Mobile-MCP: Implementing the Model Context Protocol via Android's Inter-Application Communication Mechanisms

Xiheng Li
Beijing University of Posts and Telecommunications
Beijing, China
xhl0724@gmail.com

Mengting He
The Pennsylvania State University
State College, PA, USA
mvh6224@psu.edu

Chengcheng Wan
East China Normal University
Shanghai, China
ccwan@sei.ecnu.edu.cn

Linhai Song
SKLP, ICT, Chinese Academy of Sciences
Beijing, China
songlinhai@ict.ac.cn

Abstract

We argue that mobile AI assistants require a principled and secure interface for interacting with third-party applications. We present Mobile-MCP, an Android-native realization of the Model Context Protocol that enables applications to expose fine-grained, discoverable capabilities directly to large language models. By leveraging Android's Intent mechanism, Mobile-MCP decouples AI assistants from tool implementations, allowing tools to evolve independently while preserving security boundaries. This work points toward a scalable ecosystem for building flexible and safe mobile AI assistants, moving beyond manually engineered integrations and unconstrained, GUI-based approaches.

1 Introduction

Mobile phones have become indispensable in modern daily life, supporting a wide range of activities such as entertainment [7, 15], communication [20, 27], office work [11, 14], and personal organization [3, 13]. With the advancement of large language models (LLMs), LLM-based AI assistants (*e.g.*, Apple Intelligence [2], Huawei XiaoYi [8]) have emerged as a key differentiator and selling point for smartphones. These assistants promise to help users accomplish complex tasks more efficiently through natural language interaction, thereby significantly enhancing the overall user experience.

Despite their rapid adoption, current architectures for mobile AI assistants remain fundamentally limited. Existing solutions generally fall into two categories. In the first category, AI assistants rely on coordinated APIs to interact with applications or tools [1, 2, 6, 21]. In many cases, both the assistants and their associated tools must be developed within the same company (*e.g.*, Qwen Chat [1]). This tightly coupled design significantly limits extensibility and hinders interoperability among AI assistants and tools from different developers. In the second category, AI assistants operate by observing the graphical user interface (GUI), often through screenshots, and relying on LLMs to infer appropriate actions, such as clicks or text input [5, 17–19, 25, 26]. These actions are then executed via OS-level accessibility and automation interfaces. Although more general, this approach effectively grants AI assistants the same privileges as end users, raising substantial security and privacy concerns. Moreover, it enables assistants to access all data exposed by the tools, potentially undermining the tools' commercial interests.

We argue that neither approach offers a principled, secure, and scalable foundation for building mobile AI assistants. Instead, there should be a well-defined protocol between LLM-based assistants and mobile applications. Such a protocol would allow applications to expose selected functionalities in a controlled manner, while enabling AI assistants to automatically understand and invoke these functionalities, without requiring custom development.

The Model Context Protocol (MCP) [12] is designed for this purpose, providing a standardized mechanism for tools to expose their functionalities and for LLMs to reason about and invoke them. However, MCP is primarily tailored to networked services and server-based tools. To date, there is no practical solution for natively using MCP on smartphones, nor for enabling LLMs—whether running on-device or backed by cloud computation—to safely and effectively interact with mobile applications.

In this paper, we propose Mobile-MCP, a practical design and implementation of MCP for Android smartphones that leverages Android's existing inter-application invocation mechanism—the *Intent* framework. Specifically, Android applications can declare externally callable interfaces in their manifest files and provide textual descriptions that specify the functionality of each interface, as well as the semantics of its arguments and return values, analogous to how MCP servers expose services. LLM-based AI assistants can then query the Android package manager to discover all available MCP-style interfaces, interpret their descriptions, and supply appropriate parameters to invoke the corresponding APIs via the Intent mechanism, effectively mirroring how LLMs interact with standard MCP APIs.

Our approach eliminates the need for prior API coordination between AI assistants and tool applications. Instead, tools can be dynamically added to an AI assistant's capability set and evolve independently, without requiring any modifications to the assistant's code or internal prompts. Moreover, the approach allows each application to expose its capabilities in a fine-grained manner, enabling a clear distinction between interactions initiated by end users and those initiated by AI assistants. Together, these properties enable Mobile-MCP to support a new paradigm for building mobile AI assistants that can safely and flexibly interact with a broad ecosystem of third-party applications, rather than being limited to tightly integrated or manually engineered tools.

We have formally defined the data schema that governs how AI assistants communicate with tool applications, including how tools expose their APIs, how these APIs are discovered, how they are invoked, how requests are handled and responded to, and how

results are interpreted. The formal specification is available at [24]. In this paper, we focus on the implementation aspects, and discuss how Mobile-MCP compatible AI assistants and tools should be implemented. In addition, we have developed a working prototype of Mobile-MCP to demonstrate its practical feasibility.

Overall, this paper makes the following contributions:

- We propose Mobile-MCP, an Android-native realization of MCP that leverages the platform's existing inter-application invocation mechanisms. Our design includes a formally defined protocol schema and a working prototype.
- We show that existing operating system mechanisms provide a strong foundation for building AI assistants, and we encourage future work that further explores this design space.

The source code [23] and a demonstration video [22] of Mobile-MCP have been publicly released for reference.

2 Background and Related Work

This section presents the details of MCP and explains how it differs from Mobile-MCP. It also discusses how existing mobile AI assistants interact with tool applications.

2.1 Model Context Protocol (MCP)

MCP is designed to connect large language models (LLMs) with external tools and resources [12]. The core idea of MCP is that tools and resources, acting as MCP servers, describe their APIs in natural language. This allows LLMs, as MCP clients, to understand which APIs to use and how to invoke them correctly.

The protocol consists of two layers: a data layer and a transport layer. The data layer defines the message schema used for communication between MCP clients and MCP servers and the core primitives that servers and clients must provide. For instance, MCP servers must expose three types of primitives: tools, which LLMs can execute; resources, which LLMs can read to obtain contextual information; and prompts, which provide reusable prompt templates for interacting with LLMs. Moreover, each type of primitive supports several standard methods. For example, a client can call "`tools/list`" to retrieve all available tools and then invoke a selected one. The transport layer defines how communication channels are established and managed. MCP supports two main transport mechanisms: standard input/output streams for cross-process communication when clients and servers run on the same machine, and streamable HTTP transport when they run on different machines. MCP does not support automatic service discovery or API broadcasting. Clients must already know a server's process ID or network address in order to connect to the server.

Although Mobile-MCP shares the same basic idea as MCP, there are several key differences. First, MCP is designed for networked services, while Mobile-MCP focuses on enabling AI assistants to interact with mobile applications. Second, Mobile-MCP uses Android's Intent mechanism for its transport layer and supports API broadcasting, allowing an AI assistant to discover all Mobile-MCP compatible APIs on the same device. Third, Mobile-MCP is in its early stage and is simpler than MCP as it only supports tool primitives, but an AI assistant can still gain additional context from the descriptions of the associated applications.

2.2 Existing Mobile AI Assistants

Mobile AI assistants interact with users through voice commands or natural language text, with the goal to understand user intent and automate tasks on smartphones [4, 16]. They remove the need for users to switch between apps, navigate multiple screens, or perform repetitive GUI operations (*e.g.*, pressing buttons, typing text). They can further proactively perform tasks, such as replying to messages or sending personal reminders. A key difference between AI assistants and traditional chatbots is that AI assistants can directly operate mobile applications, which allows them to create impact on digital world or even real world. Overall, AI assistants can significantly improve user experience and have become a key driver for users' smartphone upgrade decisions [9, 10].

Both researchers and practitioners have developed multiple mobile AI assistants. How they interact with mobile applications can be broadly categorized into two types. In the first type, AI assistants interact with applications through coordinated APIs [1, 2, 6, 21]. Under this model, assistants can only communicate with tools developed by the same organization (*e.g.*, Qwen Chat [1]) or with third-party applications that explicitly implement the required API specifications (*e.g.*, Apple Intelligence [2], Google Gemini [6], Xiao AI [21]). For example, Apple Intelligence limits its integration to applications in 12 domains (*e.g.*, Photos, Mail). It supports a fixed set of actions within those domains, and third-party applications must conform to Apple-defined schemas to allow Apple Intelligence to perform the actions on them. This tightly coordinated approach constrains ecosystem extensibility. AI assistants cannot seamlessly integrate with arbitrary third-party applications, and the capabilities available to them are predefined rather than dynamically discoverable or extensible at runtime.

In the second type, AI assistants understand mobile screens or monitor changes on them through screenshots or self-defined representations, use multi-modal models to determine the next GUI operation, and then issue the operation to manipulate the application [5, 17–19, 25, 26]. This method is general and does not depend on applications' specific implementations. However, it has an important limitation: applications cannot distinguish whether an action is performed by a human user or by an AI assistant. As a result, the assistant is effectively given permission to perform any operation available to the user. If the AI assistant makes any error, it could potentially harm user experience or even cause financial loss.

Mobile-MCP introduces a new approach for enabling interaction between AI assistants and mobile applications. Unlike tightly coordinated API-based integrations, it defines an open protocol with only a few basic communication steps: capability registration, capability discovery, and capability invocation. The protocol does not specify how capabilities should be selected or how arguments should be generated; these decisions are left to AI assistants' models. This design enables AI assistants to connect to a broader ecosystem of third-party applications. Moreover, Mobile-MCP allows applications to selectively expose specific functionalities to AI assistants rather than granting full access. It also enables applications to distinguish between requests initiated by human users and those initiated by AI assistants. This distinction helps applications better protect user data and safeguard their commercial interests.

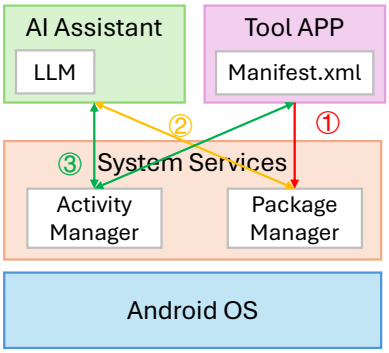

Figure 1: The workflow and architecture of Mobile-MCP.

Overall, Mobile-MCP addresses key limitations of existing communication mechanisms between AI assistants and mobile applications by improving openness, extensibility, and control.

3 Design of Mobile-MCP

Mobile-MCP can be utilized via widely adopted programming languages in coding Android applications, such as Java and Kotlin. Its transport layer is based on Android's Intent mechanism, and its data layer (message schema) is formally specified in [24].

As illustrated in Figure 1, applications participating in Mobile-MCP's workflow assume one of the two roles: (1) AI assistants that leverage local or remote LLMs to discover Mobile-MCP compatible, external APIs available on the smartphone, select appropriate APIs, invoke them with suitable parameters, and interpret the results; and (2) tool applications that expose MCP-style APIs and handle incoming requests from AI assistants. The workflow consists of three stages: Mobile-MCP API registration, Mobile-MCP API discovery, and Mobile-MCP API invocation. Although multiple AI assistants and tool applications may coexist on a single device, we illustrate Mobile-MCP using a single assistant and a single tool application for clarity in Figure 1.

① *API Registration.* Each tool application exposes its MCP-style APIs by declaring a service component using a `<service>` element in its *AndroidManifest.xml*. This element includes three metadata entries: the application's name, a textual description of its functionality, and the path to an XML file describing the application's Mobile-MCP APIs. A dedicated intent filter marks the service as Mobile-MCP-enabled, allowing AI assistants to discover it. The XML file lists all available Mobile-MCP APIs. For each API, it includes a unique identifier, a natural-language description, and detailed descriptions of the API's inputs and outputs. Each parameter is further annotated with its name, type, and semantic meaning. Upon installation, all declared Mobile-MCP APIs are automatically registered with the package manager via the *AndroidManifest.xml* in Figure 1.

② *API Discovery.* As illustrated in Figure 1, an AI assistant discovers available Mobile-MCP services by querying the package manager using the `queryIntentServices(Intent, int)` method of class `PackageManager`. The `Intent` is constructed to match the intent filter used to mark Mobile-MCP services, ensuring that only Mobile-MCP-enabled services are returned. For each service, the package manager provides the application's textual description, the list of exposed Mobile-MCP APIs, and the natural-language description of each API. Based on these descriptions and user intent, the AI assistant leverages an LLM to select an appropriate application, identify the relevant Mobile-MCP API, and determine suitable invocation parameters.

③ *API Invocation.* An AI assistant can invoke a Mobile-MCP API in two ways, depending on the interaction pattern. For repeated or stateful interactions, the assistant establishes a persistent binding to the tool application by invoking `bindService(Intent, ServiceConnection, int)` of class `Context`, where the `Intent` identifies the target Mobile-MCP service. Once bound, the assistant communicates with the tool via the `send(Message)` method of Android's `Messenger` abstraction, with the `Message` to transmit the identifier of the Mobile-MCP API and parameter values encoded as JSON. For one-shot or infrequent interactions, the assistant instead invokes `startService(Intent)` of class `Context`, embedding the target tool, API identifier, and parameters directly in the `Intent`. As illustrated in Figure 1, in both cases the requests are processed by the activity manager, which enforces privilege checks and dispatches the requests to the tool application.

Demonstration. We implement a prototype system using a clock-in application as the tool. It exposes three Mobile-MCP APIs: clocking in for the current day, clocking in for a specified day, and querying clock-in records for a given day. These APIs respectively demonstrate Mobile-MCP calls without parameters, with parameters, and with both parameters and return values, covering the full range of Mobile-MCP invocation scenarios. The AI assistant leverages OpenAI's LLM via its public APIs. The demonstration video in [22] shows that Mobile-MCP successfully supports all API invocation patterns. We further evaluate the performance of each step and find that over 99% of the execution time is spent interacting with the LLM, indicating that API invocations via Mobile-MCP do not constitute a performance bottleneck for AI assistants built on top of it.

4 Discussion and Conclusion

In this paper, we present Mobile-MCP, an Android-native implementation of MCP that leverages Android's *Intent* framework for cross-application communication. We formally define the message schema used between AI assistants and tool applications. We also implement a prototype to demonstrate that our design is practical and feasible. In the future, we plan to extend this framework by integrating additional components that AI assistants commonly require, such as long-term memory support and on-device LLM inference. We also plan to explore how to implement the A2A protocol using the same cross-application invocation mechanism.

More broadly, this work demonstrates the value of reusing and extending existing operating system mechanisms when designing new types of applications, such as AI assistants. In particular, we aim to investigate whether Android's existing privilege and permission system, especially features associated with the Intent framework, is sufficient to manage and restrict AI assistant behavior, or whether new abstractions and system mechanisms are necessary.

References

[1] Alibaba Cloud. Qwen, 2026. https://qwen.ai/home.

[2] Apple Inc. Apple intelligence, 2026. https://www.apple.com/apple-intelligence/.
[3] Atlassian. Trello, 2024. https://trello.com/.
[4] Cem Dilmegani. Mobile ai agents tested across 65 real-world tasks, 2026. https://aimultiple.com/mobile-ai-agent.
[5] Droidrun GmbH. Droidrun, 2026. https://droidrun.ai/.
[6] Google. Discover what google assistant is, 2026. https://assistant.google.com/.
[7] GooglePlay. Youtube, 2026. https://play.google.com/store/apps/details?id=com.google.android.youtube&hl=en_US.
[8] Huawei Corporation. Harmonyos 5, 2026. https://en.wikipedia.org/wiki/HarmonyOS_5.
[9] Hyunjoo Jin. Exclusive: Samsung to double ai mobile devices to 800 million units this year, 2026. https://www.reuters.com/world/china/samsung-double-mobile-devices-powered-by-googles-gemini-800-mln-units-this-year-2026-01-05/.
[10] Joan Faus. China's oppo sees ai driving demand, not worried about a bubble, 2025. https://www.reuters.com/world/china/chinas-oppo-sees-ai-driving-demand-not-worried-about-bubble-2025-10-28/.
[11] Microsoft. Outlook for ios and android, 2026. https://www.microsoft.com/en-us/microsoft-365/outlook-mobile-for-android-and-ios.
[12] Model Context Protocol Project. Model context protocol, 2026. https://modelcontextprotocol.io/docs/getting-started/intro.
[13] mylifeorganized.net. Mylifeorganized, 2026. https://www.mylifeorganized.net/.
[14] Slack Technologies, LLC. Slack, 2026. https://slack.com/downloads/android.
[15] TikTok. Tiktok, 2026. https://www.tiktok.com/download.
[16] Tom Sire. Beyond the app: How mobile ai assistants are redefining user experience, 2025. https://www.pulsion.co.uk/blog/beyond-the-app-how-mobile-ai-assistants-are-redefining-user-experience/.
[17] unimobile Contributors. Zhixing: A mobile agent development framework, 2026. https://github.com/apbocldueo/unimobile.
[18] H. Wen, Y. Li, G. Liu, S. Zhao, T. Yu, T. J.-J. Li, S. Jiang, Y. Liu, Y. Zhang, and Y. Liu. Autodroid: Llm-powered task automation in android. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking (MobiCom '24)*, Washington D.C., DC, USA, 2024.
[19] H. Wen, S. Tian, B. Pavlov, W. Du, Y. Li, G. Chang, S. Zhao, J. Liu, Y. Liu, Y.-Q. Zhang, and Y. Li. Autodroid-v2: Boosting slm-based gui agents via code generation. In *Proceedings of the 23rd Annual International Conference on Mobile Systems, Applications and Services (MobiSys '25)*, Anaheim, California, US, 2025.
[20] WhatsApp LLC. Whatsapp, 2026. https://www.whatsapp.com/.
[21] Xiaomi. Xiaomi HyperOS, 2026. https://www.mi.com/global/hyperos/.
[22] Xiheng Li, Mengting He, Chengcheng Wan and Linhai Song. The demonstration video of Mobile-MCP, 2026. https://youtu.be/Bc2LG3sR1NY.
[23] Xiheng Li, Mengting He, Chengcheng Wan and Linhai Song. Mobile-MCP, 2026. https://github.com/system-pclub/mobile-mcp.
[24] Xiheng Li, Mengting He, Chengcheng Wan and Linhai Song. Mobile-mcp specification, 2026. https://github.com/system-pclub/mobile-mcp/blob/main/spec/mobile-mcp_spec_v1.md.
[25] J. Ye, X. Zhang, H. Xu, H. Liu, J. Wang, Z. Zhu, Z. Zheng, F. Gao, J. Cao, Z. Lu, J. Liao, Q. Zheng, F. Huang, J. Zhou, and M. Yan. Mobile-agent-v3: Fundamental agents for gui automation, 2025.
[26] C. Zhang, Z. Yang, J. Liu, Y. Li, Y. Han, X. Chen, Z. Huang, B. Fu, and G. Yu. Appagent: Multimodal agents as smartphone users. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25)*, Yokohama, Japan, 2025.
[27] Zoom Communications, Inc. Zoom meetings, 2026. https://www.zoom.com/en/products/virtual-meetings/.