Keva：在未 root 的 Android 手机上运行真实 Coding Agent——一个系统工程案例研究

Architecture, failure modes, and transferable engineering lessons

作者：Keva 项目组 · v1.0 2026-09-16 · v1.1 修订 2026-09-17（见附录 C）

摘要

现代旗舰手机的内存、CPU 核心数与网络连接能力，已达到几年前工作站的水平。手机真正缺的不是**性能**，而是**许可**：移动操作系统把应用定义为 UI 表面的消费者，而不是长生命周期进程的宿主——这些进程需要能 fork 与 exec、跨小时持有状态、并在被切到后台后继续存活。

Keva 是一个 Flutter Android 应用，它把一套完整的 glibc Linux 用户态（Ubuntu rootfs）打包进 APK，在未 root 的应用沙箱内运行两个真实的 coding agent 运行时——Claude Code CLI 与 Codex app-server——并统一在一个接口背后。它是一个在手机上真正可用的 coding agent，本文是关于"这件事实际付出了什么代价"的系统工程案例研究。

本文的核心发现是：**教给我们最多的障碍不是架构层的，而是"记账、表示与可观测性"层面的失败**。架构层的障碍任何人都能预见——嵌入运行时、对抗 OEM 后台管控、在没有 ptrace 的平台上中介权限——它们费事，但都已被解决。而那些只在系统被一个真人用起来时才暴露的失败——一个完成账本被"先到的信号"而非"持有该条目的信号"清空；一个子代理的死亡状态在所有层都不可表示，因此所有层都无法处理；一个排他性保证被一个操作系统有权随时销毁的进程在内存里持有；一条失败路径走完了却什么也没记下——才是真正难的部分。

我们逐项报告每一条失败、修法、以及哪些可迁移到手机之外的场景；描述系统是如何在一人指挥多模型的验证纪律下被建造和维护的；并给出一份验证映射附录，使本文每一条技术主张都可被读者独立核验。

关键词：移动端智能体；on-device LLM；Android 沙箱；进程保活；多运行时抽象；失败案例研究

1 引言

1.1 一台你不能编程的电脑

如果一个人只拥有一部手机，他想让一个 AI 智能体帮他改代码、读文件、跑测试，他能做什么？

硬件不是瓶颈。2026 年的旗舰手机拥有 12 – 16 GB 内存、八核 CPU、稳定的千兆级网络。这个配置跑一个基于 Node.js 的命令行智能体，在算力上绰绰有余。

真正的障碍是操作系统策略。Android 把每个应用关进一个严格的应用沙箱：SELinux 策略禁止应用执行任意二进制（`execute_no_trans` 拒绝）；应用没有 root 权限，不能 `mount`、不能 `ptrace`；进程被切到后台后，会遭遇 Doze、App Standby、以及各 OEM 定制的后台清理策略；一个长时间持有 CPU 与内存的进程，会被系统视为"行为不端"而杀死。

换言之，**手机是一台你不能编程的电脑**——不是因为它的算力不够，而是因为操作系统不许可。

1.2 为什么答案不是"放云上跑"

诚实的备选方案有三个，每一个都放弃了某些东西：

- **SSH 到一台服务器**：放弃了离线能力，放弃了低延迟，而且引入了新的信任边界——你最想让智能体触碰的数据（通讯录、照片、本地文件、位置）恰恰在手机上，不在服务器上。
- **云 IDE / 远程开发环境**：同样依赖网络，且把代码与凭证都搬到了第三方。
- **云端智能体 + 手机当瘦客户端**：延迟与成本之外，最大的问题是**数据引力**——智能体需要操作的对象在端侧。

Keva 的立场是：如果用户唯一的电脑是手机，那么智能体就应该在手机上跑。不是把手机当终端，而是把它当宿主。

1.3 Keva 是什么

一句话：**Keva 是一个 Android 应用，它在未 root 的手机上运行真实的 coding agent（Claude Code CLI 与 Codex app-server），让用户可以在五英寸屏幕上完成读代码、改文件、跑测试、提交这类工作。**

它把整套 Linux 工具链（Ubuntu rootfs、Node.js、Python、两个 agent CLI）打包进 APK（release 2026.38.5 为 746 MB），安装后在设备本地解压、以子进程方式运行。它不依赖任何云端编排——模型推理走各 LLM 供应商的 API，但**智能体循环（agent loop）本身在手机上**。

1.4 贡献

本文的贡献是一个案例研究，而非一个 benchmark。具体地：

1. **设计与实现**：一个在未 root Android 应用沙箱内嵌入完整 Linux 用户态、并运行两个真实 coding agent 运行时的系统（§4）。
2. **敌对 OS 上的存活机制**：一套 watchdog 基质与跨进程孤儿核验机制（四重核验、fail-closed），处理"进程被 OS 回收后，内存里的排他性保证重建为空，而孤儿进程仍在运行"这一跨进程一致性问题（§4.4）。
3. **能力不对称的三条吸收规则**：当两个运行时能力不对称时，应在哪一层吸收差异——本文给出三个实例与三条不同的答案（§4.6）。
4. **六个真实失败案例**：核心教训是"记账、表示与可观测性"层面的失败比架构层失败更致命，且只在真实使用中暴露（§5）。
5. **一人指挥多模型的验证纪律**：三项可复用的工程实践与两个可核验的比例数字（§8）。
6. **验证映射附录**：本文每一条技术主张都映射到具体文件路径与代码行，读者可独立核验（附录 A）。

1.5 本文怎么读

- **评估者**（这东西靠谱吗）：→ §2（能力清单）、§7（扛得住吗）、§9（限制）
- **贡献者 / 研究者**（怎么被维护的、我能引用什么）：→ §8（方法论）、附录 A（验证映射）
- **对问题本身好奇**：→ §3（背景）、§4（设计）
- **想看失败案例**：→ §5

2 今天什么能跑

这一节故意放在设计章之前。一个决定要不要投入注意力的读者，不该被迫先读六页架构。

2.1 能力清单

能跑：

- 两个真实 agent 运行时（Claude Code CLI、Codex app-server）在同一接口背后切换
- 文件读写、目录浏览、shell 命令执行（在应用沙箱内的 Linux 用户态中）
- 多轮对话、子任务派发（sub-agent）、工具调用与结果渲染
- Web 搜索、代码 diff 展示、结构化工作日志（worklog）
- 本地数据库持久化会话与配置；多 LLM 供应商（Anthropic + OpenAI 兼容端点）

不能跑 / 受限：

- 需要 root 的操作（`mount`、系统分区写入）
- 需要 ptrace 的系统级权限中介（见 §4.5 的应用层替代方案及其弱点）
- 后台长时间高 CPU 任务受 OEM 省电策略威胁（见 §9）

2.2 已知坏清单（开门见山）

- 单一控制器文件 `lib/modules/chat/chat_controller.dart` 已超过 4000 行，是系统最大的可维护性债务（§9）
- 网络层没有使用成熟的 `http` / `dio` 库，而是自研 `LocalDnsServer.kt` + 代理。自 v1.1 起，应用自身的 HTTP 客户端已与 CLI 共享同一套代理发现（§5.5），但这一层仍是手写的（§9）
- 电池优化豁免是可选的，用户不开启则长任务可能被 OEM 后台管理干预（§9）
- 本文不做 benchmark 式评估（§7 明确说明原因与替代方案）

3 背景与动机

3.1 问题的真实形状

把"在手机上跑 coding agent"这个问题拆开，会得到三类性质完全不同的子问题：

类别	例子	性质
性能	内存够不够、CPU 够不够	**不是问题**。2026 年旗舰机足够
许可	SELinux 禁止 exec、无 root、无 ptrace	**硬约束，但可工程绕过**（§4.2、§4.5）
生命周期	后台被杀、进程被回收、跨进程状态不一致	**真正困难的部分**（§4.4、§5）

这个分类本身就是本文的第一个可迁移结论：**当有人告诉你"某事在手机上不可能"时，先问清楚他指的是哪一类。** 大多数"不可能"其实是"许可"类，而许可类问题在工程上往往有套利空间（§4.7）。

3.2 一个被低估的事实：智能体循环在端侧

主流的"手机 AI 助手"是**远程过程调用 + 本地 UI**：手机把用户输入发给云端，云端跑智能体循环，把渲染指令流回来。这种架构下，手机只是一个终端。

Keva 的架构是**本地智能体循环 + 远程推理**：agent loop（工具调度、文件操作、上下文管理、子任务派发）在手机本地跑，只有 LLM 推理走 API。这不是"离线优先"的教条，而是一个事实判断：**一个会操作文件的智能体，应该待在文件所在的地方。**

4 系统设计

4.1 总体架构

Keva 分为应用层（Flutter/Dart）与平台层（Android Kotlin）。

Dart 侧（`lib/`）实际包含四个顶层目录：

层	文件数	代码行数	职责
`ui/`	47	35,384	页面、组件、渲染（含 Command Transparency 卡片）
`core/`	120	31,511	运行时抽象、数据库、桥接、诊断、契约
`modules/`	50	18,590	功能模块：chat、entitlement、auth 等
`l10n/`	3	13,974	国际化（生成代码）

其中 `core/` 的大子目录：`contracts` 6,749 行、`database` 4,669 行、`codex` 4,614 行、`diagnostics` 4,557 行、`runtime` 4,115 行、`bridge` 2,970 行、`engine_pack` 2,278 行、`handoff` 412 行。依赖注入使用 Riverpod（`flutter_riverpod ^2.6.1`），但它位于 `core/providers/` 之内，不是独立的顶层架构层。

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

模块	职责
`ProcessManager.kt`	Claude CLI 子进程的构建与生命周期管理
`CodexAppServerProcessManager.kt`	Codex app-server 进程启停、stdio 转发、generation 防陈旧、共享 `EngineMutex`
`InstallManager.kt`	rootfs / Node / 两个 CLI 的安装布局（`files/mcagent-runtime/`）
`McagentKeepAliveService.kt`	前台服务 + `PARTIAL_WAKE_LOCK` + `START_STICKY`
`EngineOrphanGuard.kt`	跨应用重启的引擎排他与杀前核验
`StorageCleanManager.kt`	存储统计/清理，默认拒绝白名单

依赖选型（`pubspec.yaml`）：状态管理 `flutter_riverpod`；数据库 `drift ^2.24.0` + `sqlite3_flutter_libs`（drift 在磁盘故障时可回退内存数据库并保持单一实例共享）；密钥 `flutter_secure_storage ^9.2.4`；FFI `ffi ^2.2.0`；资产解包 `archive ^4.0.9`；图片压缩 `image ^4.5.4`；定位 `geolocator ^13.0.2`（仅前台，不做后台追踪）。

值得注意的是：**网络层没有使用 `http` 或 `dio`**，而是自研 `LocalDnsServer.kt` 与代理层。这是一个有意的工程选择（为了精确控制流量走向与 DNS 行为），但也是一笔维护债务（§9）。

4.2 在未 root 的应用内嵌入 Linux 用户态

核心思路：把一套完整的 Ubuntu rootfs（glibc、Node.js、Python、两个 agent CLI）作为资产打进 APK，安装后在应用私有目录解压，**然后在应用沙箱内以应用自身的 uid 执行其中的二进制**。

这里有一个重要的实现细节修正。最初的方案是用 PRoot（一个基于 `ptrace` 的用户态 `chroot` 替代）来提供文件系统视图。但 `ProcessManager.kt` 的文档明确记载（: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`).

即：**Claude Code CLI 实际上并不跑在 PRoot 里**，因为 PRoot 的 ptrace 拦截与 Bun（Claude Code 的 JS 运行时）的 JIT 冲突。实际做法是**直接用 rootfs 的动态链接器桥接启动**：以 `ld-linux-aarch64.so.1` 作为程序入口、把 rootfs 的库路径注入 `LD_LIBRARY_PATH`，从而在不需要任何 `chroot`/`ptrace` 的情况下执行二进制。（代码中 `buildProotCommand` 这个名字是历史遗留，它实际发射的是 ld-linux 桥接命令，并做前置条件检查：链接器缺失 `linker_missing`、ptrace 助手缺失 `supervisor_missing`、助手不可执行 `supervisor_not_executable`——见 `ProcessManagerProotTest.kt`。）

为什么用"内置运行时"而不是"首次启动下载"：这是一个明确的产品决策。整包约 750 MB，其中运行时资产占绝大部分。代价是体积，换来的是：安装即用、离线可用、不存在"首次启动下载失败"的故障面。在系统软件领域，把依赖打包进去而不是运行时拉取，是经典的"可用性换体积"权衡。

4.3 一个接口，两个运行时

Keva 在两个 agent 运行时之间抽象出一个 `AgentRuntime` 缝隙（`lib/core/runtime/`）：

```
lib/core/runtime/
├── agent_runtime.dart              # 抽象缝
├── runtime_capabilities.dart       # 声明式能力（心跳、流式粒度…）
├── runtime_coordinator.dart        # 协调器接口
├── runtime_coordinator_impl.dart   # 实现
├── coordinator_state.dart
├── runtime_identity.dart
├── runtime_auth_adapter.dart
├── runtime_configurable.dart
├── agent_event.dart                # 统一事件模型
├── agent_runtime_status.dart
├── agent_runtime_diagnostic.dart
└── adapters/
    ├── claude_cli_agent_runtime.dart      # Claude 适配器
    ├── claude_event_mapper.dart           # Claude 事件映射
    └── codex_app_server_agent_runtime.dart # Codex 适配器

lib/core/codex/
└── codex_event_mapper.dart             # Codex 事件映射（与 app-server 客户端放在一起）
```

这条缝隙是被两件事逼出来的：**供应商模型标识符的频繁变动**（见 §4.8 的"读时水合"），以及**两个运行时的能力差异**（见 §4.6）。它换来的是：接入第二个引擎的成本远低于第一个——只需要一个新的适配器对（runtime + event mapper），而不是又一次全栈改造。

4.4 在敌对的操作系统上活着

这是本文最硬的技术部分。Keva 维持一个 **watchdog 基质**，全部定义在 `lib/modules/chat/chat_controller.dart`：

机制	常量	位置	作用
turn 硬超时	`Duration(minutes: 720)` = 12 小时	`:385`	单轮对话的绝对上限
首事件 watchdog	`Duration(seconds: 180)`	`:390`	turn 开始后若 180 秒无任何语义事件则判定卡死。注释记载这是基于实测标定的：16 次 GLM 成功 turn 的首个语义事件中位数 6.1 秒、尾部 65.5 秒，180 秒约为尾部值的 2.7 倍
运行时实例 watchdog	`Duration(seconds: 15)`	`:391`	运行时实例本身的存活检查
liveness gap	`Duration(minutes: 3)`	`:392`	长时间无心跳时间隔判定
idle reclaim	`Duration(minutes: 5)`	`:394`	空闲后回收常驻进程

关键的跨进程一半：引擎是用 `setsid` 启动的（成为会话组长），因此**它会活过应用进程**。这带来一个微妙的一致性问题：如果应用被 OS 回收，应用内存里的"引擎排他性保证"会被重建为空，而那个孤儿引擎进程仍在运行。下次启动时，新应用实例需要决定是否清理它。

`EngineOrphanGuard.kt` 的做法是**杀之前先核验**，而且 fail-closed。`verify()`（`:104–113`）要求四项全部通过才允许杀：

1. `isAlive`（`:105`）——进程还在

2. `/proc/<pid>/stat` 的 starttime（`:106-107`）——进程的"化身"不可变，PID 复用必然改变此值
3. cmdline（`:108-109`）——`exec` 之后不可变
4. `/proc/<pid>` 目录的 owner uid（`:110-111`）——必须等于应用 uid

任一不可读或不匹配，`verify()` 返回 `false`——**不杀，也不报错**（`:28-29`）。`reclaimOrphan()`（`:96-101`）在核验失败时返回一个"非我物"的结果（`fullyReaped=true`），而不是强行杀戮。记录文件撕裂或畸形输入被解析为"无记录"（`:31-32`、`:125-135`），不做猜测。

`AndroidProcessProbe`（`:186-224`）负责具体解析：`starttime` 是 `/proc/<pid>/stat` 的第 22 个字段，解析时**从最后一个 `)` 之后开始切分**，以避开 `comm` 字段中可能包含的空格（`:199-209`）；uid 来自 `Os.stat`（`:221-223`）。

设计原则被写成一句注释（`EngineOrphanGuard.kt`）：

> The guard would rather miss an orphan than kill a stranger.
> 守卫宁可漏判一个孤儿，也不会误杀一个陌生人。

进程组行为本身有仪器测试覆盖（`ProcessGroupAndChildrenTest.kt`，F6/F7）：验证 `setsid` 之后进程成为自己的组长（`pgid == pid`），并确认某些 ROM 上的 `setsid` 是 toybox/busybox 小程序、在进程内直接调用 `setsid(2)` 而不 fork 这一平台差异。

4.5 没有 ptrace 的权限中介

真正的桌面智能体（如基于 LD_PRELOAD 或 seccomp 的方案）依赖系统调用拦截来做工具执行前的权限门控。Android 应用沙箱**不提供**系统调用拦截能力。

Keva 的替代方案是**应用层管道**：所有工具调用必须经由应用进程的中介层，由中介层在执行前做权限判定。这比真正的 ptrace 弱——一个绕过中介直接与引擎进程通信的路径原则上可以逃过门控。**我们明确承认这个弱点**，而不是把它包装成"等价于系统级沙箱"。在实践中，引擎是应用自己启动的子进程、stdio 由应用托管，因此"绕过中介"需要修改引擎本身——这不是不可能，但门槛远高于误用。

4.6 能力不对称该在哪一层吸收

这是本文的核心设计思想。两个运行时能力不同，强制设计者选择在哪一层吸收差异。三个实例，三种不同的答案：

不对称	在哪里吸收	规则	代码位置
一个发心跳，另一个从不发	声明式能力	按声明的能力门控，永远不要按运行时种类门控	`runtime_capabilities.dart:24-31`；`chat_controller.dart:3258`
一个流整块，另一个逐 token 流	呈现层	在两个运行时"看起来一样"的最高层吸收	`live_text_pacer.dart:3-9`
handoff 上下文的解析时机不同	交付契约	移动缝隙的位置，不要在调用处 special-case	`chat_controller.dart:2845-2882`；`chat_page.dart:4099-4142`

实例一：心跳。`runtime_capabilities.dart:24-31` 声明 `emitsHeartbeats`，默认 `false`，必须由运行时显式 opt-in。注释（`:26-30`，标记 HB-1）说明：Codex 原生不发心跳，如果为它武装心跳 watchdog，会误杀健康的 turn。因此消费侧（`chat_controller.dart:3258-3260`）只在 `capabilities.emitsHeartbeats` 为真时武装 liveness watchdog，并在活动重置路径上再校验一次（`:3299`）。进程仍活着时最多顺延 `_maxLivenessDeferrals = 3` 次（`:1665`，约 12 分钟宽容窗口），见 `:3345-3354`。

> 规则一：按声明的能力门控，永远不要按运行时种类门控。一旦写 `if (runtime is CodexRuntime)`，就把能力假设焊死在了种类判断上，新增第三个运行时时必然出错。

实例二：流式粒度。Claude CLI 不开启 `--include-partial-messages`（M-0533，`chat_controller.dart:3334`、`agent_event.dart:314-315`），因此它一次吐出整块文本；Codex 是逐 token 流。Keva 没有在 UI 层为两种形状写两套渲染，而是把两种喂法都汇入同一个 `LiveTextPacer`，并且让揭示速率成为当前积压量的纯函数（`live_text_pacer.dart:6-9`）。结果是：同一总文本以一大块喂入或逐 token 喂入，得到相同的每 tick 揭示。UI 侧对应 `chat_page.dart:5721-5722`（RT-2）。

> 规则二：在两个运行时还"看起来一样"的最高层吸收差异。呈现层是最高层——在这里吸收，下层完全不需要知道形状差异。

实例三：handoff 上下文的解析时机。这里需要区分两个分支，因为它们的时机真的不同：

- 跨运行时（Codex ↔ Claude）：brief 在发送时解析组装（`chat_controller.dart:2845-2882`，`assembleForSession` 在 `:2857`），且完全来自本地数据库（`handoff_brief_service.dart:8-13`）——刻意不询问旧运行时，因为旧运行时可能已经崩溃或凭证过期。
- 同运行时（Claude 回放）：transcript replay prefix 在切换/重启/预热时就解析并暂存（`chat_page.dart:4099-4142`，取最近 12 条消息、上限 12000 字符），在 `:1666-1669`（预热 resume 失败回退）、`:2058-2061`、`:8423` 处武装，发送时由 `chat_controller.dart:2876-2881` 拼到 prompt 前。

> 规则三：**移动缝隙的位置，不要在调用处 special-case。** 两个分支的差异被封装在"暂存 vs 组装"的契约里，发送路径只管"把暂存的东西拼上去"。

（代码读者注：`chat_controller.dart:437–441` 的注释把 handoff brief 标注为 "send time (CTX-1)"，这对跨运行时分支正确，但同运行时回放分支实为"切换时暂存 + 发送时拼接"，阅读代码时请注意区分。）

4.7 约束套利：`targetSdk = 28` 买到了什么

`android/app/build.gradle.kts:104` 将 `targetSdk` 设为 28，并且注释（`:98–103`、`:153`）明确说明这是**有意设计**：

> 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.

即：在 Android 10+ 上，`untrusted_app` 这个 SELinux 域禁止对 `app_data_file` 执行 `execute_no_trans`，所以应用进程无法 exec 从 filesDir 解压出来的 glibc 加载器与 `claude.exe`（这一点已实测确认：`avc denied execute_no_trans`）。把 targetSdk 定在 28 及以下，应用会落进 `untrusted_app_28` 域，该域允许这些操作。Termux 在不做 root 的情况下运行下载下来的原生二进制，用的是同一条路子。

这个选择有一个**顺带的、关键的回报**：Android 15 给 `dataSync` 与 `mediaProcessing` 类前台服务设了每天六小时的上限，但只针对 targetSdk 35 及以上的应用——定在 28 的应用不在这条规则里，这正是**多小时智能体运行能成为可能**的一部分。一笔为原因 A 接受的成本，在无关维度 B 上付了租金。

同一选择也有看得见的代价：Google Play Protect 会把任何 target 旧 API level 的旁加载应用标记为"为旧版 Android 打造"，并要求用户确认安装。这个提示在每一部带 GMS 的手机上都会出现，所以下载页在用户看到它之前先解释清楚。

这笔交易有保质期：它依赖平台行为，Google 未来收紧政策就可能使它失效。**我们把它写进限制（§9），而不是当作永久解法。**

4.8 读时水合（provider hydration on read）

LLM 供应商的模型标识符变动得足够频繁，以至于**存下来的配置会过期**。Keva 的做法（`lib/modules/config/config_service.dart`，语义见 M-1036）：每次读取时把内置 provider 的模型别名**重新对齐**到当前内置清单并重新持久化——因为模型标识符归应用所有，所以一次更新能到达所有已安装实例。唯一的例外是用户自有的 `custom` provider（自定义 base URL 与别名永不被覆盖）。

> 这是"数据归属决定迁移策略"的一个小例子：**应用拥有的配置，应用有责任保持新鲜；用户拥有的配置，应用无权改写。**

5 失败案例：记账、表示与可观测性

这一节是本文最重要的部分。六个案例都是真实的，都只在系统被真人在真实网络上使用时才暴露，而且**没有一个是架构层失败**。

5.1 账本归属：身份，而非数量

症状（用户报告）：*"主进程好像不知道拿死掉的 sub-agent 怎么办。"*

根因：sub-agent 在一个 pending 集合里被追踪（`Set<String> _pendingSubagentIds`，`chat_controller.dart:1604`）。完成信号从**多个通道到达**：有些携带 spawn 时的 tool-use id，有些不带。旧实现里，不带 id 的路径会从集合里弹出任一条目（FIFO pop）。因为一个 sub-agent 通常**上报两次**（一次带 id、一次不带），第二次上报就**挤掉了一个仍活着的** sub-agent 的槽位。账本提前清空 → 父 turn finalize → 被抛弃的 sub-agent 的产出被当作"已结束 turn 的产物"丢弃。

修复（M-1363 等）：账本改为**按身份关闭**。

- `onSubagentReturned`（`:2105–2118`）：只有携带 spawnId 的信号才能移除**自己的**条目；id-less 信号（Claude 的 task-notification result、Codex 的外层 turn/completed）在 `:2116` 直接 `return`。注释 `:2110–2115` 明确记载旧 FIFO pop 曾杀掉仍运行 sub-agent 的槽位。
- `onSubagentToolResult`（`:2137–2152`）：精确匹配移除；backgrounded 的占位 ack 不排空集合（`:2143–2146`）。
- `drainFinishedBackgroundTasks`（`:2090–2103`）：只在 live set **整体为空**时才释放。注释 `:2058–2082` 记载旧实现在 22 秒内进出 live set 二十余次，导致误收敛。

三条教训：

1. **计数器不是账本。** 一个不能指明"它关闭的是哪一条"的信号，不该关闭任何一条。
2. **失败在结构上不可表示**（见 § 5.2）。
3. **失败是放大器，不是原因。** 一个失败的 sub-agent 会**早期**上报，正好它的兄弟还在跑——也就是存在可抢的活槽位。**这个 bug 在成功路径里同样存在**，失败只是让它更容易被看见。

5.2 不可表示的状态

症状：用户报告 sub-agent 死了之后主进程行为异常，但代码里**找不到任何**一处能表达"这个 sub-agent 死了"。

根因：两个事件映射器都把 sub-agent 的终态**硬编码为** `completed`。在记账层，**不存在一个值表示"这个死了"**。

修复：终态不再硬编码，且 fail-closed：

- `codex_event_mapper.dart` 的 `_turnOutcome`（`:561-573`，M-0532）：`completed → success`、`failed → failure`、`interrupted → interrupted`、`inProgress` 及**任何未知值 → `failure`**（`:557-560`）。
- `claude_event_mapper.dart` 的 `_turnOutcome`（`:335-344`）：`success → success`、`timeout → timeout`、其余 → `failure`。
- `claude_event_mapper.dart:266-306`（M-1372）：此前把 origin 为 task-notification 的 result 特判为"非终态进度事件"的分支已被移除——现在每个 result 都是正常的 `AgentTurnCompleted` 并携带 `originKind`，由消费者按"是否仍有未完成工作"来 gate。注释 `:272-274` 记录了旧 bug 的用户可见症状：*"HTML 做成了也不结束"*。
- 诚实兜底仍保留：settle 超时触发的收尾**不算正常成功**（`_onSubagentsAbandoned`，`:470-475`）。

教训：**你无法处理一个你无法表示的状态。** 用户的报告字面上是对的——不是主进程"不知道怎么办"，而是在那一层**没有词**能描述这件事。这是一个类型系统层面的教训：当一个领域状态在类型里没有对应的值，所有处理它的代码都必然是错的。

5.3 只有全新设备才能看见的 bug

症状：走订阅登录路径完成 onboarding 后，composer 完全不渲染模型选项。

根因：该路径没有持久化 active provider。**任何已经配置过的设备都看不见这个 bug**——它们的 active provider 早已落盘。

教训：**新设备不是中性设备。** 它是一部具有特定历史（"空"）的设备，恰好隐藏了"依赖已有状态才暴露"的那类 bug。

5.4 只有用过的设备才能看见的 bug

症状：在日用机上每次必现，在开发机上**从不复现**。

根因：chat 表面已经学会了在切换运行时**前让出单引擎槽位**；但 Settings 与 onboarding 认证路径从没学过。开发机经常清数据，引擎从不"热"，槽位总是空闲——bug 不可能发生。日用机上引擎常常是热的，槽位被占用，每次都撞上。

教训（与 §5.3 对称）：**清过数据的设备不是中性设备。** 它恰好隐藏了"依赖已有状态才暴露"的那类 bug。长期本地状态应当与不可信输入一样，接受对抗式审视。

5.5 没人记下的那次失败

症状（2026-09-15，首个公开构建的前一天）。一部手机开着 VPN，跑在代理模式。应用迟迟没有发现有新版本发布，而同一部手机上内置的 CLI 联网一切正常。半个小时里，这两个事实并存，日志里什么都没有。

根因。 同一个应用里存在两条网络路径、两套策略。Kotlin 侧会把探测到的代理（`HTTPS_PROXY`、`NO_PROXY`）导出到每个 CLI 子进程的环境变量里（`NetworkEnv.kt:115`，`buildAuthProxyEnv`），所以智能体在代理背后能正常工作。而应用自己的 Dart 客户端——更新检查、许可证 API、模型发现——都是裸的 `HttpClient()`（`update_transport.dart`，修复前 `:23`），对代理一无所知，直接拨号，直连被拒。更糟的是后半段：更新检查的失败分支返回一个 `failure` 值，然后什么都不写（`update_service.dart`，修复前 `:128`）——没有日志、没有诊断事件、没有时间戳。设备没离线，服务器没挂，唯一的痕迹就是"没有痕迹"。

修复（S18-A/C，commits `76540347`、`363f77ca`）：一个进程只有一套代理策略——一个共享的 `AppHttpClientFactory`，它的 `findProxy` 跟随 CLI 拿到的那份快照（`app_http_client.dart:17-48`、`:243-260`），并立下"拒绝回退"的规矩（配了代理却不可达，就是错误，绝不静默切回直连）；外加一个分类过的失败（`classifyFailure`，`update_service.dart:635`）——输出一行日志和一条诊断事件（`:569`），并把最近尝试时间、最近成功时间、失败类别暴露在设置页里。

两条教训：

1. **一个进程，一套网络策略。** 当某个子系统长出自己的传输层时，它继承的环境并不是进程其余部分所观察到的环境。bug 不在任何一个客户端里，而在于存在两个客户端。
2. **一条静默的失败路径，对运维者来说就是一个不可表示的状态。** §5.2 讲的是代码说不出的状态；这一条是同一个缺陷高了一层：代码说得出来（`failure`），但人看不见。证据的缺失，在三十分钟里被当成了"没有事件"。

5.6 一处入口执行的协议等于没执行

引擎槽位纪律**只存在于它被发现的地方**（chat 表面），从来没有被写到一个两个入口都能看见的位置。Settings 与 onboarding 因此各自独立地违反了它（§5.4）。

教训：协议必须写在所有执行者都能看见的层级。 在一个入口修好一个不变式，不等于这个不变式成立。跨表面的契约应该有单一的权威定义点，否则它只是"某个文件里的注释"。

5.7 一个反直觉的总结

注意 §5.1 – §5.6 的共性：**没有一个是架构问题。** 嵌入 Linux 用户态、对抗后台管控、中介权限——这些"听起来很难"的问题都被解决了。真正造成用户可见故障的，是**记账、表示与可观测性**：

- 一个信号关闭了它不拥有的条目（§5.1）
- 一个领域状态在类型里没有值（§5.2）
- 一个不变式只在部分入口成立（§5.4、§5.6）
- 测试设备的"空历史"本身就是一个会隐藏 bug 的状态（§5.3、§5.4）
- 一个进程里跑着两套网络策略，而失败路径不产生任何输出（§5.5）

对研究者的意义：这类失败不会出现在任何 benchmark 里。SWE-bench 式的评测测的是"智能体能多正确地解决一个孤立任务"，而记账失败是**随时间积累、跨会话、依赖设备历史**的。它们只能被真实使用暴露，只能被对抗式审视发现。这是一个评测方法上的空白。

6 五英寸屏上的 CLI 智能体

一个为 27 英寸显示器和终端设计的智能体，放到五英寸屏上，每一个设计决定都要重做。Keva 的做法是 **Command Transparency**：把智能体做的每一件事都结构化地呈现给用户——worklog（工作日志）、diff（代码差异）、web（搜索结果）三类卡片。

诚实规则。当一个 turn 结束时仍有 sub-agent 未结清，旧界面会把它们**全部标成 done**。这看起来是"友好的默认值"，实际上是危险的：一个外观"已完成"的行，对应一个结局未知的任务，比没有行更糟。

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

推广到任何智能体 UI：**界面必须能说"我不知道"**。一个只能渲染"成功"与"失败"的表面，会把不确定性画成成功——因为"未知"无处可放，它必然塌缩到二值的一侧。在 §5.2 里，记账层的"不可表示"最终就是以"被画成成功"的形式被用户看见的。表示层与 UI 层的失败是**同一个失败的两端**。

7 它扛得住吗

先声明本文的评估立场：这不是一张 benchmark 表。我们没有与基线对比，原因有三：⑴ 端侧 coding agent 目前没有可比的公开基线；⑵ 真正的失败模式（§5）随时间与设备历史积累，单次 benchmark 测不出来；⑶ 我们认为对这类系统，"什么会让一次运行中断"比"平均快多少"更有信息量。

因此 §7 提供两类东西：**可核验的超时预算清单**（全部是代码里的真实常量），以及**已有的真实日志证据**。

7.1 长程预算：一份"什么能中断一次运行"的清单

这些不是性能指标，而是**系统的失效边界**。每一项都能指到代码行：

机制	值	位置	附注
turn 硬超时	12 小时	`chat_controller.dart:385`	绝对上限
首事件超时	180 秒	`chat_controller.dart:390`	标定依据：16 次 GLM 成功 turn 首语义事件中位 6.1s / 尾 65.5s
运行时实例 watchdog	15 秒	`chat_controller.dart:391`	
liveness gap	3 分钟	`chat_controller.dart:392`	仅对声明 `emitsHeartbeats` 的运行时武装（`:3258`）
心跳顺延上限	3 次 ≈ 12 分钟	`chat_controller.dart:1665`	进程仍活着时顺延，`:3345-3354`
idle reclaim	5 分钟	`chat_controller.dart:394`	回调受 `_canReclaimPersistentProcess` 守卫（`chat_page.dart:1401-1402`），turn 运行时状态非 idle 故不回收
sub-agent settle 兜底	10 分钟	`chat_controller.dart:291`	超时收尾不算正常成功
idle settle	15 秒	`chat_controller.dart:292`	
慢响应阈值	12 秒	`chat_controller.dart:393`	

早期的大纲曾引用一组会话数字（时长、工具调用数、RSS、一段静默间隔），但代码库里找不到任何实测记录支撑它们（附录 B）。我们不印未经实测的数字，本节给出的是可核验的清单与确实存在的日志证据。

7.2 已有的真实日志证据

- `log/wp4_batch4_2026-07-16/README.md:23`：记录一次 25.739 秒 的 long-bash turn。
- `log/android_logcat_relevant_2026-06-30.txt:1,4`：`task_progress` 报告单个 Explore sub-agent 内部 `tool_uses:33/34`（这是子任务内部计数，不是整轮会话的总调用数——两者不可混用）。
- `chat_controller.dart:3336-3340`：一段设备证据注释，记录了一次约 168 秒的静默事件及其处理路径。
- `log/android_exit_info_2026-06-30.txt:40`：`pss=155MB rss=187MB`——但这是应用被 `installPackageLI` 杀掉时的记录，不是引擎会话期的 RSS 曲线，不可引用为引擎内存占用。

结论：目前仓库中存在的证据足以支撑"系统能跑、长任务有明确的失效边界"这一定性结论，不足以支撑任何定量性能主张。要补齐定量评估，需要一次受控的真机会话采集——我们把这项列为明确的后续工作（§9）。

7.3 我们量不出来的东西

仅凭进程状态，无法区分"模型正在生成"与"连接已经卡死"——两者都表现为一个不消耗 CPU 的睡眠进程。能区分它们的只有之后发生了什么。这是一个基本观测极限，不是实现缺陷，但它意味

着：任何基于进程探针的 liveness 判定都有误报窗口，`_maxLivenessDeferrals = 3`（约 12 分钟宽容）就是为这个窗口买的保险。

8 维护方法论：一人指挥多模型的验证纪律

Keva 由一个人指挥多个模型协作开发。这套方法写在这里，是因为它对研究"LLM 辅助软件工程"的人有参考价值，不是为了炫过程。

分工：一个模型当**审计者**，不写实现代码；其他几个按**书面派工**执行，每次派工都带源码引用、既有工作、显式非目标。每个回执在被相信之前都要对照代码复核一遍。

三项做法承担了实际工作：

8.1 强制自检

执行者必须**回滚自己的修复**，确认新测试**以原始 bug 的精确症状失败**，然后再恢复。

> 一个不能被弄失败的测试不是证据。

8.2 空集检查（vacuity check）

当被问及"某个负断言是否还在守护什么"时，执行者实证了：**回滚两个比较，仍然让十个测试通过，而一次真实的错投就在眼前**。守卫已经变成了装饰。

覆盖率看不见这个；绿灯套件也看不见这个。这是测试有效性研究里一个重要现象：一个断言可以连续通过 N 年而早已不再保护任何东西。要发现它，必须主动做**突变式核验**（把被保护的东西破坏掉，看测试是否报警）。

8.3 对抗式复核

一个派生的根因判断会被交给第二个模型，指令是**反驳它**。

8.4 两个应当知道的比例

- **约三分之一的模型生成审计发现没能活过逐行核验**。生成的 review 应被当作 lead generator，永远不是 verdict。
- **审计者自己的根因判断曾被一个执行者用证据推翻**。一个测试文件的 diff 只含一行注释，被当作"线上回归"的证明；实际上是一个兄弟测试集因一次有意的行为变化更新过，这个文件被漏掉了。推理看起来合理、便宜、**错了**——这正是反驳步骤不可省的原因。

8.5 对 PR 的意义

门槛不是"测试通过"，而是"**你能证明测试在你的修复缺席时会失败**"。

9 限制与威胁

当下、具体、不打太极。

#	限制	性质
1	`chat_controller.dart` 单文件 >4000 行，是最大的可维护性债务	工程债务，待拆分
2	网络层自研 `LocalDnsServer.kt` + 代理，未用成熟 `http` / `dio`；应用层客户端现已共享代理发现（§5.5），但该层仍是手写	维护负担，错误处理面更窄
3	电池优化豁免可选，不开启则 OEM 后台管理可能干预长任务	平台依赖
4	`targetSdk = 28` 的 SELinux/后台服务套利依赖平台行为，会过期	平台依赖（§4.7）
5	无 ptrace，权限中介是应用层管道，弱于系统级拦截	架构取舍（§4.5）
6	评估不是 benchmark：无定量性能数据，无基线对比	有效性威胁（§7）
7	一个维护者，bus factor = 1	组织风险
8	地理定位仅前台，不做后台追踪	产品边界
9	在没有代理的情况下，国内部分运营商网络直连许可证与更新端点会超时（2026-09-15 在国内 5G 网络实测）；首次激活依赖这条路径	部署依赖

对评估有效性的诚实交代（威胁 #6）：本文所有定量性质的主张都限于"代码中可核验的常量"与"已有日志记录"。我们**没有**做受控实验、没有样本量、没有统计显著性。§5 的案例研究是**轶事性证据**（anecdotal），虽然每条都可核验到代码行。读者应据此调整对本结论的信任度。

10 结论

Keva 证明了一件事是可行的：**在未 root 的 Android 手机上，运行一个真实的、多运行时的 coding agent**——不是远程终端，不是 API 封装，而是在本地跑着完整 agent loop 的系统。

但本文真正想留下的，不是"我们做到了"，而是三类可迁移的知识：

1. **问题分类学**（§3.1）：把"不可能"拆成性能/许可/生命周期三类。性能通常不是问题，许可可以套利，生命周期才是硬骨头。
2. **记账、表示与可观测性的失败模式**（§5）：一个信号关闭它不拥有的条目；一个状态在类型里没有值；一个不变式只在部分入口成立；一条失败路径不留下任何证据。这些失败**只在真实使用中暴露**，现有 benchmark 完全测不到——这是一个评测方法上的空白，值得研究者投入。
3. **吸收层的选择**（§4.6）：能力不对称应在最高层（呈现层）或声明层（能力标志）吸收，而不是在调用处按运行时种类 special-case。

唯一值得在这里做的断言，因为它关于方法而非优先权：这样造——审计、派工、拒绝相信一个修复直到它先被证明会失败——比一个人单干产出了更好的代码。作为经验陈述，提供给读者自己检验，不是结论。

参考文献

官方文档与平台规范

1. Android Developers. *Background execution limits / Doze mode / App Standby*. developer.android.com.（应用后台限制的权威说明，§4.4 与 §9 限制 3 的依据）
2. Android Developers. *Protecting users with SELinux / `execute_no_trans`*. source.android.com.（§4.2 与 §4.7 中 SELinux 策略的依据）
3. Android Developers. *App resources / `targetSdkVersion` 行为变更*.（§4.7 约束套利的平台机制）
4. Android Developers. *Process and thread lifecycle / foreground services / wake locks*.（§4.4 保活机制的官方语义）
5. Flutter 团队. *Flutter 官方文档*，以及 Dart 语言的平台通道（MethodChannel）文档. flutter.dev.（§4.1 跨层通信机制）
6. Simolus3. *drift: Reactive persistence library for Flutter and Dart*. pub.dev/packages/drift.（§4.1 数据库选型）
7. PRoot 项目. *proot: chroot without root, using ptrace*. github.com/proot-me/proot.（§4.2 中被放弃的 PRoot 方案的原始来源）
8. Termux 项目. *Termux: Android terminal emulator and Linux environment*. github.com/termux/termux.（Android 上运行 Linux 用户态的先行者，本文 §4.2 的思路来源之一）
9. Anthropic. *Claude Code 官方文档*. docs.claude.com/claude-code.（§4.2 中 Claude CLI 的运行时行为与 `--include-partial-messages` 选项）
10. OpenAI. *Codex CLI / app-server 文档*. github.com/openai/codex.（§4.3 第二运行时）

学术文献

11. Yao, S. et al. (2023). *ReAct: Synergizing Reasoning and Acting in Language Models*. ICLR 2023.（工具使用型智能体的基础范式，§3.2 的背景）
12. Shinn, N. et al. (2023). *Reflexion: Language Agents with Verbal Reinforcement Learning*. NeurIPS 2023.（智能体自我修正，与 §8 自检纪律的方法论关联）
13. Jimenez, C. et al. (2024). *SWE-bench: Can Language Models Resolve Real-World GitHub Issues?* ICLR 2024.（§5.7 中"现有 benchmark 测不到记账失败"的方法论对照）
14. Liu, X. et al. (2024). *AgentBench: Evaluating LLMs as Agents*. ICLR 2024.（智能体评测的现有范式与其局限）
15. Wang, L. et al. (2024). *A Survey on Large Language Model based Autonomous Agents*. Frontiers of Computer Science.（智能体系统综述，§3 与 §4 的定位背景）

> 说明：本文遵循"诚实引用"原则——只列在写作过程中实际参考过的资料，不做位置对比，不主张任何优先权。学术文献用于说明方法背景；官方文档与开源项目是工程事实的来源。

附录 A：验证映射表

本文的每一条技术主张都映射到代码库中的具体位置。这是本文对"同行评审"的替代——读者不用问人，自己就能核验。

#	论文主张	代码位置
A1	targetSdk = 28 是为绕过 SELinux 的有意设计	`android/app/build.gradle.kts:98-103, 104, 153`
A2	Claude CLI 不走 PRoot（与 Bun JIT 冲突），改用 ld-linux 桥接	`ProcessManager.kt:30-32`；`buildProotCommand` `:2664-2668`、`:3467`；`ProcessManagerProotTest.kt:29/48/67`
A3	rootfs 安装布局	`InstallManager.kt`；`StorageCleanManager.kt:100`（`files/mcagent-runtime/`）
A4	setsid 启动引擎，活过应用进程	`EngineOrphanGuard.kt:15`；`ProcessGroupAndChildrenTest.kt`（F6/F7）
A5	杀前四核验，fail-closed	`EngineOrphanGuard.kt:104-113`（核验）、`:28-29`（fail-closed）、`:96-101`（reclaimOrphan）、`:186-224`（AndroidProcessProbe，starttime=字段22，从最后 `)` 解析 `:199-209`，uid 来自 `Os.stat` `:221-223`）
A6	watchdog 常量	`chat_controller.dart:385`（12h）、`:390`（180s）、`:391`（15s）、`:392`（3min）、`:394`（5min）、`:291`（10min settle）、`:292`（15s idle settle）、`:393`（12s 慢响应）
A7	liveness 只对声明心跳的运行时武装	`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`；顺延上限 `:1665`
A8	流式粒度在呈现层吸收	`live_text_pacer.dart:3-9`；`chat_page.dart:5721-5722`；M-0533 关闭 `--include-partial-messages`：`chat_controller.dart:3334`、`agent_event.dart:314-315`
A9	handoff 两种解析时机	跨运行时发送时组装：`chat_controller.dart:2845-2882`（`assembleForSession` `:2857`）、`handoff_brief_service.dart:8-13`；同运行时切换时暂存：`chat_page.dart:4099-4142`，武装点 `:1666-1669, 2058-2061, 8423`，发送时拼接 `chat_controller.dart:2876-2881`
A10	账本按身份关闭	`_pendingSubagentIds` `chat_controller.dart:1604`；`onSubagentReturned:2105-2118`（id-less 信号 `:2116` return；旧 FIFO pop 的记载 `:2110-2115`）；`onSubagentToolResult:2137-2152`；`drainFinishedBackgroundTasks:2090-2103`（旧实现误收敛记载 `:2058-2082`）
A11	终态 fail-closed	`lib/core/codex/codex_event_mapper.dart:557-573`（M-0532）；`claude_event_mapper.dart:335-344`、`:266-306`（M-1372，旧症状 "HTML 做成了也不结束" `:272-274`）；settle 收尾不算成功 `chat_controller.dart:470-475`
A12	idle reclaim 受状态守卫	`chat_controller.dart:3722-3734`（武装）、`:3656-3657`（调用点）；`chat_page.dart:1401-1402`（守卫）
A13	依赖选型	`pubspec.yaml:16-17`（Riverpod）、`:20-21`（drift）、`:22`（secure storage）、`:37`（archive）、`:38`（ffi）、`:61-63`（geolocator，仅前台）
A14	保活服务	`McagentKeepAliveService.kt:18-24`（前台服务 + PARTIAL_WAKE_LOCK + START_STICKY）
A15	存储清理默认拒绝	`StorageCleanManager.kt:10-23`（默认拒绝白名单、不跟随符号链接）

#	论文主张	代码位置
A16	真实日志证据	`log/wp4_batch4_2026-07-16/README.md:23`（25.739s turn）；`log/android_logcat_relevant_2026-06-30.txt:1,4`（tool_uses 33/34）；`chat_controller.dart:3336-3340`（168s 静默事件）；`log/android_exit_info_2026-06-30.txt:40`（pss=155MB，非引擎会话 RSS）
A17	架构规模	`lib/` 各层行数见 §4.1；`chat_controller.dart` >4000 行
A18	网络层自研	无 `http`/`dio` 依赖（`pubspec.yaml`）；`LocalDnsServer.kt`
A19	一个进程一套代理策略；失败被分类并记录	`NetworkEnv.kt:115`（`buildAuthProxyEnv`）、`MainActivity.kt:136`（`getProxyConfig` 通道）；`lib/core/net/app_http_client.dart:17-48`（解析器）、`:243-260`（工厂，`withProxyRetry`）；`lib/modules/update/update_service.dart:635`（`classifyFailure`）、`:569`（日志行）；修复前状态：`update_transport.dart:23`（`HttpClient.new`）与 `update_service.dart:128`（静默 `failure`），见 commit `76540347^`

附录 B：本文与早期大纲的差异

本文取代 `doc/paper/Keva_Outline_v1_*`（2026-08-16，Opus 起草）。相对该大纲，本文做了以下经核实的修正：

大纲主张	事实	本文处理
"五层架构 UI → Riverpod → modules → core"	`lib/` 实际 4 个顶层目录，Riverpod 在 `core/providers/` 内	§4.1 改为四层
"通过 PRoot 嵌入完整 glibc 用户态"	Claude CLI **不跑在 PRoot 里**（Bun JIT 冲突），用 ld-linux 桥接	§4.2 精确化
"sub-agent 终态在两个映射器中仍硬编码为 completed"	**已修复**：fail-closed（M-0532），账本按身份关闭（M-1363/M-1372）	§5.1、§5.2 改为"过去不可表示 → 现在已可表示"
"一次真实会话：11 分钟 / 32 次工具调用 / RSS 146 – 155 MB / 5.5 分钟静默"	仓库中**无任何实测记录**支撑这些数字	§7 改为可核验预算清单 + 已有日志证据
"handoff：一个在切换时解析，另一个在发送时"	实为**同运行时 vs 跨运行时**之别，非两个运行时之别	§4.6 实例三区分两分支

附录 C：v1.1 的变化（2026-09-17）

章节	变化	原因
页眉、§ 1.3、§ 4.2	包体积 600 MB → 746 MB（release 2026.38.5）；§ 4.2 的"约 600 MB"改为"约 750 MB"	实测产物
§ 4.3	目录树更正：`codex_event_mapper.dart` 位于 `lib/core/codex/`，不在 `adapters/` 下	验证映射的完整性
§ 4.7	`targetSdk` 注释改为逐字引用；六小时规则精确化（Android 15、`dataSync` / `mediaProcessing`、API 35+）；记下 Play Protect 提示这一可见代价	准确性
§ 5.5（新增）	第六个失败案例：一个进程两套网络策略，以及一条什么都不记的失败路径（2026-09-15）	v1.0 之后、发布测试期间暴露
§ 5.6 – 5.7、§ 1.4、§ 10	重新编号；"记账与表示"扩展为"记账、表示与可观测性"	与 § 5.5 一致
§ 7.1	早期大纲会话数字的免责声明缩短，并指向附录 B	可读性
§ 9	限制 2 更新；新增限制 9（无代理时运营商直连超时）	2026-09-15 实测
附录 A	A11 路径写全；新增 A19 对应 § 5.5	可核验性