AOHP: An Open-Source OS-Level Agent Harness for Personalized, Efficient and Secure Interaction

Shanhui Zhao[1*], **Jiacheng Liu**[2*], **Guohong Liu**[1*], **Jichao Yan**[1], **Jialei Ye**[2], **Yuhao Yang**[3], **Hao Wen**[1], **Shizuo Tian**[1], **Yizhen Yuan**[1], **Yuxuan Chen**[1], **Yunxin Liu**[1†], **Ju Ren**[1], **Ya-Qin Zhang**[1], **Chao Huang**[3], **Yao Guo**[2], **Yuanchun Li**[1†]
[1]Tsinghua University [2]Peking University [3]The University of Hong Kong

[*]Co-primary authors. [†]Corresponding Authors ({liuyunxin,liyuanchun}@air.tsinghua.edu.cn)
Source Code: https://github.com/aohp-os/aohp

AI agents are driving a new software paradigm, with the ability to autonomously call tools, extract information, manage memory, and complete tasks that span applications and data sources. Most existing end-user operating systems, however, are designed for application-centric workflows and offer little native support for AI agents. This mismatch limits the wider adoption of agents and leads to execution overhead and safety risks when running agents on conventional systems.
While the concept of agent-native operating systems is emerging, the research community lacks an open testbed to explore the architectural primitives desired for agent-mediated interaction. We present AOHP (Android Open Harness Project), an OS-level agent harness built on the Android Open Source Project (AOSP). The core design principle of AOHP is to treat agents as first-class OS actors, enabling adaptive user interfaces and agent-friendly runtime environments. AOHP preserves the mature Android software and hardware ecosystem while introducing three agent-oriented system mechanisms: personalized service composition, efficient agent interfaces, and secure information flow. Based on preliminary experiments on challenging tasks covering key capabilities of OS agents, AOHP shows clear advantages in task completion (+21.12% completion rate), execution cost (-51.55% token cost), and security-policy compliance.

1. Introduction

AI agents are fundamentally changing how humans interact with digital environments, moving from isolated chat interfaces into the operating system itself. Modern agents can call command-line tools, inspect graphical interfaces, use application APIs, and coordinate multi-step tasks [25, 13, 2]. Agent-mediated interaction changes the role of the operating system: the OS still hosts applications, but it also needs to support agents that observe services, invoke capabilities, and enforce user intent across app boundaries.

Conventional personal operating systems are app-centric by design. Interfaces are fixed by system and application developers. GUIs are rendered for direct human visual operation rather than agent manipulation. Software lifecycle assumes one active app at a time, and permission mechanisms enforce protections at app boundaries but fail to track sensitive data across an agent's context and tool calls. These assumptions create significant execution overhead and security gaps for agent-mediated workflows.

AOHP addresses this gap by redesigning Android as an agent-native harness via OS-level abstractions and framework modifications. It preserves Android's hardware support, open-source framework, and app compatibility while adding mechanisms that make services composable, efficient, and auditable for OS-level agents.

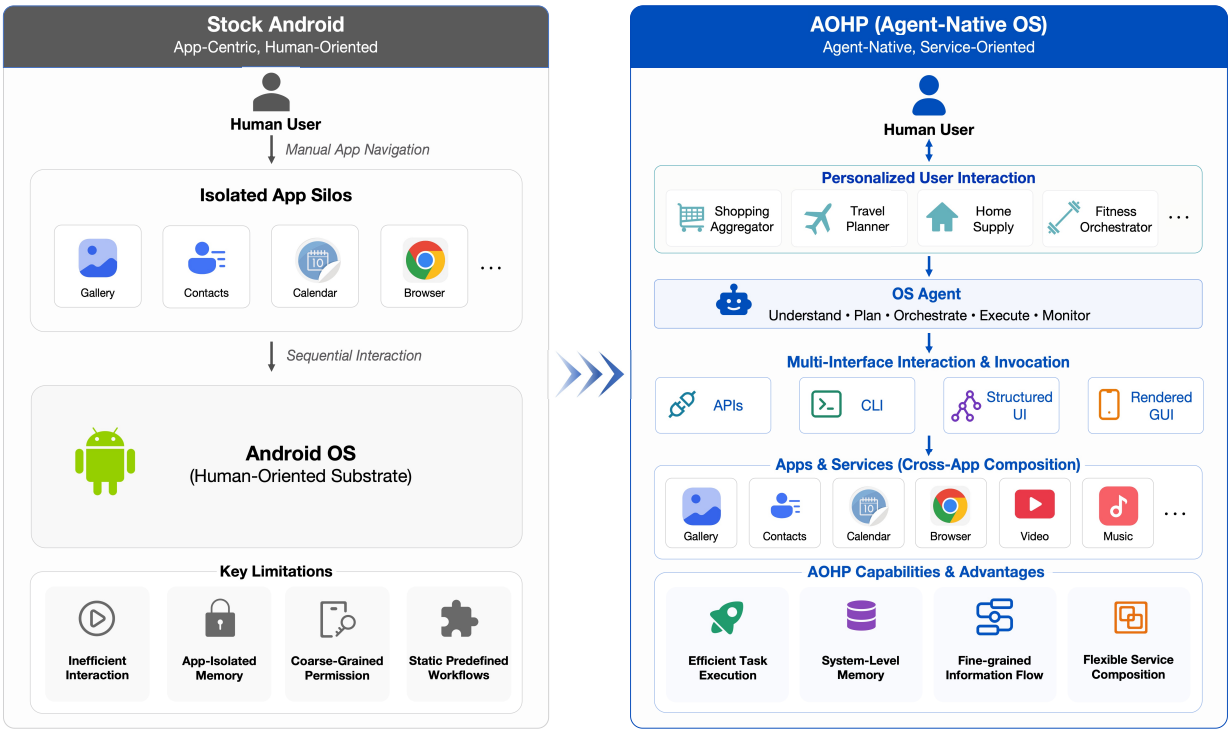

Figure 1 | App-centric Android and agent-native AOHP.

With such OS-level supports, AOHP is able to create a new experience where **the OS can proactively adapt to the user** by generating personalized service interfaces around user intent, rather than conventionally letting users adapt to the predefined fixed OS and app interfaces. For example, instead of requiring manual switching among individual shopping apps, AOHP can expose a task-level shopping entrance that aggregates product information and purchase actions across multiple services. The user interacts with the high-level concept of shopping; the OS agent handles service discovery, invocation, memory management, and policy enforcement under the hood. Figure 1 illustrates this difference between app-centric Android and agent-native AOHP.

In summary, this report makes the following contributions:

- **Agent-Native OS Architecture:** We identify the architectural mismatch between app-centric OSes and agent workflows, proposing a design where services are interface-neutral capabilities, and the OS manages cross-app personalization and sensitive state.
- **System Implementation:** We design and implement AOHP on AOSP, introducing three core mechanisms: (1) *personalized service composition* for synthesizing task-level entrances; (2) *efficient agent interfaces* supporting parallel background execution, structured UI, and event streams; and (3) *secure information flow* that sandboxes sensitive values via information flow tracking.
- **Empirical Evaluation:** We evaluate AOHP using OpenClaw agents on a set of self-crafted mobile tasks that demand complex cross-app interaction. Compared to stock Android, AOHP raises the average completion rate from 54.44% to 75.56%, reduces LLM token consumption by 51.55% and accelerates task execution by 44.21%. Security case studies confirm that AOHP effectively restricts private plaintext exposure while preserving legitimate task execution.

2. Background & Motivation

The need for AOHP comes from a change in *how* users interact with digital services.

2.1. Emerging Demand of Adaptive User Interfaces

Traditional operating systems expose functionality through app-defined interfaces. Users see menus, buttons, pages, and workflows chosen by the OS and app developers. This means that users can only passively adapt to the services and information determined by the developers, which may be tedious and distracting, even biasing the users' perception. The issue becomes increasingly severe as the apps are growing larger and dominating.

A promising solution to develop adaptive user interfaces. In an adaptive OS, the user interface can be deeply personalized around user intent. For example, a user who shops across multiple marketplaces should not need to reason about which app owns which capability. Instead, the OS can generate a shopping entrance that aggregates product search, comparison, coupons, delivery constraints, and purchase actions from different services. The entrance is not a static app installed by one developer; it is assembled from app, GUI, CLI, and API interfaces and personalized by system memory.

2.2. Agents Become a Major Workload

AI agents are promising to enable such adaptive experiences. Modern agents can operate applications, issue commands, and interact with web or app services. In many workflows, the agent can serve as an execution layer between user intent and app-level operations: it reads application state, invokes system functions, transfers data across services, waits for asynchronous events, and acts repeatedly over time.

Agent-mediated interaction exposes a gap in conventional OS design. Existing systems assume app-mediated interaction, while agents have different operational properties. They process structured text more efficiently than pixels, run multiple tasks in parallel, retain long-lived context, and make tool calls faster than users can inspect them. A system designed around app-mediated interaction cannot make agent execution efficient or safe by construction.

2.3. Why We Need A OS-level Harness

To make the agent-mediated adaptive user interfaces feasible, efficient and secure, many components in the system and framework layers should be redesigned, while many other componnents can be reused. Therefore, building a harness over an existing mature OS is an ideal choice. We choose Android as the substrate for AOHP for practical reasons.

Rich application ecosystem. Android offers a broad app ecosystem covering communication, productivity, commerce, content, and device control, which is useful to foster rich custom services.

Mature hardware support. Android already runs on a wide spectrum of devices with established support for drivers, sensors, networking, and power management.

Open-source accessibility. The Android Open Source Project allows deep modifications to system services, framework layers, UI stacks, and runtime policies, which makes it a practical base for fine-grained systems research [8].

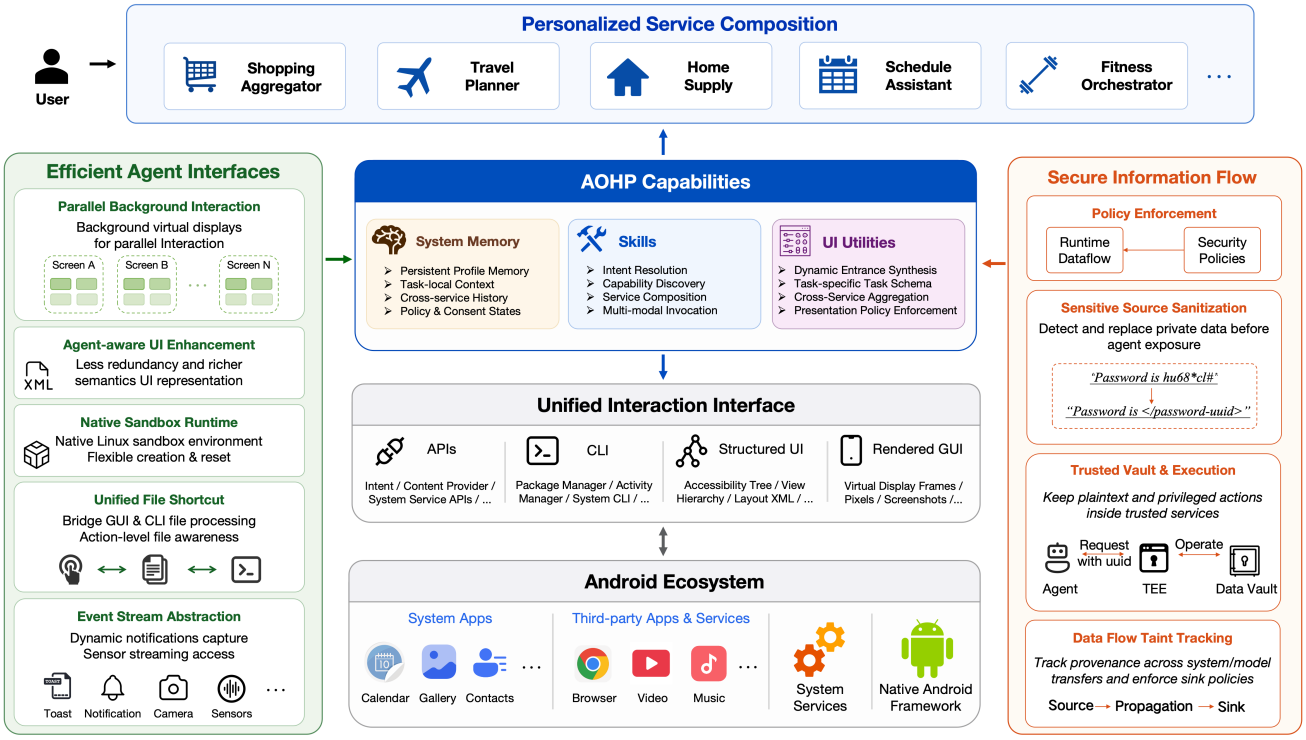

Figure 2 | Architecture overview of AOHP.

3. System Design

AOHP turns the requirements above into three design principles. First, **user interfaces should be composed on demand**: the OS agent should break app and service silos by discovering and recomposing capabilities across providers into personalized interaction surfaces driven by the user's intent, context, and state. Second, **agent-service interaction should be compatible, efficient, and interface-neutral**: the OS should provide a unified substrate that preserves legacy Android app interaction while supporting agent-oriented service logic, so agents can access capabilities with lower overhead and higher accuracy. Third, **sensitive data should be isolated by default**: agents should not observe private plaintext by default, while the system enforces fine-grained information-flow tracking, approval, and audit throughout the task.

Figure 2 shows the resulting architecture. Vertically, AOHP is organized into four layers. The bottom layer is the **Android ecosystem**, which keeps existing apps, system services, hardware resources, and platform APIs as the compatibility base. Above it, the **unified interaction interface** normalizes both traditional Android interfaces and emerging agent interfaces into four invocation modes: API, CLI, Structured UI, and Rendered GUI. This layer lets agents select a compact symbolic path when available and fall back to visual operation when compatibility requires it.

The third layer is the **AOHP capabilities** layer. It reorganizes services and agent functions provided by apps, system components, and agent tools into system memory, skills, and UI utilities. System memory stores preferences, task state, histories, and policy decisions outside any single app. Skills package reusable service capabilities and execution routines. UI utilities support the construction of generated

entrances and task-specific interaction surfaces. The top layer is **personalized service composition**, where these capabilities are assembled into new user-facing application surfaces tailored to the user's current task. This layer represents the application model enabled by AOHP: the user interacts with a personalized service entrance, while the OS agent resolves the underlying service graph and execution path.

Horizontally, AOHP includes two cross-layer mechanisms. **Efficient agent interfaces** optimize how agents access system resources, app state, files, events, and isolated execution substrates. They reduce visual-processing overhead, avoid unnecessary foreground serialization, and make agent actions less dependent on brittle GUI navigation. **Secure information flow** provides the stricter protection model required by agent-mediated execution. It sanitizes sensitive sources, routes private values through trusted vault operations, propagates taint metadata, and records auditable traces before sensitive data reaches external sinks or state-changing actions.

The following sections detail the three core parts of this design. Section 4 explains how AOHP discovers and recomposes underlying capabilities into personalized service entrances. Section 5 describes the agent-native interfaces that make service access and task execution efficient. Section 6 presents the security mechanisms that isolate private data and enforce system-level information-flow policies.

4. Personalized Service Composition

The most visible change in AOHP is that interaction surfaces become personalized and generated rather than fully predefined. A conventional app exposes developer-chosen functions. AOHP lets the OS agent synthesize service entrances around recurring goals, shifting interaction from app navigation to task-level service access.

4.1. Generated Service Entrances

Generated entrances are user-facing shells backed by OS-managed service composition. A shopping entrance can aggregate product search from multiple providers, normalize product attributes, apply preferences such as size and budget, and expose a task-specific interface for comparison and purchase.

Each entrance contains three parts: a task schema, a service graph, and a presentation policy. The task schema defines what the user is trying to accomplish, such as "compare shoes under a budget" or "refill household supplies". The service graph maps this task to concrete service capabilities. The presentation policy decides which intermediate results should be shown to the user and which can remain agent-internal. This separation lets AOHP personalize the entrance without hiding consequential decisions.

4.2. Capability Discovery and Composition

AOHP builds these entrances by discovering service capabilities across API, CLI, and GUI channels. Each capability is represented with input/output schemas, preconditions, side effects, and policy labels. The OS agent can then compose capabilities into a higher-level workflow. This design lets legacy apps participate through GUI export while allowing newer services to expose more direct APIs.

Composition is constrained by policy. For example, product search may be freely parallelized across

providers, whereas purchase submission is a state-changing action that requires explicit confirmation. Similarly, a delivery address can be used to estimate shipping only through the information-flow sandbox. The generated entrance is both a convenience layer and a policy-enforcement surface.

4.3. Cross-Service Personalization

System memory lets personalization survive app boundaries. Preferences learned while using one service can improve another service, subject to policy. For example, a user's preferred delivery window learned from one shopping workflow can be used when comparing products from another marketplace. The design requirement is that memory remains OS-mediated: personalization can be shared, audited, and revoked without depending on each app's private data model.

AOHP distinguishes between persistent profile memory, task-local memory, and sensitive memory. Persistent profile memory stores stable preferences. Task-local memory stores temporary state, such as candidate products or partially completed forms. Sensitive memory stores private values through sandbox indices. This distinction prevents personalization from becoming an uncontrolled accumulation of private context.

5. Efficient Agent Interfaces

AOHP exposes execution environments, UI semantics, storage, and events as agent-native primitives. These abstractions reduce visual-processing overhead, rigid sequential execution, and brittle cross-app handoffs in agent-mediated workflows.

5.1. Parallel Background Interaction

Conventional mobile OSes couple app lifecycles to the physical display. AOHP decouples execution from the screen through lightweight virtual displays, allowing agents to run waiting-heavy or independent workflows in the background without preempting the active foreground session.

5.2. Agent-aware UI Enhancement

Conventional app GUIs contain rendering details that are unnecessary for agent reasoning. AOHP abstracts GUIs into structured representations with lower redundancy and richer semantics, while retaining rendered GUI fallback for visual components.

5.3. Native Sandbox Runtime

App-mediated GUI, API, and CLI paths do not cover all agent work. Agents often need a local execution substrate for computation, transformation, and tooling. AOHP includes a native, OS-managed sandbox runtime that can be created and reset as an execution surface independent of app-facing interfaces. Agents can execute code, process data, and host long-running services inside the sandbox, then return structured results to the task context without placing intermediate steps in the agent context.

5.4. Unified File Shortcut

Cross-app agent workflows often rely on files as shared intermediate artifacts, such as saving an attachment in one app and reusing it in another. On stock systems, these artifacts remain implicit: GUI actions may create or modify storage, but agents lack a stable, OS-level account of the outcome. Conversely, programmatic file operations have no uniform way to invoke app-native affordances such as system share flows when the destination expects them. AOHP treats files as first-class task objects at the OS boundary. GUI interactions that affect storage are reflected back as structured file observations, so agents can reason about what changed without inferring paths from screenshots or per-app pickers. In the reverse direction, the same layer lets agents hand a resolved artifact to another interface, either through direct programmatic access or by launching the appropriate system UI flow on a chosen display. This unifies file-producing GUI steps and file-consuming programmatic steps into a single cross-app data plane and reduces brittle handoffs across opaque storage layouts.

5.5. Event Stream Abstraction

Operating systems continuously generate asynchronous and transient events that are difficult to capture with request-response interfaces. AOHP introduces an *Event Stream* abstraction that lets agents subscribe to, process, and unsubscribe from continuous data sources. It currently supports two stream types:

- **Dynamic Notification Capture:** Transient system events, such as Toasts, pop-ups, or push notifications, often disappear before an agent can poll them. AOHP implements a notification buffer to intercept and retain these short-lived messages, so agents do not miss critical UI context.
- **Sensor Streaming Access:** To perceive the physical environment, AOHP streams hardware sensor data (e.g., accelerometer, gyroscope, microphone, or camera events). Agents can process real-time physical states without repeated polling.

The abstraction separates event generation from consumption, allowing agents to react to system and environmental changes without repeated polling.

6. Secure Information Flow

AOHP treats sensitive data as OS-controlled state rather than agent-visible context. By default, private plaintext is replaced with typed references before it reaches the agent; trusted system components mediate plaintext operations, external transfers, and approvals while preserving evidence for audit. This model is necessary because agent tasks cross app, tool, memory, and service boundaries where conventional app permissions cannot track how private data propagates.

6.1. Policy Enforcement

AOHP enforces privacy policies over runtime data use rather than only over static permissions or application identities. For each sensitive operation, the policy layer evaluates the data source, requested purpose, destination, action sensitivity, and approval state. This lets ordinary non-sensitive flows proceed while requiring consent or blocking transfers and state-changing actions involving private data.

The same policy context makes authorization more understandable to users. When approval is needed, AOHP can explain the requested use in terms of source, purpose, destination, and downstream effect,

rather than presenting an opaque permission prompt. Thus, enforcement is tied to the concrete use of private data in the task.

6.2. Sensitive Source Sanitization

AOHP adopts a conservative protection strategy for sensitive sources. Before sensitive content enters the agent context, AOHP replaces plaintext with typed placeholders such as `<payment-card:uuid>` or `<home-address:uuid>`. These placeholders preserve task-level meaning while hiding the underlying value.

The system maintains a data vault that stores sensitive values behind opaque identifiers. AOHP sanitizes supported sources, including application pages, files, event streams, API responses, system memory, and user interactions. Developer-provided annotations can mark sensitive fields explicitly; when annotations are unavailable, AOHP applies conservative detection rules to protect likely sensitive content.

6.3. Trusted Vault and Execution

When an agent needs to operate on sensitive information, it submits an intent to a trusted vault executor using the corresponding placeholder. The executor checks policy, obtains user approval when necessary, and performs operations such as formatting, comparison, validation, or composition inside the trusted environment. If the result is still sensitive, the executor returns another placeholder rather than plaintext.

The trusted executor also mediates sensitive transfers to external interfaces. For GUI use, it can fill an approved field directly; for API or CLI use, it can substitute plaintext at the system boundary while keeping it out of the agent context. This design lets agents complete tasks that require private data while exposing sensitive values only to trusted system components.

6.4. Data-Flow Taint Tracking

Sanitization protects sensitive sources at entry points; taint tracking preserves their provenance after they are used. Once sensitive data enters AOHP, it is associated with taint metadata that follows the value through copying, transformation, composition, and transfer. This allows AOHP to preserve the information-flow chain even when private data is used indirectly across multiple task steps.

At system exits and other policy-relevant boundaries, AOHP checks tainted data before it is displayed, stored, submitted, or transmitted. The resulting taint path also provides an audit trail for explaining which source reached which sink through which task steps. In this way, the agent can reason over sensitive references, while the operating system remains responsible for tracking and controlling when private data may leave trusted components.

7. Evaluation

We evaluate AOHP with the OpenClaw agent and compare it against stock Android. Stock Android is the app-centric baseline: the agent receives conventional GUI observations and operates through ordinary app interactions. AOHP is the agent-native setting: the same agent additionally uses generated service entrances, structured UI observations, system CLIs/APIs, virtual execution, and sandboxed information flow.

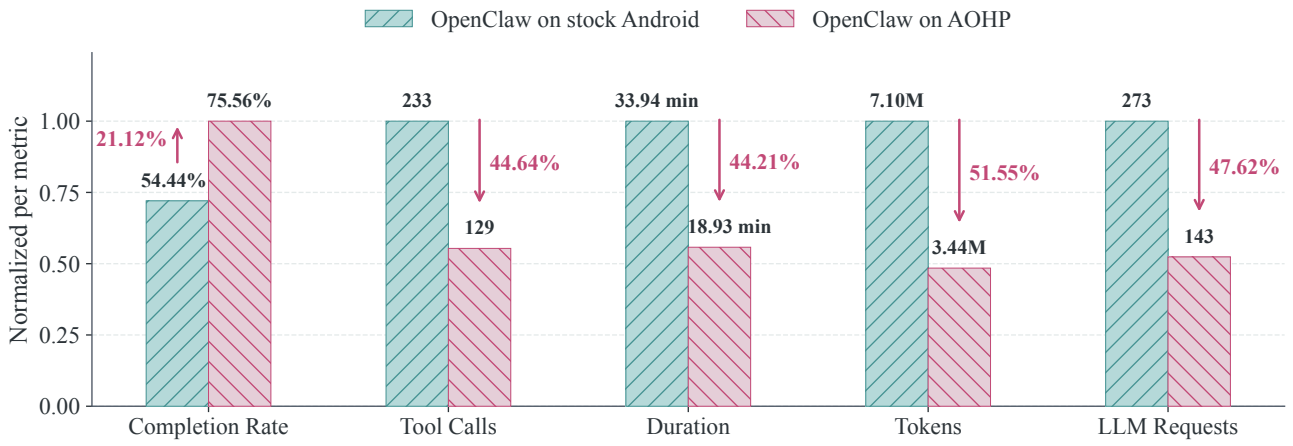

Figure 3 | Overall comparison between OpenClaw on AOHP and stock Android. Completion rate is averaged over all tasks; tool calls, duration, tokens, and LLM requests are aggregated over the tasks both settings solve completely.

Our benchmark targets the capabilities required for *personalized service composition*. It contains 30 real-world mobile tasks built exclusively from real applications. The tasks span five core capability categories: (i) GUI operation, (ii) non-GUI operation, (iii) event capture, (iv) multi-source information retrieval, and (v) memory management. A sixth hybrid category composes these primitives into longer cross-capability workflows, mirroring the service-composition setting that AOHP is designed for. Each category contains five tasks; Appendix A lists all of them. We measure task completion, success count, tool calls, duration, token usage, LLM requests, and security-policy behavior.

7.1. Functionality and Performance

We run all benchmark tasks with the OpenClaw agent under both settings. Each task is scored at the granularity of objective checkpoints, so a task that is only partially completed still earns partial credit; we report the resulting checkpoint-weighted *completion rate*. Figure 3 summarizes the outcome.

On AOHP, OpenClaw reaches a 75.56% average completion rate (20 tasks fully solved and 5 tasks partially completed). With the same agent on stock Android, the completion rate drops to 54.44% (13 tasks fully solved and 7 tasks partially completed). AOHP thus raises the average completion rate by 21.12 points and fully solves seven more tasks than the baseline. The gains concentrate on tasks involving transient notifications, fine-grained in-app GUI manipulation, and multi-step cross-app or memory-dependent workflows, where AOHP leverages the interfaces provided in Section 5 to improve the capabilities of the agent.

7.2. Efficiency Analysis

To compare execution cost on equal footing, we restrict the efficiency analysis to the 11 tasks that both settings solve completely. This removes the confound of unequal task difficulty and isolates the cost of producing the same successful outcome. Table 1 reports the aggregates over this commonly-solved subset, and we analyze the two cost dimensions below.

Table 1 | Execution cost on the tasks solved completely by both settings.

Setting	Tool Calls	Duration (min)	Tokens	LLM Requests
OpenClaw on stock Android	233	33.94	7,103,192	273
OpenClaw on AOHP	**129**	**18.93**	**3,441,759**	**143**
Reduction	44.64%	44.21%	51.55%	47.62%

Tool Calls and Duration. Even though the baseline agent can drive some operations through direct shell commands, stock Android offers few agent-native interfaces, so cross-app workflows and in-app GUI operations still rely largely on GUI navigation—climbing view hierarchies, scrolling, and re-issuing taps as interface depth grows. AOHP shortens these paths through the *Unified File Shortcut*, *Agent-aware UI Enhancement*, and *Event Stream Abstraction*, which offer more agent-oriented interaction. The savings therefore come from interface-intensive and data-processing workflows rather than a uniform per-step speedup, yielding a 44.64% reduction in tool calls and a 44.21% reduction in total duration across the commonly-solved subset.

Token Consumption and LLM Requests. Token cost grows with the number of steps, since the tool calls and observations accumulated during execution keep enlarging the context. Because AOHP completes the same tasks in fewer steps and returns more compact observations, both prompt length and the number of round-trips shrink. On the commonly-solved subset, AOHP cuts total tokens by 51.55% (3.44M vs. 7.10M) and LLM requests by 47.62% (143 vs. 273), with input tokens down 51.50% and output tokens down 57.48%.

7.3. Information-Flow Security

AOHP evaluates information-flow security with a purpose-built annotated payment application. The app exposes account information, payment credentials, transfer inputs, confirmation actions, invoice files, and transaction-related event streams. Around this app, we construct case-driven tests that exercise source sanitization, sink and action mediation, vault-token use, taint propagation, file handling, event handling, and fail-closed behavior. Table 2 summarizes the main enforced cases. In our prototype evaluation on this annotated payment app, AOHP enforced all five cases in Table 2 as expected.

Table 2 | Security checks using an annotated payment application.

Check	Expected Behavior	Result
Sensitive display	Account, card, phone, and transaction fields appear as vault references in the agent-visible UI, not plaintext.	Pass
Ordinary actions	Non-sensitive controls and ordinary files are allowed without extra approval.	Pass
Sensitive actions	Transfer fields, payment confirmation, and sensitive file sharing require user consent.	Pass
Unsupported access	Requests outside the declared policy scope fail closed instead of exposing sensitive data.	Pass
Sensitive events	Transaction-related event streams redact sensitive fields and preserve taint metadata.	Pass

8. Related Work

8.1. GUI Agents and Mobile Task Automation

Recent work has shown that large-language-model-based agents can operate computers and smartphones through graphical interfaces. Existing approaches span modular systems and end-to-end policies. Modular systems such as the AutoDroid series, Agent S, and OS-Copilot compose planners, retrievers, memory modules, and grounding components to improve reuse and error recovery [17, 18, 1, 20]. End-to-end approaches such as AppAgent, UI-TARS, Mobile-Agent, and SeeAct directly infer actions from current observations, often using multimodal models over screenshots or UI descriptions [28, 14, 29, 16, 26]. Related benchmarks, such as AndroidWorld, OSWorld, AndroidLab, and MobileWorld, evaluate agents in dynamic GUI environments [15, 22, 23, 10].

These systems improve GUI-agent performance, but they mainly treat the operating system as a fixed substrate. AOHP studies the execution environment itself: instead of proposing only a stronger agent policy, it redesigns the OS interfaces and enforcement mechanisms that such policies depend on.

8.2. Android Automation and Interface Abstraction

Before the rise of LLM agents, Android automation and testing systems had already explored how to interact with mobile interfaces programmatically. Systems such as DroidBot emphasized UI-guided exploration, while earlier instruction-following work mapped natural language requests into mobile action sequences [12, 11]. More recent interface-understanding models, including ScreenAI, OS-ATLAS, Aguvis, and other GUI grounding methods, improve semantic interpretation of screen content [3, 21, 24, 4].

AOHP uses screen understanding as one component. Existing automation and GUI-agent systems still inherit app-centric rendering paths, foreground-oriented execution assumptions, and app-defined interaction surfaces. AOHP moves these concerns into operating-system abstractions, including personalized service entrances, agent-aware UI enhancement, parallel background interaction, and system-managed task traces.

8.3. System Safety, Provenance, and Recoverability

Recent security studies show that tool-using agents face a broad attack surface in which untrusted external content can hijack tool execution or exfiltrate private data. Indirect prompt injection first exposed how LLM-integrated applications blur the boundary between data and instructions [9]; benchmarks such as InjecAgent and AgentDojo further systematize these risks in realistic tool-use settings [27, 6]. These results motivate system mechanisms that mediate data, authority, and actions rather than relying only on model-level robustness.

Recent systems defenses have moved toward deterministic enforcement at execution boundaries. FIDES and f-secure LLM systems apply information-flow control to agent planners and LLM pipelines, using labels, taint propagation, and policy checks to prevent untrusted data from steering sensitive decisions [5, 19]. AOHP follows this systems-oriented direction but moves the enforcement boundary to the mobile OS. Building on the mobile taint-tracking lineage of TaintDroid [7], AOHP tracks and mediates sensitive data across app UI, files, event streams, vault references, generated entrances, and user approval loops.

9. Conclusion and Future Work

AOHP is an agent-native open fork of Android for agent-mediated personal computing. It keeps Android compatibility while adding personalized service composition, efficient agent interfaces, and secure information flow. Preliminary results with OpenClaw show higher task completion and lower execution cost than stock Android; the security case study shows that sensitive data can be redacted, tracked, and mediated at system boundaries.

Compatibility coverage. Future work should expand support for apps with custom rendering, anti-automation logic, and undocumented behavior. This requires stronger structured UI extraction, rendered-GUI fallback that fails predictably, and clearer compatibility guidelines for app developers.

Capability discovery. AOHP depends on accurate service descriptions, side-effect labels, and policy metadata. Future versions should combine developer-provided descriptors with automatic capability inference so that legacy apps can be integrated with less manual annotation.

Resource scheduling. Background execution must respect mobile resource, thermal, and memory limits. A fuller prototype should expose scheduling policies for virtual displays, sandbox runtimes, event streams, and foreground user activity.

Policy usability. Fine-grained information-flow control depends on approvals that users can understand without excessive interruption. Future work should improve approval UI, trace review, and policy explanations for purpose, destination, retention, and consent.

References

[1] Saaket Agashe, Jiuzhou Han, Shuyu Gan, Jiachen Yang, Ang Li, and Xin Wang. Agent s: An open agentic framework that uses computers like a human. In *International Conference on Learning Representations*, volume 2025, pages 22924–22946, 2025.

[2] Anthropic. Claude Code. https://docs.anthropic.com/en/docs/claude-code/overview, 2026. Accessed: 2026-06-11.

[3] Gilles Baechler, Srinivas Sunkara, Maria Wang, Fedir Zubach, Hassan Mansoor, Vincent Etter, Victor Cărbune, Jason Lin, Jindong Chen, and Abhanshu Sharma. Screenai: A vision-language model for ui and infographics understanding. *arXiv preprint arXiv:2402.04615*, 2024.

[4] Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Li YanTao, Jianbing Zhang, and Zhiyong Wu. Seeclick: Harnessing gui grounding for advanced visual gui agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9313–9332, 2024.

[5] Manuel Costa, Boris Köpf, Aashish Kolluri, Andrew Paverd, Mark Russinovich, Ahmed Salem, Shruti Tople, Lukas Wutschitz, and Santiago Zanella-Béguelin. Securing ai agents with information-flow control. *arXiv preprint arXiv:2505.23643*, 2025.

[6] Edoardo Debenedetti, Jie Zhang, Mislav Balunovic, Luca Beurer-Kellner, Marc Fischer, and Florian Tramèr. Agentdojo: A dynamic environment to evaluate prompt injection attacks and defenses for llm agents. *Advances in Neural Information Processing Systems*, 37:82895–82920, 2024.

[7] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):1–29, 2014.

[8] Google. Android open source project. https://source.android.com/, 2026.

[9] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. Not what you've signed up for: Compromising real-world llm-integrated applications with indirect prompt injection. In *Proceedings of the 16th ACM workshop on artificial intelligence and security*, pages 79–90, 2023.

[10] Quyu Kong, Xu Zhang, Zhenyu Yang, Nolan Gao, Chen Liu, Panrong Tong, Chenglin Cai, Hanzhang Zhou, Jianan Zhang, Liangyu Chen, et al. Mobileworld: Benchmarking autonomous mobile agents in agent-user interactive and mcp-augmented environments. *arXiv preprint arXiv:2512.19432*, 2025.

[11] Yang Li, Jiacong He, Xin Zhou, Yuan Zhang, and Jason Baldridge. Mapping natural language instructions to mobile ui action sequences. In *Proceedings of the 58th annual meeting of the association for computational linguistics*, pages 8198–8210, 2020.

[12] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th international conference on software engineering companion (ICSE-C)*, pages 23–26. IEEE, 2017.

[13] OpenClaw Contributors. OpenClaw. https://docs.openclaw.ai/, 2026. Accessed: 2026-06-11.

[14] Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, et al. Ui-tars: Pioneering automated gui interaction with native agents. *arXiv preprint arXiv:2501.12326*, 2025.

[15] Chris Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, et al. Androidworld: A dynamic benchmarking environment for autonomous agents. In *International Conference on Learning Representations*, volume 2025, pages 406–441, 2025.

[16] Junyang Wang, Haiyang Xu, Haitao Jia, Xi Zhang, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. Mobile-agent-v2: Mobile device operation assistant with effective navigation via multi-agent collaboration. *Advances in Neural Information Processing Systems*, 37:2686–2710, 2024.

[17] Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. Autodroid: Llm-powered task automation in android. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, pages 543–557, 2024.

[18] Hao Wen, Shizuo Tian, Borislav Pavlov, Wenjie Du, Yixuan Li, Ge Chang, Shanhui Zhao, Jiacheng Liu, Yunxin Liu, Ya-Qin Zhang, et al. Autodroid-v2: Boosting slm-based gui agents via code generation. In *Proceedings of the 23rd Annual International Conference on Mobile Systems, Applications and Services*, pages 223–235, 2025.

[19] Fangzhou Wu, Ethan Cecchetti, and Chaowei Xiao. System-level defense against indirect prompt injection attacks: An information flow control perspective. *arXiv preprint arXiv:2409.19091*, 2024.

[20] Zhiyong Wu, Chengcheng Han, Zichen Ding, Zhenmin Weng, Zhoumianze Liu, Shunyu Yao, Tao Yu, and Lingpeng Kong. Os-copilot: Towards generalist computer agents with self-improvement, 2024.

[21] Zhiyong Wu, Zhenyu Wu, Fangzhi Xu, Yian Wang, Qiushi Sun, Chengyou Jia, Kanzhi Cheng, Zichen Ding, Liheng Chen, Paul Pu Liang, et al. Os-atlas: Foundation action model for generalist gui agents. In *International Conference on Learning Representations*, volume 2025, pages 5090–5108, 2025.

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

[23] Yifan Xu, Xiao Liu, Xueqiao Sun, Siyi Cheng, Hao Yu, Hanyu Lai, Shudan Zhang, Dan Zhang, Jie Tang, and Yuxiao Dong. Androidlab: Training and systematic benchmarking of android autonomous agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2144–2166, 2025.

[24] Yiheng Xu, Zekun Wang, Junli Wang, Dunjie Lu, Tianbao Xie, Amrita Saha, Doyen Sahoo, Tao Yu, and Caiming Xiong. Aguvis: Unified pure vision agents for autonomous gui interaction. *arXiv preprint arXiv:2412.04454*, 2024.

[25] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

[26] Jiabo Ye, Xi Zhang, Haiyang Xu, Haowei Liu, Junyang Wang, Zhaoqing Zhu, Ziwei Zheng, Feiyu Gao, Junjie Cao, Zhengxi Lu, et al. Mobile-agent-v3: Fundamental agents for gui automation. *arXiv preprint arXiv:2508.15144*, 2025.

[27] Qiusi Zhan, Zhixiang Liang, Zifan Ying, and Daniel Kang. Injecagent: Benchmarking indirect prompt injections in tool-integrated large language model agents. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 10471–10506, 2024.

[28] Chi Zhang, Zhao Yang, Jiaxuan Liu, Yanda Li, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. Appagent: Multimodal agents as smartphone users. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, pages 1–20, 2025.

[29] Boyuan Zheng, Boyu Gou, Jihyung Kil, Huan Sun, and Yu Su. Gpt-4v (ision) is a generalist web agent, if grounded. *arXiv preprint arXiv:2401.01614*, 2024.

A. Benchmark Tasks

Our benchmark comprises 30 real-world mobile tasks grouped into five core capability categories plus a hybrid category that composes them, with five tasks each. Table 3 lists all tasks in capability order. Memory-management tasks (Category 5) are two-stage: stage A executes the task, and stage B asks memory-related questions.

Table 3 | Benchmark tasks grouped by capability category.

Category 1 — GUI Operation

1. Open Tulsi Gallery, then open the first image whose picture contains the text AOHP, enter edit mode, and set Brightness to +57. Brightness setting completed, end the task without exiting the brightness setting interface.
2. Open the Notes app. Change the app theme to Dark, then change Alignment to Center.
3. Open Calendar and create an event on the 15th of this month titled Development Meeting with description AOHP Development Meeting, starting at 11:15 AM and ending at 1:30 PM.
4. In Markor, merge `q1/notepad.md`, `q2/draft.md`, and `q3/minutes.md` in that order into `sprint_summary.md` with a blank line between each file, then save.
5. The Notes entry Team Roster contains Alice's mobile number. Create a new contact Alice with that number and text the number to HR at 10086.

Category 2 — Non-GUI Operation

1. Delete exactly one Markdown file at `/sdcard/Documents/AOHP/benchmark/projects/archive/2024/q4/test.md`.
2. Use Markor to open the Markdown file at `/sdcard/Documents/AOHP/benchmark/projects/archive/2024/q4/test.md`. Finish after the file is opened.
3. Move `/sdcard/Documents/AOHP/benchmark/projects/archive/2024/month_03/ledger.md` to `/sdcard/Documents/AOHP/benchmark/projects/archive/2024/month_04/ledger.md`.
4. Compute the product of all numeric cells in `/sdcard/Documents/AOHP/benchmark/data/metrics_2024.csv` and write the integer result to `/sdcard/Download/metrics_product.txt`.
5. From `/sdcard/Documents/AOHP/benchmark/data/api_metrics.json`, compute the P99 latency in milliseconds (rounded to integer) using the nearest-rank method on `samples_ms`—sort ascending, let n be the sample count, take the value at 0-based index $\lceil 0.99n \rceil - 1$—and write it to `/sdcard/Download/latency_p99.txt`.

Category 3 — Event Capture

1. HR will send an onboarding SMS shortly. When the notification arrives, read Alice's 11-digit mobile number from it and create a new contact Alice with that number.
2. A bank SMS with a login verification code will arrive soon. After the notification appears, open Notes, create a note titled Bank OTP, and write only the 6-digit code in the body.
3. In Clock, create a 5-minute timer and start it. Wait for the timer-finished notification, then send Alice an SMS saying "Lunch has been heated and is ready".
4. Bob will text you soon. After Bob's SMS notification appears, open the camera and take a photo.
5. Wait for the salary deposit bank notification. When it appears, send Alice an SMS saying "Salary's in — I'm treating you to a big dinner tonight!"

Category 4 — Multi-source Information Retrieval

Continued on next page

Table 3 — *continued from previous page*

1. Please find and compile information for the next Development Meeting in the given apps—the start time (MM-DD HH:MM; the Calendar app does not show the year, so omit the year) of the Development Meeting event in the Calendar app, the first agenda item from Meeting Prep in the Notes app, and organizer Alice's mobile number in the Contacts app. Write three lines to `/sdcard/Documents/AOHP/answers/ir_dev_meeting_brief.txt`.

2. I'm preparing for a business trip to Paris. Please find and compile the following information in the given apps—the Paris Trip arrival date in Calendar, comma-separated items still to buy from Alice's SMS in Messages, and the total spent (integer USD) on #Paris-tagged items in `travel/paris_purchases.md` in the Markor app. Write to `/sdcard/Documents/AOHP/answers/ir_paris_trip_pack.txt` (three lines—arrival date (MM-DD), shopping list, total amount as an integer).

3. Please find and compile May reimbursement information in the given apps—the three amounts on `receipt_aohp.png` in Gallery ($12.50, $8.00, $15.75) and reimbursable entries for the same month in Markor `q3/minutes.md`. Write the total reimbursable amount (integer USD) to `/sdcard/Documents/AOHP/answers/ir_reimbursement_may_total.txt`.

4. Please find and compile the Board Review arrangement in the given apps—the meeting start time and duration from the Calendar app, and the meeting room from colleague Carl's latest SMS in the Messages app. Write one line to `/sdcard/Documents/AOHP/answers/ir_board_meeting_confirm.txt` as `start_time;room_name` (time format MM-DD HH:MM).

5. Please find and compile the shipment reminder in the given apps—the product name and order number for the pending shipment from the Mia Gift Order note in Notes, and Mia's phone number in Contacts. Write one line to `/sdcard/Documents/AOHP/answers/ir_order_mia_shipment.txt` as `product_name;order_number;phone_number` (phone_number digits only).

Category 5 — Memory Management

1. Open Calendar and create Development Meeting on the 15th of next month (11:15–13:30, description AOHP Development Meeting) and add Task AOHP Task under the 30th of next month (description Fork AOSP in aohp-os organization).

Question: What is the end time of the Development Meeting on the 15th of next month? What is the title of the Task on the 30th of next month? Answer in two lines.

2. From the HR SMS, add contact Alice with mobile and extension.

Question: What are the last four digits of Alice's mobile number?

3. In Markor, merge `/sdcard/Documents/AOHP/benchmark/projects/archive/2024/q1/notepad.md`, `q2/draft.md`, and `q3/minutes.md` in that order into `/sdcard/Documents/AOHP/benchmark/projects/archive/2024/sprint_summary.md` with a blank line between each file, then save.

Question: What is the full filename of the third source file you merged?

4. In Notes, open the note AOHP Backlog. Based on the todo items listed in the AOHP Backlog note, create a Checklist named AOHP TODO in the Notes app.

Question: What is the second todo item in the AOHP Backlog note?

5. In Markor, open `/sdcard/Documents/AOHP/benchmark/projects/archive/2024/q4/test.md` by navigating from the Markor root.

Question: How many folder levels did you open from the Markor root to reach that file? Reply with a single number.

Category 6 — Hybrid (Capability Composition)

1. Change the next Development Meeting in Calendar to 2 hours long, then set an alarm in Clock 15 minutes before that meeting's start time.

Continued on next page

Table 3 — *continued from previous page*

2. Text Carl asking which room Board Review is in. While waiting, open Calendar and change the Board Review on the 20th of next month start time to 13:00. When his reply notification appears, set the event location to the room name from the reply.

3. A colleague will SMS a meeting time. After the notification, check Calendar—if free, reply OK and schedule a 1-hour event; if busy, reply "Not available in this time slot." to the inviter and text the conflicting meeting organizer Sorry, I can't attend the meeting.

4. In Notes checklist AOHP Cleanup, mark Archive Phoenix prototypes as done, then delete all Phoenix-related files from the Download folder.

5. Family dinner this Saturday. Bob will send a menu SMS shortly; Notes Family Dinner lists drinks to prepare. Text Alice the wanted dishes and drinks in the format `dish1,dish2,...;drink1,drink2,...`