PicoClaw Mobile Wrapper Architecture

A scholarly account of the iOS and Android agent interfaces, use cases, and release implications

PicoClaw Product and Engineering

August 2026

Release framing. This paper describes the product architecture as an implementation and design snapshot. It distinguishes the mobile interface, the PicoClaw Go core, provider services, and platform-constrained automation. It is not a claim that every proposed autonomy feature is available on every device.

Abstract

PicoClaw is a lightweight agent system whose core is written in Go and whose value lies in coordinating language-model reasoning, session context, tools, and user approvals. This paper examines the product wrappers created for iOS and Android: native, deliberately simple conversational interfaces that turn that core into a device-appropriate agent experience. The wrappers share a provider-neutral conversation contract, durable session memory, credential isolation, and visible control boundaries. They diverge where the operating systems diverge. Android can embed a Go mobile runtime and expose narrowly authorized device capabilities through Android system APIs. iOS is best treated as a native Swift client with direct provider chat for foreground work and a paired Mac harness for richer agent execution. The analysis argues that the wrapper is not cosmetic packaging; it is the control plane that makes an agent understandable, governable, and releasable. Its principal implications concern privacy, accessibility, human authority, platform policy, and the distinction between a chat response and an executed action.

Keywords: mobile agents; human-in-the-loop automation; Go mobile runtime; iOS; Android; session memory; language-model providers; accessibility.

1 Introduction

Mobile AI products are often reduced to a text field over a model API. That simplification obscures the work required to make a useful agent: it must retain the relevant conversational state, identify the provider and permissions in force, present action requests before they occur, and report evidence of what happened. PicoClaw approaches the problem from the opposite direction. Its Go core supplies an agent loop, model-provider routing, sessions, tools, skills, channels, scheduled work, and event handling. The iOS and Android applications are wrappers around that capability, not separate model products.

The product objective is an elegant, low-friction interface: a readable conversation, an obvious composer, clear provider status, a visible session boundary, and an action record. The wrapper must also avoid a common but consequential product error: presenting a conversational answer as if it had completed an external task. A response, a tool request, a permission grant, a completed action, and a failed action are separate product states.

2 System Boundary and Shared Contract

The mobile system has four layers. First, the *presentation layer* is a native iOS or Android application with a chat surface, session controls, provider selection, settings, and run history. Second, the *agent layer* normalizes conversation turns, tool calls, approvals, cancellation, and streaming events. Third, the *PicoClaw core* supplies the agent loop, session history and compression, provider adapters, skills, and tool orchestration. Fourth, the *execution layer* is either a remote model provider, a paired Mac harness, or a platform-approved device capability.

Native mobile interface ⟷ canonical session and event contract ⟷ PicoClaw agent core ⟷ provider / approved tool host

iOS: Swift foreground client or paired Mac harness | Android: Kotlin/Compose plus embedded Go runtime

The canonical contract is intentionally provider-independent. A message has a role, content, attachments where permitted, time, and an immutable session identifier. A run emits typed events such as `assistant_delta`, `tool_proposal`, `approval_required`, `tool_result`, `run_completed`, and `run_failed`. This makes GPT, Gemini, xAI, and future providers interchangeable at the interface boundary without pretending that their native APIs are identical. The application owns rendering and user consent; adapters own request formatting and response normalization.

3 Session Memory as Product Infrastructure

Conversation history is a functional requirement, not a provider feature. A direct call that sends only the latest prompt has no usable memory even if the chosen model advertises a large context window. PicoClaw's session layer addresses this by retaining a stable session identifier, appending user, assistant, and tool records in order, and retrieving a bounded context for each subsequent turn. When the history exceeds a selected budget, compression or summarization produces a smaller durable representation while preserving the decisions, facts, and unresolved work needed for continuation.

This design has three consequences. First, a Gemini request, a GPT request, and an xAI request can each receive the same canonical session transformed by their adapter. Second, "New chat" and "Clear memory" are meaningful controls: they create or erase a local session rather than merely clearing the visible message list. Third, credentials remain separate from content history. Provider tokens are held in Keychain on iOS or Keystore-backed storage on Android; session records are stored in protected app-private storage and never treated as an API-key container.

Longer-lived memory should be consented, inspectable, and optional. A release-ready implementation distinguishes (a) the active session, (b) a user-visible saved preference or fact, and (c) transient provider-side context. The first is necessary for continuity; the second needs a review and deletion surface; the third is not a substitute for local memory and must be described honestly.

4 iOS Wrapper: Native, Foreground, and Paired When Needed

The iOS application is a native Swift/Xcode product tested through the iOS Simulator and distributed through TestFlight. Its direct chat path uses a native network client for foreground provider requests, a keyboard-safe multi-line composer, streaming output, and a protected local session. This is the appropriate shape for asking, drafting, analyzing, and continuing a conversation on an iPhone.

For rich tool execution, iOS has material platform limits. The platform does not offer a general-purpose, third-party facility for silently driving other applications. Screen broadcast is mediated by ReplayKit, and the capability should be started by the user through the approved broadcast flow rather than represented as unrestricted device control [1]. The product architecture therefore places advanced agent execution on a paired, user-controlled Mac harness. The iPhone shows pairing state, session state, approvals, and run receipts; the paired harness owns the agent loop and any authorized desktop tools.

This division is valuable rather than deficient. It keeps the phone interface simple, preserves the iOS security model, and avoids misleading claims that a mobile chat wrapper can perform arbitrary background actions. A user may connect with an API key for direct provider chat or pair to an owned harness for subscription-backed or tool-enabled work. A third-party iOS application must not ask the user to supply a Codex CLI device code: that code grants access to a separate CLI authorization flow, not to a generic mobile app. The wrapper's safe authentication posture is user-provided provider credentials or an explicit, secure pairing mechanism.

5 Android Wrapper: Embedded Runtime with Bounded Device Capability

The Android application uses Kotlin and Jetpack Compose for the interface and is being assembled around a Go mobile binding for the PicoClaw runtime. This is the more direct route to a self-contained mobile agent: Compose renders the conversation and approval sheet; a Kotlin runtime bridges to a gomobile artifact; the Go core performs the session and agent work; and a Kotlin tool host invokes only Android capabilities the user has authorized.

Android offers more feasible on-device integration points, but its permission model remains central to the product. An accessibility service is a user-enabled system service intended to assist users with disabilities; Android's documentation expressly frames it as an accessibility mechanism and requires careful declaration and user activation [2, 3]. MediaProjection similarly requires user-mediated consent for screen capture. Consequently, device tools are capability-scoped: the app must state the intended action, request the relevant system permission, surface an approval at the

agent boundary, and report a receipt. The architecture never treats accessibility or screen access as a hidden universal automation channel.

Dimension	iOS wrapper	Android wrapper
Native layer	SwiftUI/Xcode, foreground-native chat	Kotlin/Compose, native chat and controls
Core connection	Direct provider chat; paired Mac harness for advanced execution	Embedded Go mobile runtime plus Kotlin tool host
Session memory	Protected local store; shared session semantics	Protected app-private store; shared Go session semantics
Device action model	Explicitly paired / user-mediated platform flows	Capability-scoped tools with system permissions and approvals
Release implication	Strong sandbox alignment; no claim of arbitrary iOS control	Greater local agency, with heightened accessibility and disclosure duties

Table 1: Equivalent product promises, different platform implementations.

6 Use Cases

The wrapper architecture supports several concrete product classes.

1. **Persistent personal work sessions.** A user can ask a model to draft, plan, compare, or explain across multiple turns while the actual session state, not merely the visible chat, carries forward.

2. **Field and accessibility assistance.** On Android, a consented tool host can support guided workflows whose benefit comes from the combination of model reasoning and approved accessibility-oriented interaction. The product must remain usable without device automation and never make accessibility access a silent prerequisite.

3. **Paired desktop orchestration.** An iPhone can be the portable command and approval surface for work whose tools execute on a user-controlled Mac. This supports richer workflows while preserving a comprehensible trust boundary.

4. **Multi-provider resilience.** A user may choose a provider based on capability, latency, privacy terms, or cost while retaining the same local conversation and approval model. Provider switching is a routing decision, not loss of the product's identity.

5. **Auditable delegated work.** Runs can record what the assistant proposed, what the user approved, which tool ran, and what result was returned. This is useful for operational tasks where a fluent answer is insufficient evidence of completion.

7 Release Implications

The central release implication is that trust must be visible in the product. A release should expose provider identity, account or pairing state, current session identity, memory controls, pending approvals, and final run status. It should never represent a network response as an action receipt. On-device credentials require platform-native protection; local session content should have deletion behavior that matches its user-facing label; and logs intended for support must be scrubbed of secrets and sensitive message content.

Provider policy and commercial terms also matter. A ChatGPT subscription and API billing are separate arrangements; a mobile wrapper should not imply that a consumer ChatGPT login automatically authorizes API use [4]. Where a paired Mac uses a local Codex app-server-style interface, it should be reachable only through an authenticated pairing channel and not exposed as a public endpoint [5]. For APIs, release operations should use a separate project key, restrictive spend limits, observability, and rotation procedures.

The Android product requires a particularly explicit disclosure review. Any accessibility, notification, screen-capture, or package-visibility capability should be justified by a user-facing feature, invoked only after context-specific approval, and tested against the relevant Play policies. The iOS product requires a parallel review of entitlement, background behavior, and ReplayKit disclosures. In both cases, the simplest release path is to ship excellent chat, durable session memory, transparent provider settings, and a safe approval model before expanding automation scope.

8 Limitations and Research Questions

Several limitations remain inherent. Context compression can lose detail; therefore summaries should remain inspectable and be tested with task-continuation cases. Provider outputs vary in tool use, latency, and policy behavior; a normalized contract reduces interface inconsistency but cannot erase model differences. Mobile operating systems deliberately restrict cross-application control, so parity should mean equal clarity of intent and evidence, not identical privileges. Finally, durable memory raises governance questions: what is retained, when it expires, whether it can be exported, and how a user can correct it.

Future evaluation should measure more than message quality. Useful release metrics include successful continuation after a new turn, correct preservation of a tool result, clear-memory effectiveness, approval comprehension, task completion with a visible receipt, and permission-denial recovery. These metrics evaluate the wrapper as a human-agent system rather than a thin API client.

9 Conclusion

PicoClaw's mobile wrappers transform a capable Go agent core into a governed product surface. Their shared contract gives users consistent sessions, providers, approvals, and evidence. Their platform-specific designs honor the actual security models of iOS and Android: an iPhone remains a strong native conversational and paired-control client, while Android can host a more local agent runtime under explicit capability boundaries. The product opportunity is not unrestricted automation. It is trustworthy, understandable delegation: a mobile agent that remembers the right

context, asks before acting, and makes the outcome legible.

References

[1] Apple. (2026). *ReplayKit*. Apple Developer Documentation. https://developer.apple.com/documentation/replaykit

[2] Android Developers. (2026). *Build an accessibility service*. https://developer.android.com/guide/topics/ui/accessibility/service

[3] Android Developers. (2026). *AccessibilityService*. https://developer.android.com/reference/android/accessibilityservice/AccessibilityService.html

[4] OpenAI Help Center. (2026). *Is API usage included in ChatGPT subscriptions?* https://help.openai.com/en/articles/8156019-is-api-usage-included-in-chatgpt-subscriptions-even-if-i-have-a-paid-chatgpt-account

[5] OpenAI. (2026). *Codex app server*. https://learn.chatgpt.com/docs/app-server.md