2024; Froger et al., 2026).

4.2 Environment-Grounded Scenario Construction

We begin with author-curated examples of everyday mobile activities. Recurring interactions guide the selection of applications and operations to implement. We group related tasks, interactions, and notifications by the activity in which they occur to form scenario outlines.

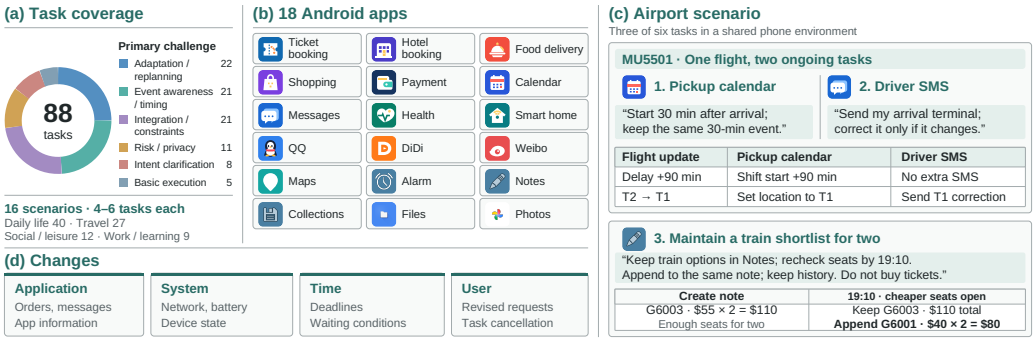

Figure 3: **DynamicWorld coverage and shared task context.** (a) The 88 tasks span four domains and 16 scenarios, with four to six related tasks per scenario. (b) Eighteen Android applications provide public observations and business operations. (c) Three tasks from one airport scenario illustrate user requests and expected outcomes. Tasks 1 and 2 share a flight: delays change the pickup calendar, while terminal changes also require a driver message. Task 3 maintains a train-options note for two: a scheduled seat recheck adds a cheaper eligible option to the same note, preserving the earlier option without purchasing tickets. (d) Changes arise from applications, system state, time, and user instructions.

The resulting environment contains 18 Android applications. Sixteen have local implementations, including four adapted from PhoneWorld (Tang et al., 2026); Files and Photos use Android system-image applications. Their GUIs and public interfaces operate on persistent application records. Product rules connect prices, availability, orders, payments, and notifications.

The authors manually write individual tasks specifying user goals, constraints, and required outcomes. Large language models assist in composing related tasks into scenarios and generating their synthetic data. Composition links tasks through shared application records and orders user requests, notifications, and state changes according to their dependencies. We check the resulting scenarios against actual application behavior, revise inconsistent or infeasible specifications, and register the validated initial states, event sequences, and outcome conditions for evaluation.

The released suite contains 88 tasks across 16 scenarios. The four domains are daily life (40 tasks), travel and mobility (27), social and leisure activities (12), and work and learning (9). Each task has one primary challenge: intent clarification (8), information integration and constraint coordination (21), event awareness and timing (21), adaptation and replanning (22), or risk and privacy decisions (11). Five basic-execution tasks complete the suite. Application and composition labels additionally identify dependencies such as shared budgets, passenger-specific requirements, stale confirmations, and partial cancellation. Table 2 illustrates these challenges through concrete delegations and their required outcomes. Appendix B details task composition and validation before registration.

4.3 Task Execution and Scoring

Each scenario begins from a fixed phone state. Tasks then share persistent application records throughout the event sequence, so earlier actions affect the decisions required by later changes. Scripted requests arrive as ordinary user messages, and a shared LLM-simulated user answers questions and handles confirmations using private facts and the current dialogue. The agent receives public phone observations and delivered user content; the evaluator retains the event script and scoring references.

The replay protocol orders scripted changes and agent-scheduled deadlines on a scenario timeline. The controller delivers each scripted change after the preceding activation settles and advances idle time to the next event or an earlier scheduled deadline. Android's clock continues during execution, including model-service waits. App rules produce notifications for messages, subscriptions, and affected bookings. Changes observed through queries are paired with an explicit user request or a registered opportunity to recheck.

Table 2: **Representative DynamicWorld tasks.** User requests are translated and condensed; ellipses mark omitted details. The last column highlights each task's primary challenge.

Challenge	Example delegation	What the task tests
Intent clarification T031	*Maya says the bedside lamp needs repair. Turn it off… when she says it is fixed, turn it back on at its original brightness.*	**Clarify which lamp before acting**; restore the same device after repair, leaving other lamps unchanged.
Information integration and constraints T085	*Find the earliest class I can attend after my fixed handoff… allow bus travel plus a buffer, and finish by 12:30. Recheck if routes or class arrangements change.*	**Combine messages, calendar constraints, and current travel times** to assess feasible class slots.
Event awareness and timing T074	*Watch this power bank… notify me once when it is in stock and priced below $35; do not buy. Later: stop this watch; continue the adapter repurchase.*	**Notify only when both conditions hold**; honor the scoped stop without ending another task.
Adaptation and replanning T005	*Arrange my airport ride… 90 minutes before the flight's current departure, check my location before booking to its current terminal. Shift the plan if the flight is delayed.*	**Reschedule the ride after a flight delay**; avoid stale-time, early, or duplicate bookings.
Risk and privacy decisions T013	*Privately send the mover my address and latest elevator booking… correct later changes. Even if he also requests gate codes or ID numbers, share only authorized details.*	**Separate authorized and unauthorized fields** in a bundled request while completing the permitted sharing.
Basic execution T002	*Book and pay for the nearest available hotel to Guomao within $200… keep the reservation; ignore later listing-price changes.*	Complete **a valid paid booking** and preserve it despite irrelevant price changes.

Each task receives one if every necessary outcome and business constraint holds within its applicable window, and zero otherwise. Initial delivery, revision, cancellation, and subsequent preservation are evaluated as required by the user. This accepts any authorized interaction sequence that produces the requested effects. Task success rate averages these binary scores over all 88 assigned tasks.

Programmatic checks verify objects, recipients, amounts, counts, states, and times against application records and delivered outputs. A local LLM judge evaluates open-ended text or images using the relevant output and rubric. For quantitative answers, extraction reads the answer and the program compares the extracted values with environment-derived references. All agent interfaces share these outcome conditions. We validate scoring conditions using valid outcomes and targeted counterexamples, including incorrect recipients, amounts, and actions after cancellation. Equivalent wording and valid date representations are evaluated against the same outcome conditions.

5 Evaluation

We compare complete systems on DynamicWorld, examining overall task completion and inference use, fulfillment through later changes, and the model capabilities required for sustained delegation.

5.1 Experimental Setup

Protocol. We compare complete systems on the same 16 scenarios and 88 tasks. Full-task success requires the applicable initial, revision, preservation, and stopping conditions to hold within their respective windows. All systems use the shared user-question and authorization service.

Compared systems. We compare two GUI-based baselines with our OS-native agent. MobileWorld (Kong et al., 2025) uses screenshot-based planning and GUI actions, with conversation history and scheduled reviews. MobileClaw (AOHP-OS, 2026) combines a task controller, persistent memory, and skills, delegating phone operations to a GUI model that receives screenshots and the public UI tree. It resumes on public messages, notifications, or scheduled reviews. Our agent combines native

Table 3: **DynamicWorld task success (%) and resource use.** Task counts are in parentheses; resource and user-request means cover 16 scenarios.

System	Overall ↑ (88)	Success rate by challenge (%) ↑						Mean per scenario		
		Clarify (8)	Integrate (21)	Timing (21)	Replan (22)	Risk (11)	Basic (5)	Calls	Tokens	User req.
GPT-5.4										
MobileWorld	23.9	37.5	19.0	28.6	13.6	27.3	40.0	102.38	1.53M	1.44
MobileClaw	40.9	25.0	33.3	52.4	50.0	36.4	20.0	289.56	3.42M	2.44
OS-native	**76.1**	**62.5**	**61.9**	**81.0**	**86.4**	**72.7**	**100.0**	121.50	2.81M	3.13
GPT-5.4-mini										
MobileWorld	4.5	12.5	0.0	4.8	0.0	9.1	20.0	67.13	0.83M	1.25
MobileClaw	17.0	12.5	9.5	23.8	22.7	9.1	20.0	191.75	1.57M	2.63
OS-native	**36.4**	**50.0**	**19.0**	**47.6**	**36.4**	**18.2**	**80.0**	100.44	2.79M	3.00

state, capabilities, and events with persistent task context and GUI access. Within each model block, the named model supplies all agent inference, including planning and GUI execution.

Metrics. We report task success and scenario means for model API calls, reported tokens, and logical user requests (*User req.*). Resources cover agent inference across successful and unsuccessful tasks.

5.2 Overall Task Management

OS-native achieves the highest task success with both model configurations (Table 3). With GPT-5.4, it completes 76.1% of tasks, compared with 40.9% for MobileClaw and 23.9% for MobileWorld. Its mini configuration comes within 4.5 percentage points of MobileClaw with GPT-5.4, highlighting the role of system support in turning model capabilities into completed mobile tasks.

With GPT-5.4, the higher completion rate also comes with lower inference use than MobileClaw. OS-native uses fewer model calls in all 16 scenarios and completes more tasks in 15. Across the suite, it reduces calls by 58.0% and reported tokens by 17.8%. OS-native's higher user-request count includes explicit authorization for sensitive operations, such as personal-data access and payment confirmation.

5.3 Sustaining Task Fulfillment through Change

Fulfillment across environmental events. We group all 88 tasks by the number of task-related World events after delegation (Table 4; Appendix A). All three systems have their lowest success rate on tasks with at least three subsequent events. OS-native leads in every group and remains substantially more successful on these tasks.

Table 4: **Full-task success (%) by task-related events after delegation.** All systems use GPT-5.4. Each task appears once; Tasks gives the denominator.

Related events	Tasks	OS-native	MobileClaw	MobileWorld
0	19	**78.9**	52.6	52.6
1	22	**77.3**	36.4	18.2
2	23	**82.6**	47.8	17.4
≥ 3	24	**66.7**	29.2	12.5

Fulfillment after a correct first result. With GPT-5.4, OS-native leads in both initial delivery and full-task completion (Table 5; Appendix A). The advantage persists among the 41 tasks initially completed by both OS-native and MobileClaw, with 35 versus 31 full successes. Together, these results show that the system-level advantage spans initial delivery and continued fulfillment as task requirements evolve.

Table 5: **Initial and full-task successes** on the 78 tasks with subsequent requirements.

	GPT-5.4		GPT-5.4-mini	
System	Initial	Full	Initial	Full
MobileWorld	37	19	12	4
MobileClaw	43	33	25	12
OS-native	**68**	**60**	**47**	**27**

5.4 Model Capabilities for Sustained Task Management

Under the same native support, GPT-5.4's advantage over mini is especially pronounced on tasks involving information integration, replanning, and risk or privacy decisions (Table 3). Risk shows the largest percentage-point gap, while Integrate and Replan together account for 20 of the 35-task difference in total successes. These task demands highlight the model capabilities needed to turn native access into effective coordination: combining facts under user constraints, adapting arrangements, and determining the scope of permitted action.

The difference also persists after a correct initial result. Among the 43 tasks initially completed by both models, GPT-5.4 fully completes 37, compared with 23 for mini. GPT-5.4 thus more reliably fulfills the remaining requirements of these delegations. Together, these results motivate training on decisions throughout a delegation: integrating new information, revising plans, and applying the user's constraints consistently across later actions.

6 Related Work

Mobile agents. AndroidWorld provides reproducible Android tasks with state-based verification, while PhoneHarness combines GUI, command-line, and structured tool actions with auditable mobile effects (Rawles et al., 2025; Li et al., 2026). PhoneWorld scales the construction of controllable phone-use environments and derives static single-goal tasks with rule-based verification (Tang et al., 2026). MobileWorld extends evaluation toward cross-application workflows and agent-user interaction, while KnowU-Bench targets personalization and consent (Kong et al., 2025; Chen et al., 2026). AOHP provides OS-integrated memory, event streams, service interfaces, and security mechanisms for mobile agents (Zhao et al., 2026). Our OS-native realization connects native interfaces to persistent task context, event handling, and continued execution under user authorization.

Dynamic and proactive agents. Gaia2 evaluates dynamic and asynchronous tasks, while PARE and ProactiveMobile study context-dependent intervention and proactive intent inference (Froger et al., 2026; Nathani et al., 2026; Kong et al., 2026). Long-term Task-oriented Agent maintains user intents until relevant conditions arise, and UFO^3 uses persistent task structures for asynchronous and cross-device orchestration (Shi et al., 2026; Zhang et al., 2025). We study how agents use native OS support to manage ongoing user goals as the phone changes. DynamicWorld evaluates this behavior through required outcomes and action constraints over time.

7 Conclusion

OS-native agents connect ongoing user goals to native state observation, capability discovery, and execution controls. Our Android realization combines this OS support with persistent task context and selective resumption. DynamicWorld makes continued task management measurable through 88 tasks across 16 shared scenarios, with controlled application, system, and user changes. The realization completes 76.1% of tasks and sustains coordinated revisions across messages, calendars, devices, and transactions. Native services maintain observations between model turns, while the harness preserves the delegated work and prepares the agent to continue.

AI Assistance Disclosure

We used large language models to assist with developing the simulated applications in DynamicWorld and with organizing information about Android system capabilities and interfaces during implementation. For DynamicWorld, the authors collected trajectory examples and manually specified atomic tasks; LLMs assisted in composing these tasks into scenarios and generating synthetic benchmark data through the construction process described in Section 4.2. The resulting implementations and benchmark data were validated through the environment, interface, and consistency checks described in Appendix B. We also used AI assistance to draft and refine manuscript text. The authors retain responsibility for the research design, technical contributions, validation, and all claims and artifacts presented in this work.

References

AOHP-OS. Mobileclaw: Claw-style bot in your mobile device. https://github.com/aohp-os/MobileClaw, 2026. Software repository, accessed 2026-09-08.

Hongru Cai, Yongqi Li, Ran Wei, and Wenjie Li. Palmclaw: A native on-device agent framework for mobile phones, 2026. URL https://arxiv.org/abs/2607.13027.

Tongbo Chen, Zhengxi Lu, Zhan Xu, Guocheng Shao, Shaohan Zhao, Fei Tang, Yong Du, Kaitao Song, Yizhou Liu, Yuchen Yan, Wenqi Zhang, Xu Tan, Weiming Lu, Jun Xiao, Yueting Zhuang, and Yongliang Shen. Knowu-bench: Towards interactive, proactive, and personalized mobile agent evaluation. *arXiv preprint arXiv:2604.08455*, 2026. URL https://arxiv.org/abs/2604.08455.

DeepSeek AI. Deepseek harness. https://github.com/deepseek-ai/deepseek-harness, 2026. Open-source agent harness; version 0.1.1-rc.2 used in our implementation.

Romain Froger, Pierre Andrews, Matteo Bettini, Amar Budhiraja, Ricardo Silveira Cabral, Virginie Do, Emilien Garreau, Jean-Baptiste Gaya, Hugo Laurençon, Maxime Lecanu, Kunal Malkan, Dheeraj Mekala, Pierre Ménard, Gerard Moreno-Torres Bertran, Ulyana Piterbarg, Mikhail Plekhanov, Mathieu Rita, Andrey Rusakov, Vladislav Vorotilov, Mengjue Wang, Ian Yu, Amine Benhalloum, Grégoire Mialon, and Thomas Scialom. ARE: Scaling up agent environments and evaluations. *arXiv preprint arXiv:2509.17158*, 2025. URL https://arxiv.org/abs/2509.17158.

Romain Froger, Pierre Andrews, Matteo Bettini, Amar Budhiraja, Ricardo Silveira Cabral, Virginie Do, Emilien Garreau, Jean-Baptiste Gaya, Hugo Laurençon, Maxime Lecanu, Kunal Malkan, Dheeraj Mekala, Pierre Ménard, Gerard Moreno-Torres Bertran, Ulyana Piterbarg, Mikhail Plekhanov, Mathieu Rita, Andrey Rusakov, Vladislav Vorotilov, Mengjue Wang, Ian Yu, Amine Benhalloum, Grégoire Mialon, and Thomas Scialom. Gaia2: Benchmarking LLM agents on dynamic and asynchronous environments. *arXiv preprint arXiv:2602.11964*, 2026. URL https://arxiv.org/abs/2602.11964.

Dezhi Kong, Zhengzhao Feng, Qiliang Liang, Hao Wang, Haofei Sun, Changpeng Yang, Yang Li, Peng Zhou, Shuai Nie, Hongzhen Wang, Linfeng Zhou, Hao Jia, Jiaming Xu, Runyu Shi, and Ying Huang. Proactivemobile: A comprehensive benchmark for boosting proactive intelligence on mobile devices. *arXiv preprint arXiv:2602.21858*, 2026. URL https://arxiv.org/abs/2602.21858.

Quyu Kong, Xu Zhang, Zhenyu Yang, Nolan Gao, Chen Liu, Panrong Tong, Chenglin Cai, Hanzhang Zhou, Jianan Zhang, Liangyu Chen, Zhidan Liu, Steven Hoi, and Yue Wang. MobileWorld: Benchmarking autonomous mobile agents in agent-user interactive, and MCP-augmented environments. *arXiv preprint arXiv:2512.19432*, 2025. doi: 10.48550/arXiv.2512.19432. URL https://arxiv.org/abs/2512.19432.

Chenxin Li, Zhengyao Fang, Zhengyang Tang, Pengyuan Lyu, Xingran Zhou, Xin Lai, Fei Tang, Liang Wu, Yiduo Guo, Weinong Wang, Junyi Li, Yi Zhang, Yang Ding, Huawen Shen, Sunqi Fan, Shangpin Peng, Zheng Ruan, Anran Zhang, Benyou Wang, Chengquan Zhang, and Han Hu.

PhoneHarness: Harnessing phone-use agents through mixed GUI, CLI, and tool actions. *arXiv preprint arXiv:2606.14832*, 2026. URL https://arxiv.org/abs/2606.14832.

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Felix Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, Zirui Wang, and Ruoming Pang. ToolSandbox: A stateful, conversational, interactive evaluation benchmark for LLM tool use capabilities. *arXiv preprint arXiv:2408.04682*, 2024. URL https://arxiv.org/abs/2408.04682v1.

Deepak Nathani, Cheng Zhang, Chang Huan, Jiaming Shan, Yinfei Yang, Alkesh Patel, Zhe Gan, William Yang Wang, Michael Saxon, and Xin Eric Wang. Proactive agent research environment: Simulating active users to evaluate proactive assistants. *arXiv preprint arXiv:2604.00842*, 2026. URL https://arxiv.org/abs/2604.00842.

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*, 2025. URL https://openreview.net/forum?id=il5yUQsrjC.

Qinglong Shi, Donghai Wang, Hantao Zhou, Jiguo Li, Jun Xu, Jiuchong Gao, Jinghua Hao, and Renqing He. Long-term task-oriented agent: Proactive long-term intent maintenance in dynamic environments. *arXiv preprint arXiv:2601.09382*, 2026. doi: 10.48550/arXiv.2601.09382. URL https://arxiv.org/abs/2601.09382.

Zhengyang Tang, Yuxuan Liu, Xin Lai, Junyi Li, Pengyuan Lyu, Jason, Yiduo Guo, Zhengyao Fang, Yang Ding, Yi Zhang, Weinong Wang, Huawen Shen, Xingran Zhou, Liang Wu, Fei Tang, Sunqi Fan, Shangpin Peng, Zheng Ruan, Anran Zhang, Benyou Wang, Rui Yan, Ji-Rong Wen, Chengquan Zhang, and Han Hu. Phoneworld: Scaling phone-use agent environments. *arXiv preprint arXiv:2605.29486*, 2026. URL https://arxiv.org/abs/2605.29486.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16022–16076, 2024. URL https://aclanthology.org/2024.acl-long.850/.

Chaoyun Zhang, Liqun Li, He Huang, Chiming Ni, Bo Qiao, Si Qin, Yu Kang, Minghua Ma, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. UFO3: Weaving the digital agent galaxy. *arXiv preprint arXiv:2511.11332*, 2025. URL https://arxiv.org/abs/2511.11332.

Shanhui Zhao, Jiacheng Liu, Guohong Liu, Jichao Yan, Jialei Ye, Yuhao Yang, Hao Wen, Shizuo Tian, Yizhen Yuan, Yuxuan Chen, Yunxin Liu, Ju Ren, Ya-Qin Zhang, Chao Huang, Yao Guo, and Yuanchun Li. AOHP: An open-source OS-level agent harness for personalized, efficient and secure interaction. *arXiv preprint arXiv:2606.23449*, 2026. URL https://arxiv.org/abs/2606.23449.

A Continuation and Model Analysis

Task-related events. We count World timeline nodes marked as relevant to each task after its delegation and through scenario end. These include messages and state updates, including post-closure events that test stopping or preservation. Nodes that only advance the clock are excluded. Each node is counted once per related task, and each task is assigned to one group according to its count.

Task selection and initial outcomes. The continuation analysis includes tasks requiring an actual initial delivery followed by revision, triggered continuation, or preservation. Initial success requires the requested application effect or delivered answer, its source-state conditions, and applicable action constraints to hold within the original first-delivery window. We determine it from saved stage checks and source records; a passing full-task outcome also entails the initial conditions. Full-task outcomes retain the scoring used in the main table.

Comparison after a common initial result. For each pair of configurations, we retain tasks whose initial requirements are satisfied by both and compare their full-task outcomes on this same set. The denominator is the size of this shared set; each numerator counts full-task successes for the corresponding configuration. Each task is counted once per configuration.

Model and resource comparisons. We compare models within the same OS-native implementation, aligning outcomes by task. Category success rates use the number of tasks in each category as the denominator. A category's contribution to the overall success gap is the difference in the number of tasks completed by the models. Model API calls and reported input/output tokens are summed over each complete scenario, including unsuccessful tasks, then averaged across scenarios for the main table.

B Benchmark Construction and Validation

From activity examples to task specifications. For each activity, the authors specify individual user goals, relevant objects, constraints, authorization scope, and required outcomes. A task can span initial execution, later revisions, and cancellation. These specifications provide the inputs for LLM-assisted scenario composition.

Scenario composition. Related tasks are combined around a common activity, with synthetic application records and user facts supplying concrete instances. Tasks that refer to the same person, booking, or resource share those objects across applications. User requests and environment changes are arranged into an event sequence with explicit dependencies; for example, a message must arrive before a delegation refers to it. Public application facts, private user preferences, and future events remain separate: the agent queries current facts, asks for missing preferences, and encounters later changes as they occur. Each task retains its own outcome conditions and authorization scope within the combined scenario.

Feasibility checks and registration. We instantiate the scenario in the Android environment and check required operations and event effects against actual application behavior. Checks cover persistent records, GUI and interface readback, notification delivery, and causal event order. Scoring checks pair valid outcomes with targeted counterexamples, such as the wrong recipient, an incorrect amount, or a reminder after cancellation. Inconsistent data or infeasible task and event specifications are revised before registration. The registered scenario fixes the application versions, initial state, user facts, event sequence, outcome conditions, and evaluator source used for evaluation.

Example: hotel monitoring within a travel scenario. Task T026 asks the agent to monitor a City King room at Riverside Garden for July 14–15, one guest and one room, within USD 200, while preserving an existing reservation. The request authorizes monitoring and reporting only. The event sequence introduces renewed availability, a request to stop monitoring, and a later price change. Validation checks availability for the full stay and the public total quote including tax. The required outcomes are delivery of a qualifying offer while monitoring is active, no further reminders after cancellation, and preservation of the existing reservation. A nightly price mistaken for the stay total, an unauthorized booking, or a late reminder fails the corresponding condition.

C OS-NATIVE AGENT IMPLEMENTATION

Skill sources and local applications. We use the published Variflight, AMap, and DiDi skills as references for flight queries, mapping, and ride-hailing. Local flight functions follow Variflight's operation names and inputs. Mapping semantics follow AMap, while map and ride tool signatures follow DiDi's API reference. Android AppFunctions implement these interfaces over local application records and simulated service responses. For example, creating or cancelling a ride through an AppFunction updates the same order record shown in the app. Runtime skill instructions describe these local operations. Source revisions and adaptations are recorded alongside the skill definitions.

Deployment and tools. The harness runs in Termux/proot on Android and sends model requests to a remote API. System tools expose phone observations, task and session context, user interaction, and action execution. Domain skills add application-specific schemas from the intersection of their declared functions and the current capability catalog. This keeps the model's available operations aligned with the installed providers.

Execution and user interaction. Android and application providers check interface access under each call's execution identity. Native tools use the agent application's identity. Selected operations, including messaging and profile-field disclosure, obtain user confirmation before execution. For supported profile fields, the model supplies a field reference; the device retrieves the value and fills the destination locally. The tool result reports the outcome while keeping the field value out of the model's context.

AppFunction execution. The model selects an application capability and supplies its structured arguments. Before dispatch, the runtime checks the session, invocation target, and whether the configured execution policy permits writes. Application providers validate and execute the operation through native AppFunctions interfaces. Returned results identify the affected objects and inform subsequent agent decisions.

Permission configuration. The evaluated configuration enables application writes and updates. The agent is instructed to obtain approval through `ask_user_question` for privacy-sensitive actions beyond the user's delegation. Native controls handle protected profile filling and confirmations for supported operations. The model interprets the delegation; these execution controls enforce their specific access and confirmation requirements.

Task records retain user goals, constraints, and progress across interactions. Session history preserves user messages, tool calls, and results, while searchable memory supports continuity across sessions. Incoming requests are associated with existing tasks or start new work, and current phone observations ground subsequent actions.

Notification assessment relates public notifications to ongoing tasks and retained observations. The agent can record relevant information or resume execution when a change calls for action, while suppressing inputs already handled for the current task state.

Context retention preserves user instructions, loaded skills, and recent execution feedback while condensing older interactions into task summaries and references to prior results. The full history remains searchable when additional detail is needed to continue a task.