Keva: Running a Real Coding Agent on an Unrooted Android Phone — A Systems-Engineering Case Study

Architecture, failure modes, and transferable engineering lessons

The Keva Project Team · v1.0 2026-09-16 · **v1.1 revised 2026-09-17** (see Appendix C)

Abstract

A modern flagship phone has the memory, CPU cores, and connectivity of a workstation from a few years ago. What it does not have is **permission**: mobile operating systems define an application as a consumer of UI surfaces, not as a host for long-lived processes that fork and exec, hold state for hours, and survive being backgrounded.

Keva is a Flutter Android application that bundles a complete glibc Linux userland (an Ubuntu rootfs) inside the APK and runs two real coding-agent runtimes — the Claude Code CLI and the Codex app-server — behind a single interface, inside an unrooted application sandbox. It is a working coding agent on a phone. This paper is a systems-engineering case study on what that actually cost.

The paper's central finding is that **the obstacles that taught us the most were not architectural**. They were failures of **accounting and representation** that only surface when the system is used by a real human: a completion ledger drained by whichever signal arrived first rather than the signal that *owned* the entry; a sub-agent death state that no layer could represent and therefore no layer could handle; an exclusivity guarantee held in memory by a process the OS is free to destroy at any moment. The architectural obstacles — embedding the runtime, surviving OEM background management, mediating permissions without ptrace — were predictable and were solved.

We report each failure, its fix, and what generalizes beyond phones; describe how the system is built and maintained under a verification discipline in which one human directs several models; and include a verification map so that every technical claim in this paper can be independently checked by the reader.

Keywords: on-device agents; on-device LLM; Android sandbox; process keep-alive; multi-runtime abstraction; failure case study

1 Introduction

1.1 A computer you are not allowed to program

Suppose a person owns only a phone, and wants an AI agent to help them edit code, read files, and run tests. What can they do?

Hardware is not the bottleneck. A 2026 flagship phone has 12–16 GB of RAM, eight CPU cores, and stable gigabit-class networking. That is more than enough to run a Node.js-based command-line agent.

The real obstacle is operating-system policy. Android confines every app inside a strict application sandbox: SELinux policy forbids executing arbitrary bundled binaries (`execute_no_trans` denies it); the app has no root, so no `mount` and no `ptrace`; once backgrounded, a process faces Doze mode, App Standby, and OEM-customized background cleaners; a process that holds CPU and memory for hours is treated as misbehaving and killed.

In other words, **the phone is a computer you are not allowed to program** — not because it lacks compute, but because the OS does not permit it.

1.2 Why the answer is not "run it in the cloud"

There are three honest alternatives, and each gives something up:

- **SSH into a server**: gives up offline capability and low latency, and introduces a new trust boundary — the data an agent most needs to touch (contacts, photos, local files, location) is on the phone, not on the server.
- **A cloud IDE / remote dev environment**: still network-dependent, and moves code and credentials to a third party.
- **A cloud agent with the phone as a thin client**: besides latency and cost, the fundamental problem is **data gravity** — the objects an agent operates on live on the device.

Keva's position: if a user's only computer is their phone, then the agent should run on the phone. Not the phone as a terminal — the phone as a **host**.

1.3 What Keva is

In one sentence: **Keva is an Android application that runs a real coding agent (the Claude Code CLI and the Codex app-server) on an unrooted phone, letting the user read code, edit files, run tests, and commit work from a five-inch screen.**

It bundles an entire Linux toolchain (an Ubuntu rootfs, Node.js, Python, and two agent CLIs) into the APK (746 MB for release 2026.38.5); after install this is extracted locally on the device and run as child processes. It depends on no cloud orchestration — model inference hits LLM-provider APIs, but **the agent loop itself runs on the phone**.

1.4 Contributions

The contribution of this paper is a case study, not a benchmark. Specifically:

1. **Design and implementation**: a system that embeds a complete Linux userland inside an unrooted Android app sandbox and runs two real coding-agent runtimes (§4).
2. **Survival on a hostile OS**: a watchdog substrate and a cross-process orphan-verification mechanism (four checks, fail-closed) addressing the consistency problem "after the OS reclaims the app, the in-memory exclusivity guarantee is reconstructed empty while the orphan engine is still running" (§4.4).
3. **Three rules for absorbing capability asymmetry**: where a difference between two runtimes should be absorbed — three instances, three different answers (§4.6).
4. **Six real failure cases**, whose central lesson is that failures of accounting, representation, and observability are more damaging than architectural failures, and only surface in real use (§5).
5. **A verification discipline for human-directed multi-model development**: three reusable practices and two checkable ratios (§8).
6. **A verification map**: every technical claim in this paper maps to a concrete file path and line, so the reader can check any of it without asking anyone (Appendix A).

1.5 How to read this paper

- **Evaluators** (is this thing sound?): → §2 (capabilities), §7 (does it hold up), §9 (limitations)

- **Contributors / researchers** (how is it maintained, what can I cite?): → §8 (methodology), Appendix A (verification map)
- **Curious about the problem**: → §3 (background), §4 (design)
- **Here for the failures**: → §5

2 What Works Today

This section deliberately precedes the design chapter. A reader deciding whether to invest attention should not be forced through six pages of architecture first.

2.1 Capability inventory

Works:
- Two real agent runtimes (Claude Code CLI, Codex app-server) switchable behind one interface
- File I/O, directory browsing, and shell command execution (inside the app-sandbox Linux userland)
- Multi-turn conversation, sub-agent dispatch, tool invocation and result rendering
- Web search, code-diff display, structured work logs (worklog)
- Local database persistence for sessions and configuration; multiple LLM providers (Anthropic + OpenAI-compatible endpoints)

Does not work / limited:
- Anything requiring root (`mount`, writing system partitions)
- System-level permission mediation requiring ptrace (see §4.5 for the application-layer substitute and its weaknesses)
- Long background CPU tasks remain threatened by OEM power-management policies (§9)

2.2 Known-broken list, up front

- The single controller file `lib/modules/chat/chat_controller.dart` has grown past **4,000 lines** — the system's largest maintainability debt (§9)
- The network layer uses no mature `http`/`dio` library; it is a self-built `LocalDnsServer.kt` + proxy. Since v1.1 the app's own HTTP clients share the CLI's proxy discovery (§5.5), but the layer remains hand-built (§9)
- The battery-optimization exemption is **optional**; if not granted, OEM background management may intervene during long runs (§9)
- This paper performs no benchmark-style evaluation (§7 states why, and what is offered instead)

3 Background and Motivation

3.1 The true shape of the problem

Splitting "run a coding agent on a phone" apart yields three very different categories of subproblem:

Category	Example	Nature
Performance	Enough RAM? Enough CPU?	**Not the problem.** A 2026 flagship is sufficient
Permission	SELinux forbids exec; no root; no ptrace	**Hard constraint, but engineerable** (§4.2, §4.5)
Lifecycle	Background kills, process reclaim, cross-process state inconsistency	**The genuinely hard part** (§4.4, §5)

This taxonomy is itself the first transferable conclusion: **when someone tells you something is "impossible on a phone," ask which category they mean.** Most "impossibilities" are permission-class, and permission-class problems usually have exploitable slack (§4.7).

3.2 An underrated fact: the agent loop on-device

Mainstream "phone AI assistants" are **remote procedure calls plus local UI**: the phone sends user input to the cloud, the cloud runs the agent loop, and rendering instructions stream back. Under that architecture the phone is a terminal.

Keva's architecture is **local agent loop plus remote inference**: the agent loop (tool scheduling, file operations, context management, sub-agent dispatch) runs locally on the phone; only LLM inference hits an API. This is not "offline-first" dogma — it is a factual judgment: **an agent that touches files should live where the files are.**

4 System Design

4.1 Overall architecture

Keva has an application layer (Flutter/Dart) and a platform layer (Android Kotlin).

Dart side (`lib/`) contains four top-level directories:

Layer	Files	Lines	Responsibility
`ui/`	47	35,384	Pages, widgets, rendering (incl. Command Transparency cards)
`core/`	120	31,511	Runtime abstraction, database, bridge, diagnostics, contracts
`modules/`	50	18,590	Feature modules: chat, entitlement, auth, …
`l10n/`	3	13,974	Internationalization (generated)

Within `core/` , the larger subdirectories: `contracts` 6,749; `database` 4,669; `codex` 4,614; `diagnostics` 4,557; `runtime` 4,115; `bridge` 2,970; `engine_pack` 2,278; `handoff` 412 lines. Dependency injection uses Riverpod (`flutter_riverpod ^2.6.1`), but it lives inside `core/providers/` — it is not a separate top-level architectural layer.

Kotlin side (`android/app/src/main/kotlin/com/mcagent/mcagent/`):

Module	Responsibility
`ProcessManager.kt`	Construction and lifecycle of the Claude CLI child process
`CodexAppServerProcessManager.kt`	Codex app-server start/stop, stdio forwarding, generation staleness guard, shared `EngineMutex`
`InstallManager.kt`	Install layout for rootfs / Node / both CLIs (`files/mcagent-runtime/`)
`McagentKeepAliveService.kt`	Foreground service + `PARTIAL_WAKE_LOCK` + `START_STICKY`
`EngineOrphanGuard.kt`	Cross-restart engine exclusivity and pre-kill verification
`StorageCleanManager.kt`	Storage accounting/cleanup; default-deny allowlist

Dependency choices (`pubspec.yaml`): state management `flutter_riverpod` ; database `drift ^2.24.0` + `sqlite3_flutter_libs` (drift falls back to an in-memory database on disk failure while keeping a single shared instance); secrets `flutter_secure_storage ^9.2.4` ; FFI `ffi ^2.2.0` ; asset unpacking `archive ^4.0.9` ; image compression `image ^4.5.4` ; location `geolocator ^13.0.2` (foreground only, no background tracking).

Worth noting: **the network layer uses neither `http` nor `dio`** , but a self-built `LocalDnsServer.kt` and proxy layer. This is a deliberate engineering choice (precise control over traffic shaping and DNS behavior), and it is also a maintenance debt (§9).

4.2 Embedding a Linux userland inside an unrooted app

The core idea: bundle a complete Ubuntu rootfs (glibc, Node.js, Python, both agent CLIs) as APK assets, extract it into the app's private directory on install, and then **execute its binaries inside the application sandbox, running as the app's own uid**.

One implementation detail deserves correction here. The original plan was to use PRoot (a ptrace-based userspace `chroot` substitute) to provide the filesystem view. But `ProcessManager.kt` documents (:30-32):

> Instead of PRoot (which conflicts with Bun's JIT in Claude Code v2.x), this launches Claude Code directly using the Ubuntu rootfs's dynamic linker (`ld-linux-aarch64.so.1`).

That is: **the Claude Code CLI does not actually run under PRoot**, because PRoot's ptrace interception conflicts with Bun (Claude Code's JS runtime) JIT compilation. What actually happens is **direct launch bridged through the rootfs's dynamic linker**: `ld-linux-aarch64.so.1` is used as the program entry point and the rootfs library paths are injected via `LD_LIBRARY_PATH` , executing binaries without any `chroot` or `ptrace` at all. (The `buildProotCommand` name is historical; it actually emits the ld-linux bridged command and checks preconditions: missing linker → `linker_missing` , missing ptrace helper → `supervisor_missing` , helper not executable → `supervisor_not_executable` — see `ProcessManagerProotTest.kt` .)

Why a bundled runtime rather than a first-launch download: a deliberate product decision. The full package is about 750 MB, of which runtime assets are the overwhelming majority. The cost is size; what it buys is: works immediately after install, works offline, and eliminates the failure surface of "first-launch

download failed." Bundling dependencies instead of fetching them at runtime is a classic availability-for-size trade in systems engineering.

4.3 One interface, two runtimes

Keva abstracts a single `AgentRuntime` seam between the two agent runtimes (`lib/core/runtime/`):

```
lib/core/runtime/
├── agent_runtime.dart              # the seam
├── runtime_capabilities.dart       # declared capabilities (heartbeat, streaming granularity…)
├── runtime_coordinator.dart        # coordinator interface
├── runtime_coordinator_impl.dart   # implementation
├── coordinator_state.dart
├── runtime_identity.dart
├── runtime_auth_adapter.dart
├── runtime_configurable.dart
├── agent_event.dart                # unified event model
├── agent_runtime_status.dart
├── agent_runtime_diagnostic.dart
└── adapters/
    ├── claude_cli_agent_runtime.dart      # Claude adapter
    ├── claude_event_mapper.dart           # Claude event mapping
    └── codex_app_server_agent_runtime.dart # Codex adapter

lib/core/codex/
└── codex_event_mapper.dart             # Codex event mapping (lives with the app-server client)
```

Two things forced this seam: **frequent vendor model-identifier churn** (see §4.8, hydration on read), and **capability differences between the two runtimes** (see §4.6). What it bought: onboarding the second engine cost far less than the first — one new adapter pair (runtime + event mapper) rather than another full-stack rewrite.

4.4 Staying alive on a hostile OS

This is the hardest technical part of the paper. Keva maintains a **watchdog substrate**, all defined in `lib/modules/chat/chat_controller.dart` :

Mechanism	Constant	Location	Role
Hard turn timeout	`Duration(minutes: 720)` = 12 h	`:385`	Absolute ceiling on one turn
First-event watchdog	`Duration(seconds: 180)`	`:390`	If no semantic event arrives within 180 s of turn start, declare it stuck. The comment records the calibration: across 16 successful GLM turns, median time to first semantic event 6.1 s, tail 65.5 s — 180 s is ~2.7× the tail
Runtime-instance watchdog	`Duration(seconds: 15)`	`:391`	Liveness of the runtime instance itself
Liveness gap	`Duration(minutes: 3)`	`:392`	Long no-heartbeat interval
Idle reclaim	`Duration(minutes: 5)`	`:394`	Reclaim the resident process after idleness

The cross-process half of the story. Engines are started under `setsid` (becoming session leaders), so **they outlive the app process**. This creates a subtle consistency problem: if the OS reclaims the app, the app's in-memory "engine exclusivity guarantee" is reconstructed empty, while the orphan engine process **is still running**. On next start, the fresh app instance must decide whether to clean it up.

`EngineOrphanGuard.kt`'s answer is **verify before killing, fail-closed**. `verify()` (`:104–113`) requires all four to pass before a kill is permitted:

1. `isAlive` (`:105`) — the process still exists
2. **starttime** from `/proc/<pid>/stat` (`:106–107`) — a process's "incarnation" is immutable; PID reuse necessarily changes this value
3. **cmdline** (`:108–109`) — immutable after `exec`
4. **owner uid** of the `/proc/<pid>` directory (`:110–111`) — must equal the app's uid

If any is unreadable or mismatched, `verify()` returns `false` — **no kill, and no error** (`:28–29`). `reclaimOrphan()` (`:96–101`) returns a "not ours" result (`fullyReaped=true`) when verification fails, rather than killing anyway. Torn or malformed record files are parsed as "no record" (`:31–32`, `:125–135`) — no guessing.

`AndroidProcessProbe` (`:186–224`) does the parsing: `starttime` is field 22 of `/proc/<pid>/stat`, and it is sliced **from after the last `)`** to survive spaces inside the `comm` field (`:199–209`); the uid comes from `Os.stat` (`:221–223`).

The design principle is one comment in `EngineOrphanGuard.kt`:

> The guard would rather miss an orphan than kill a stranger.

The process-group behavior itself is covered by instrumented tests (`ProcessGroupAndChildrenTest.kt`, F6/F7): they verify the process becomes its own group leader after `setsid` (`pgid == pid`), and confirm the platform quirk that on some ROMs `setsid` is a toybox/busybox applet that calls `setsid(2)` in-process and **does not fork**.

4.5 Permission mediation without ptrace

A real desktop agent (using LD_PRELOAD or seccomp, say) relies on syscall interception to gate tool execution before it happens. The Android application sandbox **offers no syscall interception**.

Keva's substitute is an **application-layer pipeline**: every tool invocation must pass through a mediation layer in the app process, which performs permission checks before execution. This is weaker than real ptrace — a path that bypasses the mediator and talks to the engine process directly could in principle escape gating. **We state this weakness plainly** rather than dressing it up as equivalent to a system-level sandbox. In practice the engine is a child process started by the app with its stdio hosted by the app, so "bypassing the mediator" requires modifying the engine itself — not impossible, but a far higher bar than misuse.

4.6 Where to absorb a capability asymmetry

This is the paper's central design idea. Two runtimes with different capabilities force a choice of layer. Three instances, three different answers:

Asymmetry	Absorbed at	Rule	Code
One emits heartbeats, one never does	**Declared capability**	Gate on declared capability, **never on runtime kind**	`runtime_capabilities.dart:24–31`; `chat_controller.dart:3258`
One streams whole blocks, one per token	**Presentation**	Absorb at the **highest layer** where both still look identical	`live_text_pacer.dart:3–9`
Handoff context resolved at different times	**Delivery contract**	Move the seam, **don't special-case call sites**	`chat_controller.dart:2845–2882`; `chat_page.dart:4099–4142`

Instance 1: heartbeats. `runtime_capabilities.dart:24–31` declares `emitsHeartbeats`, default `false`, requiring explicit opt-in per runtime. The comment (`:26–30`, tag HB-1) explains: Codex does not emit heartbeats natively, and arming a heartbeat watchdog for it would **kill healthy turns**. So the consumer (`chat_controller.dart:3258–3260`) arms the liveness watchdog only when `capabilities.emitsHeartbeats` is true, and **re-checks the same flag** on the activity-reset path (`:3299`). While the process is still alive, deferral is capped at `_maxLivenessDeferrals = 3` (`:1665`, roughly a 12-minute grace window), see `:3345–3354`.

> Rule 1: **Gate on declared capability, never on runtime kind.** The moment you write `if (runtime is CodexRuntime)`, you have welded a capability assumption onto a kind check, and adding a third runtime will necessarily break it.

Instance 2: streaming granularity. The Claude CLI does not enable `--include-partial-messages` (M-0533, `chat_controller.dart:3334`, `agent_event.dart:314–315`), so it emits text in **whole blocks**; Codex streams **per token**. Rather than writing two renderers in the UI layer, Keva feeds both shapes into the same `LiveTextPacer` and makes **the reveal rate a pure function of current backlog** (`live_text_pacer.dart:6–9`). The result: the same total text, fed as one block or token-by-token, yields **the same reveal per tick**. The UI side is `chat_page.dart:5721–5722` (RT-2).

> Rule 2: **Absorb at the highest layer where both runtimes still look identical.** The presentation layer is that layer — absorbing there means lower layers need not know about the shape difference at all.

Instance 3: when handoff context is resolved. Two branches must be distinguished, because their timing genuinely differs:

- **Cross-runtime** (Codex ↔ Claude): the brief is assembled **at send time** (`chat_controller.dart:2845–2882`, `assembleForSession` at `:2857`), and it is built **entirely from the local database** (`handoff_brief_service.dart:8–13`) — deliberately without asking the old runtime, which may have crashed or hold stale credentials.
- **Same runtime** (Claude replay): the transcript replay prefix is resolved and stashed **at switch/restart/prewarm time** (`chat_page.dart:4099–4142`, taking the most recent 12 messages, capped at 12,000 characters), armed at `:1666–1669` (prewarm-resume failure fallback), `:2058–2061`, and `:8423`; at send time `chat_controller.dart:2876–2881` prepends it to the prompt.

> Rule 3: **Move the seam, don't special-case the call site.** The difference between the two branches is encapsulated in the "stash vs. assemble" contract; the send path only knows to "prepend whatever was stashed."

(For code readers: the comment at `chat_controller.dart:437–441` labels the handoff brief "send time (CTX-1)" — accurate for the cross-runtime branch, but the same-runtime replay branch is switch-time stashing plus send-time prepending. Keep the distinction in mind when reading.)

4.7 Constraint arbitrage: what `targetSdk = 28` bought

`android/app/build.gradle.kts:104` sets `targetSdk` to **28**, and the comments (`:98–103`, `:153`) state plainly that this is **intentional**:

> targetSdk 28: on Android 10+ (API 29+) the `untrusted_app` SELinux domain forbids `execute_no_trans` on `app_data_file`, so the app process cannot exec the downloaded glibc loader / claude.exe from filesDir (verified: avc denied execute_no_trans). Targeting <=28 places the app in the `untrusted_app_28` domain, which permits it. Same approach Termux uses to run downloaded native binaries without root.

This choice has an **incidental and crucial side benefit**: Android 15 caps `dataSync` and `mediaProcessing` foreground services at six hours per day, but only for apps targeting API 35 or later — an app targeting 28 sits outside that rule, which is part of what makes **multi-hour agent runs possible at all**. A cost accepted for reason A paying rent in an unrelated dimension B.

The same choice has a visible cost on the way in: Google Play Protect flags any sideloaded app targeting an old API level ("built for an older version of Android") and asks the user to confirm the install. That prompt appears on every GMS phone, so the download page explains it before the user sees it.

This trade has a shelf life: it depends on platform behavior, and future policy tightening by Google could end it. **We list it as a limitation (§9) rather than treating it as a permanent solution.**

4.8 Provider hydration on read

LLM vendors' model identifiers change often enough that **stored configuration goes stale**. Keva's approach (`lib/modules/config/config_service.dart`, semantics per M-1036): on every read, built-in providers' model aliases are **re-aligned** to the current built-in list and re-persisted — because model identifiers are owned by the app, one update reaches every installed instance. The sole exception is the user-owned `custom` provider (user-defined base URL and aliases are never overwritten).

> A small illustration of "data ownership determines migration strategy": **configuration the app owns, the app has a duty to keep fresh; configuration the user owns, the app has no right to rewrite.**

5 Failure Cases: Accounting and Representation

This is the most important section of the paper. All six cases are real, all surfaced only when a real human used the system on a real network, and **not one** is an architectural failure.

5.1 Ledger ownership: identity, not cardinality

Symptom (user report): *"The main process doesn't seem to know what to do with a dead sub-agent."*

Root cause. Sub-agents are tracked in a pending set (`Set<String> _pendingSubagentIds`, `chat_controller.dart:1604`). Completion signals arrive over **several channels**: some carry the spawning tool-use id, some do not. In the old implementation, the id-less path popped an *arbitrary* entry (FIFO pop). Because a sub-agent commonly reports **twice** — once with its id, once without — the second report took the slot of a sub-agent that was **still alive**. The ledger emptied early, the parent turn finalized, and the abandoned sub-agent's output was discarded as belonging to a finished turn.

Fix (M-1363 and related): the ledger now closes **by identity**.

- `onSubagentReturned` (`:2105–2118`): only a signal carrying the spawnId may remove **its own** entry; id-less signals (Claude's task-notification result, Codex's outer turn/completed) simply `return` at `:2116`. The comment at `:2110–2115` records that the old FIFO pop killed live sub-agents' slots.
- `onSubagentToolResult` (`:2137–2152`): exact-match removal; a backgrounded placeholder ack does not drain the set (`:2143–2146`).
- `drainFinishedBackgroundTasks` (`:2090–2103`): releases only when the live set is **empty as a whole**. The comment at `:2058–2082` records the old implementation entering and leaving the live set more than twenty times within 22 seconds, causing false convergence.

Three lessons:

1. **A counter is not a ledger.** A signal that cannot name the entry it closes must not close one.
2. **Failure was structurally unrepresentable** (see §5.2).
3. **Failure amplified, it did not cause.** A failing sub-agent reports **early**, while its siblings are still running — exactly when a live slot exists to steal. **The bug was present on successful runs too**; failure merely made it easier to see.

5.2 The unrepresentable state

Symptom. The user reported that after a sub-agent died the main process behaved oddly — yet no code anywhere could express "this sub-agent died."

Root cause. Both event mappers hard-coded a sub-agent's terminal state as `completed`. At the accounting layer, **no value meant "this one died."**

Fix. Terminal state is no longer hard-coded, and it is fail-closed:

- `codex_event_mapper.dart`, `_turnOutcome` (`:561–573`, M-0532): `completed → success`, `failed → failure`, `interrupted → interrupted`, `inProgress` and **any unknown value** → **`failure`** (`:557–560`).
- `claude_event_mapper.dart`, `_turnOutcome` (`:335–344`): `success → success`, `timeout → timeout`, everything else → `failure`.
- `claude_event_mapper.dart:266–306` (M-1372): the branch that previously special-cased task-notification results as non-terminal progress events has been **removed** — every result is now a normal `AgentTurnCompleted` carrying an `originKind`, and consumers gate on "is there still unfinished

work." The comment at `:272–274` records the old bug's user-visible symptom: *"the HTML got built but the turn never ended."*

- The honest fallback is preserved: a settle-timeout-driven finish **does not count as normal success** (`_onSubagentsAbandoned`, `:470–475`).

Lesson. You cannot handle a state you cannot represent. The user's report was literally correct — it was not that the main process "didn't know what to do," it was that there was **no word** for the thing at that layer. This is a type-system-level lesson: when a domain state has no corresponding value in the types, every piece of code that handles it is necessarily wrong.

5.3 The bug only a fresh device could show

Symptom. Completing onboarding via the subscription-login path left the composer rendering no model affordance at all.

Root cause. That path never persisted the active provider. Any **already-configured** device could not see it — its active provider had long since been written to disk.

Lesson. A new device is not a neutral device. It is a device with a particular history ("empty") that happens to hide the class of bug that only appears when some prior state exists.

5.4 The bug only a used device could show

Symptom. Reproduced every time on a daily driver; **never** on the development device.

Root cause. The chat surface had learned to yield the single engine slot before switching runtimes; the Settings and onboarding auth paths never had. The development device, whose data is frequently cleared, never has a warm engine — the slot is always free and the bug cannot occur. On a daily driver the engine is often warm, the slot is occupied, and the bug is hit every time.

Lesson (the mirror of §5.3): **a cleared device is not a neutral device.** It happens to hide the class of bug that depends on pre-existing state. Long-lived local state deserves the same adversarial scrutiny as untrusted input.

5.5 The failure nobody logged

Symptom (2026-09-15, the day before the first public build). On a phone whose VPN ran in proxy mode, the app never noticed a newly published update. The bundled CLI on the same phone reached the network without trouble. For half an hour the two observations coexisted with no error anywhere.

Root cause. Two network paths with two policies inside one app. The Kotlin side exports the discovered proxy (`HTTPS_PROXY`, `NO_PROXY`) into the environment of every CLI child process (`NetworkEnv.kt:115`, `buildAuthProxyEnv`), so the agents work behind the proxy. The app's own Dart clients — the update check, the license API, model discovery — were plain `HttpClient()` instances (`update_transport.dart`, pre-fix line 23) that knew nothing about that proxy and dialed direct; direct was refused. The second half is worse: the update check's failure branch returned a `failure` value and **wrote nothing** (`update_service.dart`, pre-fix `:128`) — no log line, no diagnostic event, no timestamp. The device was not offline, the server was not down, and the only trace was the absence of a trace.

Fix (S18-A/C, commits `76540347`, `363f77ca`): one proxy policy per process — a shared `AppHttpClientFactory` whose `findProxy` follows the same snapshot the CLIs get (`app_http_client.dart:17–48`, `:243–260`), with a refuse-to-fall-back rule (a configured proxy that is unreachable is an error, never a silent switch to direct); and a classified failure (`classifyFailure`, `update_service.dart:635`) that emits one log line and one diagnostic event (`:569`) and surfaces the last attempt, last success, and failure class in Settings.

Two lessons:

1. **One process, one network policy.** When a subsystem grows its own transport, the environment it inherits is not the environment the rest of the process observes. The bug was not in either client; it was in having two.
2. **A silent failure path is an unrepresentable state for the operator.** §5.2 was about a state the *code* could not name. This is the same defect one layer up: the code could name it (`failure`) but gave the human no way to see it. Absence of evidence became, for thirty minutes, evidence of absence.

5.6 A protocol enforced in one place is not enforced

The engine-slot discipline **existed only where it was discovered** (the chat surface). It was never written down anywhere both surfaces could see. Settings and onboarding therefore each independently violated it (§5.4).

Lesson. A protocol must live at a layer all its executors can see. Fixing an invariant at one entry point does not make the invariant hold. A cross-surface contract needs a single authoritative definition point, or it is merely "a comment in some file."

5.7 A counterintuitive summary

Note what §5.1–§5.6 have in common: **none is an architecture problem.** Embedding a Linux userland, surviving background management, mediating permissions — the "sounds hard" problems all got solved. What actually caused user-visible faults was **accounting and representation**:

- A signal closed an entry it did not own (§5.1)
- A domain state had no value in the types (§5.2)
- An invariant held at only some entry points (§5.4, §5.6)
- The "empty history" of a test device is itself a state that hides bugs (§5.3, §5.4)
- One process running two network policies, and a failure path with no output (§5.5)

Why this matters to researchers. None of these failures can appear in a benchmark. SWE-bench-style evaluation measures "how correctly an agent solves an isolated task," whereas accounting failures **accumulate over time, cross sessions, and depend on device history.** They can only be exposed by real use and only found by adversarial scrutiny. That is a gap in evaluation methodology.

6 A CLI Agent on a Five-Inch Screen

An agent designed for a 27-inch monitor and a terminal requires every design decision to be remade for a five-inch screen. Keva's answer is **Command Transparency**: everything the agent does is rendered in

structured form — worklog, diff, and web cards.

The honesty rule. When a turn ended with sub-agents still unaccounted for, the old interface marked them all **done**. That looks like a "friendly default"; it is actually dangerous:

> A finished-looking row for work whose outcome is unknown is worse than no row.

Generalized to any agent UI: **the interface must be able to say "I don't know."** A surface that can only render "success" and "failure" will render uncertainty as success — because "unknown" has nowhere to go, it necessarily collapses onto one side of the binary. In §5.2, the accounting layer's unrepresentable state ultimately became visible to the user precisely as "rendered as success." The representation failure and the UI failure are **two ends of the same failure**.

7 Does It Hold Up?

The evaluation stance, stated up front. This is not a benchmark table. We do not compare against a baseline, for three reasons: (1) there is no comparable public baseline for on-device coding agents; (2) the real failure modes (§5) accumulate over time and device history, and a single benchmark cannot capture them; (3) we believe "what can cut a run short" is more informative than "how much faster on average."

So §7 offers two things instead: **a verifiable timeout-budget inventory** (all real constants in the code), and **the log evidence that actually exists**.

7.1 The long-horizon budget: an inventory of what can interrupt a run

These are not performance figures; they are the system's **failure boundaries**. Each one points at a line of code:

Mechanism	Value	Location	Note
Hard turn timeout	12 h	`chat_controller.dart:385`	Absolute ceiling
First-event timeout	180 s	`chat_controller.dart:390`	Calibrated on 16 successful GLM turns: median 6.1 s / tail 65.5 s to first semantic event
Runtime-instance watchdog	15 s	`chat_controller.dart:391`	
Liveness gap	3 min	`chat_controller.dart:392`	**Armed only for runtimes declaring `emitsHeartbeats`** (`:3258`)
Heartbeat deferral cap	3 ≈ 12 min	`chat_controller.dart:1665`	Deferred while the process is alive, `:3345–3354`
Idle reclaim	5 min	`chat_controller.dart:394`	Callback guarded by `_canReclaimPersistentProcess` (`chat_page.dart:1401–1402`); while a turn runs the state is not idle, so no reclaim
Sub-agent settle fallback	10 min	`chat_controller.dart:291`	A settle-driven finish **does not count as normal success**
Idle settle	15 s	`chat_controller.dart:292`	
Slow-response threshold	12 s	`chat_controller.dart:393`	

An earlier outline quoted session figures (wall clock, tool count, RSS, a quiet interval) for which no measurement record exists in the repository (Appendix B). We do not print unmeasured numbers; this section gives the verifiable inventory and the log evidence that does exist.

7.2 The log evidence that actually exists

- `log/wp4_batch4_2026-07-16/README.md:23` : records a **25.739-second** long-bash turn.
- `log/android_logcat_relevant_2026-06-30.txt:1,4` : `task_progress` reporting `tool_uses:33/34` inside a single Explore sub-agent (an intra-subtask count, **not** the total for the turn — the two must not be conflated).
- `chat_controller.dart:3336–3340` : a **device-evidence comment** recording a roughly **168-second quiet event** and how it was handled.
- `log/android_exit_info_2026-06-30.txt:40` : `pss=155MB rss=187MB` — but this is the record of the **app being killed by `installPackageLI`** , **not** an engine-session RSS curve; it must not be cited as engine memory usage.

Conclusion. The evidence in the repository today supports the qualitative claim "the system runs, and long tasks have explicit failure boundaries." It does **not** support any quantitative performance claim. Closing that gap requires a controlled on-device session capture — we list that as explicit future work (§9).

7.3 What we cannot measure

From process state **alone**, "the model is generating" and "the connection has stalled" are indistinguishable — both present as a sleeping process consuming no CPU. Only **what happens next** separates them. This is a fundamental observational limit, not an implementation defect, but it means any process-probe-based liveness decision has a false-positive window; `_maxLivenessDeferrals = 3` (about 12 minutes of grace) is the insurance bought for that window.

8 Maintenance Methodology: One Human Directing Multiple Models, Under a Verification Discipline

Keva is built by **one person** directing **several models**. The method is documented here because it is relevant to anyone researching "LLM-assisted software engineering" — not to be clever about process.

The division of labor: one model acts as **auditor** and writes no implementation code; others implement against **written dispatches** carrying source references, prior work, and explicit non-goals. Every receipt is re-verified against the code before it is believed.

Three practices carry the weight:

8.1 Mandatory self-check

An implementer must **revert its own fix** and show the new test failing **with the exact symptom of the original bug**, then restore it.

> A test that cannot be made to fail is not evidence.

8.2 Vacuity checks

Asked whether a negative assertion still guarded anything, an implementer demonstrated that **reverting two comparisons still let ten tests pass while a real mis-delivery was present.** The guard had become decoration.

Coverage cannot see this; a green suite cannot see this. This is an important phenomenon in test-effectiveness research: an assertion can pass continuously for years while no longer protecting anything. To find it, you must perform an explicit **mutation check** — break the thing being protected, and see whether the test alarms.

8.3 Adversarial re-verification

A derived root-cause judgment is handed to a second model with instructions to **refute** it.

8.4 Two ratios worth knowing

- **About one third of model-produced audit findings did not survive line-by-line verification.** Generated review is a lead generator, never a verdict.
- **The auditor's own root-cause call was once overturned by an implementer with evidence.** A test file whose diff contained a single annotation was taken as proof of a production regression; in fact a

sibling suite had been updated for a deliberate behavior change and this file had been missed. The reasoning was plausible, cheap, and **wrong** — which is why the refutation step is not optional.

8.5 What this means for a pull request

The bar is not "tests pass." It is "**you can show the test failing without your fix.**"

9 Limitations and Threats to Validity

Current, specific, unhedged.

#	Limitation	Nature
1	`chat_controller.dart` is a single file >4,000 lines — the largest maintainability debt	Engineering debt, pending decomposition
2	Self-built network layer (`LocalDnsServer.kt` + proxy) instead of mature `http` / `dio` ; app-level clients now share proxy discovery (§5.5) but the layer is still hand-built	Maintenance burden; narrower error handling surface
3	Battery-optimization exemption is **optional**; without it, OEM background management may intervene in long tasks	Platform dependency
4	The `targetSdk = 28` SELinux/background-service arbitrage depends on platform behavior and **will expire**	Platform dependency (§4.7)
5	No ptrace; permission mediation is an application-layer pipeline, weaker than system-level interception	Architectural trade-off (§4.5)
6	**The evaluation is not a benchmark**: no quantitative performance data, no baseline comparison	Threat to validity (§7)
7	One maintainer. Bus factor one.	Organizational risk
8	Location is foreground-only; no background tracking	Product boundary
9	Without a proxy, direct connections from some mobile carriers to the license and update endpoints time out (observed 2026-09-15 on a 5G network in China); first activation depends on that path	Deployment dependency

Honest note on validity (threat #6). Every quantitative-flavored claim in this paper is limited to "constants verifiable in the code" and "records that exist in the logs." We ran **no controlled experiment**, have **no sample size**, and claim **no statistical significance**. The case studies in §5 are **anecdotal evidence**, though each is verifiable down to a line of code. Readers should calibrate their trust in this paper's conclusions accordingly.

10 Conclusion

Keva demonstrates that one thing is feasible: **running a real, multi-runtime coding agent on an unrooted Android phone** — not a remote terminal, not an API wrapper, but a system running a complete agent loop locally.

But what this paper wants to leave behind is not "we did it." It is three categories of transferable knowledge:

1. **A problem taxonomy** (§3.1): split "impossible" into performance / permission / lifecycle. Performance is usually fine; permission can often be arbitraged; lifecycle is the real hard part.
2. **The failure mode of accounting, representation, and observability** (§5): a signal closing an entry it does not own; a state with no value in the types; an invariant holding at only some entry points; a failure path that produces no evidence. These failures **surface only in real use** and are invisible to existing benchmarks — a gap in evaluation methodology that researchers would do well to fill.
3. **The choice of absorption layer** (§4.6): capability asymmetry should be absorbed at the highest layer (presentation) or at the declaration layer (capability flags) — not by special-casing on runtime kind at call sites.

The one claim worth making here, because it is about method rather than priority: building this way — auditing, dispatching, refusing to believe a fix until it has been shown to fail — produced better code than working alone would have. Said as an experience, offered for the reader to test, not as a result.

References

Official documentation and platform specifications

1. Android Developers. *Background execution limits / Doze mode / App Standby*. developer.android.com. (Authoritative account of app background restrictions; basis for §4.4 and limitation 3 in §9.)
2. Android Developers. *Protecting users with SELinux /* `execute_no_trans`. source.android.com. (Basis for the SELinux discussion in §4.2 and §4.7.)
3. Android Developers. *App resources / behavior changes by* `targetSdkVersion`. (The platform mechanism behind the constraint arbitrage in §4.7.)
4. Android Developers. *Process and thread lifecycle / foreground services / wake locks*. (Official semantics for the keep-alive mechanisms in §4.4.)
5. The Flutter team. *Flutter documentation*, and the Dart platform-channels (MethodChannel) documentation. flutter.dev. (The cross-layer communication mechanism in §4.1.)
6. Simolus3. *drift: Reactive persistence library for Flutter and Dart*. pub.dev/packages/drift. (The database choice in §4.1.)
7. The PRoot project. *proot: chroot without root, using ptrace*. github.com/proot-me/proot. (The original source of the PRoot approach discussed — and ultimately not used for Claude — in §4.2.)
8. The Termux project. *Termux: Android terminal emulator and Linux environment*. github.com/termux/termux. (A predecessor in running a Linux userland on Android; one of the idea sources for §4.2.)
9. Anthropic. *Claude Code documentation*. docs.claude.com/claude-code. (Claude CLI runtime behavior and the `--include-partial-messages` option in §4.2.)
10. OpenAI. *Codex CLI / app-server documentation*. github.com/openai/codex. (The second runtime in §4.3.)

Academic literature

11. Yao, S. et al. (2023). *ReAct: Synergizing Reasoning and Acting in Language Models*. ICLR 2023. (The foundational tool-use agent paradigm; background for §3.2.)
12. Shinn, N. et al. (2023). *Reflexion: Language Agents with Verbal Reinforcement Learning*. NeurIPS 2023. (Agent self-correction; methodologically related to the self-check discipline in §8.)
13. Jimenez, C. et al. (2024). *SWE-bench: Can Language Models Resolve Real-World GitHub Issues?* ICLR 2024. (The methodological contrast in §5.6 — existing benchmarks do not capture accounting failures.)
14. Liu, X. et al. (2024). *AgentBench: Evaluating LLMs as Agents*. ICLR 2024. (Existing agent-evaluation paradigms and their limits.)
15. Wang, L. et al. (2024). *A Survey on Large Language Model based Autonomous Agents*. Frontiers of Computer Science. (Survey grounding the positioning of §3 and §4.)

> Note on citation practice: this paper lists only material actually consulted during writing. It does not position against related work and claims no priority. Academic literature is cited for methodological background; official documentation and open-source projects are the source of engineering facts.

Appendix A: Verification Map

Every technical claim in this paper maps to a concrete location in the codebase. This is the paper's substitute for peer review — the reader can verify any of it without asking anyone.

#	Claim in this paper	Code location
A1	`targetSdk = 28` is an intentional SELinux workaround	`android/app/build.gradle.kts:98, 104, 153`
A2	Claude CLI does not run under PRoot (Bun JIT conflict); ld-linux bridging instead	`ProcessManager.kt:30–32` ; `buildProotCommand` `:2664–2668` , `:3467` ; `ProcessManagerProotTest.kt:29/48/67`
A3	Rootfs install layout	`InstallManager.kt` ; `StorageCleanManager.kt:100` (`files/mcagent-runtime/`)
A4	Engines started under `setsid` , outliving the app	`EngineOrphanGuard.kt:15` ; `ProcessGroupAndChildrenTest.kt` (F6/F7)
A5	Four-check pre-kill verification, fail-closed	`EngineOrphanGuard.kt:104–113` (verify), `:28–29` (fail-closed), `:96–101` (reclaimOrphan), `:186–224` (AndroidProcessProbe; starttime = field 22, sliced from the last `)` at `:199–209` ; uid via `Os.stat` at `:221–223`)
A6	Watchdog constants	`chat_controller.dart:385` (12 h), `:390` (180 s), `:391` (15 s), `:392` (3 min), `:394` (5 min), `:291` (10 min settle), `:292` (15 s idle settle), `:393` (12 s slow response)
A7	Liveness armed only for heartbeat-declaring runtimes	`runtime_capabilities.dart:24–31` ; `claude_cli_agent_runtime.dart:145` (true); `codex_app_server_agent_runtime.dart:358` (false); `chat_controller.dart:3258–3260, 3299` ; deferral cap `:1665`
A8	Streaming granularity absorbed at presentation	`live_text_pacer.dart:3–9` ; `chat_page.dart:5721–5722` ; M-0533 disabling `--include-partial-messages` : `chat_controller.dart:3334` , `agent_event.dart:314–315`
A9	The two handoff resolution timings	Cross-runtime assemble-at-send: `chat_controller.dart:2845–2882` (`assembleForSession` `:2857`), `handoff_brief_service.dart:8–13` ; same-runtime stash-at-switch: `chat_page.dart:4099–4142` , armed at `:1666–1669, 2058–2061, 8423` , prepended at send by `chat_controller.dart:2876–2881`
A10	Ledger closes by identity	`_pendingSubagentIds` `chat_controller.dart:1604` ; `onSubagentReturned:2105–2118` (id-less signal returns at `:2116` ; old FIFO pop documented `:2110–2115`); `onSubagentToolResult:2137–2152` ; `drainFinishedBackgroundTasks:2090–2103` (old false-convergence documented `:2058–2082`)
A11	Terminal state is fail-closed	`lib/core/codex/codex_event_mapper.dart:557–573` (M-0532); `claude_event_mapper.dart:335–344` , `:266–306` (M-1372; old symptom "the HTML got built but the turn never ended" at `:272–274`); settle finish not counted as success, `chat_controller.dart:470–475`
A12	Idle reclaim is state-guarded	`chat_controller.dart:3722–3734` (arming), `:3656–3657` (call site); `chat_page.dart:1401–1402` (guard)

#	Claim in this paper	Code location
A13	Dependency choices	`pubspec.yaml:16–17` (Riverpod), `:20–21` (drift), `:22` (secure storage), `:37` (archive), `:38` (ffi), `:61–63` (geolocator, foreground only)
A14	Keep-alive service	`McagentKeepAliveService.kt:18–24` (foreground service + PARTIAL_WAKE_LOCK + START_STICKY)
A15	Storage cleanup default-deny	`StorageCleanManager.kt:10–23` (default-deny allowlist, no symlink following)
A16	Real log evidence	`log/wp4_batch4_2026-07-16/README.md:23` (25.739 s turn); `log/android_logcat_relevant_2026-06-30.txt:1,4` (tool_uses 33/34); `chat_controller.dart:3336–3340` (168 s quiet event); `log/android_exit_info_2026-06-30.txt:40` (pss=155MB, **not** engine session RSS)
A17	Architecture scale	Line counts per layer in §4.1; `chat_controller.dart` >4,000 lines
A18	Self-built network layer	No `http` / `dio` dependency (`pubspec.yaml`); `LocalDnsServer.kt`
A19	One proxy policy per process; failures classified and logged	`NetworkEnv.kt:115` (`buildAuthProxyEnv`), `MainActivity.kt:136` (`getProxyConfig` channel); `lib/core/net/app_http_client.dart:17–48` (resolver), `:243–260` (factory, `withProxyRetry`); `lib/modules/update/update_service.dart:635` (`classifyFailure`), `:569` (log line); pre-fix: `update_transport.dart:23` (`HttpClient.new`) and `update_service.dart:128` (silent `failure`) at commit `76540347^`

Appendix B: Differences from the Earlier Outline

This paper supersedes `doc/paper/Keva_Outline_v1_*` (2026-08-16, drafted by Opus). Relative to that outline, the following **verified** corrections were made:

Outline claim	Fact	Handling in this paper
"Five-layer architecture: UI → Riverpod → modules → core"	`lib/` has four top-level directories; Riverpod lives in `core/providers/`	§4.1 gives four layers
"Embeds a full glibc userland via PRoot"	The Claude CLI does **not** run under PRoot (Bun JIT conflict); it uses ld-linux bridging	§4.2 corrects this
"Sub-agent terminal state is **still** hard-coded as `completed` "	**Fixed**: fail-closed (M-0532); the ledger closes by identity (M-1363/M-1372)	§5.1, §5.2 present it as "was unrepresentable → now representable"
"One real session: 11 min / 32 tool calls / RSS 146–155 MB / 5.5-min quiet interval"	**No measurement record** for any of these four numbers exists in the repository	§7 presents the verifiable budget inventory plus existing log evidence, and says so explicitly
"Handoff: one resolves at switch time, the other at send time"	The distinction is **same-runtime vs cross-runtime**, not one runtime vs the other	§4.6 instance 3 separates the two branches

Appendix C: Changes in v1.1 (2026-09-17)

Section	Change	Reason
Header, §1.3, §4.2	Package size 600 MB → 746 MB (release 2026.38.5)	Measured artifact
§4.3	Directory tree corrected: `codex_event_mapper.dart` lives in `lib/core/codex/`, not under `adapters/`	Verification-map integrity
§4.7	The `targetSdk` comment is now quoted verbatim; the six-hour rule is stated precisely (Android 15, `dataSync` / `mediaProcessing`, API 35+); the Play Protect prompt is recorded as a visible cost	Accuracy
§5.5 (new)	Sixth failure case: two network policies in one process, and a failure path that logged nothing (2026-09-15)	Surfaced after v1.0 during release testing
§5.6–5.7, §1.4, §10	Renumbered; "accounting and representation" widened to include observability	Consistency with §5.5
§7.1	The outline-figures disclaimer shortened and pointed at Appendix B	Readability
§9	Limitation 2 updated; limitation 9 added (carrier direct-connect timeouts without a proxy)	Observed 2026-09-15
Appendix A	A11 path made explicit; A19 added for §5.5	Verifiability