DiPECS 可行性研究报告

——端侧 AIOS 的数据链路、治理边界与实现路径

可行性研究报告

项目名称：DiPECS

作者单位：DiPECS Team

报告类型：可行性研究报告

完成时间：2026 年 4 月 15 日

目录

1 引言 **3**

2 项目目标与最小闭环 **4**

3 技术可行性分析 **4**

3.1 用户行为信息采集可行性 4

3.2 时间、地点与设备状态采集可行性 5

3.3 本地脱敏与上下文整理可行性 6

3.4 云端大模型与 skills 判断可行性 6

3.5 Skills 自迭代可行性 7

3.6 Rust 核心逻辑可行性 7

4 技术栈、工程实现与验证环境 **8**

4.1 工程实现可行性 8

4.2 开发环境与验证平台可行性 10

5 决策与动作执行可行性 **10**

5.1 RuleBased 基线 10

5.2 统计预测模型 11

5.3 云端 LLM 后端 11

5.4 动作执行可行性 11

6 验证与评估可行性 **12**

7 进度与风险分析 **13**

7.1 阶段计划 13

7.2 风险分析 13

7.2.1 权限与隐私风险 14
7.2.2 网络与云端服务风险 14
7.2.3 Skills 自迭代风险 14
7.2.4 动作收益不确定风险 14
7.2.5 系统复杂性和项目范围风险 14

8 局限性与展望 **15**

9 结论 **15**

DiPECS 可行性研究报告

——端侧 AIOS 的数据链路、治理边界与实现路径

摘要： **目的：**评估 DiPECS 端侧 AIOS 原型在 Android/Linux 环境下，围绕本地采集、本地整理、云端 skill 判断、本地优化与 skill 自迭代构建最小闭环的技术可行性。**方法：**从用户行为采集、设备上下文采集、本地脱敏与上下文整理、云端大模型与 skills 判断、skills 自迭代、Rust 核心实现、决策与动作执行、验证环境以及风险等方面进行论证。**结果：**Android 官方 API、Kotlin + Rust 技术栈、结构化事件模型与离线 replay 足以支撑可运行的端侧原型；云端 LLM 可通过 skills 完成意图判断，skills 自迭代和真实动作收益需后续逐步验证。**结论：**DiPECS 的最小闭环具备可行性，应优先保证本地数据链路、隐私边界、策略治理和可审计执行正确，再逐步扩展采集源、skills 和动作桥接。

关键词： 可行性分析；端侧智能系统；隐私空气隙；动作治理；Android 公开 API

1 引言

DiPECS 的目标不是实现一个通用聊天助手，而是在 Android/Linux 场景下验证端侧 AIOS 的核心机制：系统如何从多源事件中获得上下文，如何在本地完成隐私脱敏，如何把上下文交给本地规则或云端模型进行意图判断，以及如何确保模型建议不会绕过本地策略直接执行。

参考可行性研究方案，DiPECS 的闭环可概括为：本地采集、本地整理、云端大模型调用 skills 判断、本地优化、skills 自迭代。其中，本地采集与整理负责把 Android 官方 API 和系统状态转换为结构化、脱敏后的事件流；云端大模型根据上下文选择并调用合适的 skills，由 skills 输出候选动作、置信度和风险等级；本地策略模块对云端输出进行校验后，仅执行低风险的预加载或资源准备动作；执行反馈与用户后续行为再回流到 skill 迭代链路，用于优化触发条件和参数。

当前项目采用 Kotlin + Rust 的工程路线实现上述闭环。Kotlin 负责 Android 应用层能力，包括权限申请、通知监听、前台服务、设备状态采集和动作桥接；Rust 负责统一事件模型、隐私空气隙、窗口聚合、决策路由、策略审查、动作生命周期、离线 replay 和审计记录。两侧通过 JSONL/schema 边界传输结构化数据，使同一套 Rust 核心逻辑既能处理真实 Android trace，也能处理离线测试夹具[1]。

系统主链路可以概括为：`RawEvent` → `PrivacyAirGap` → `SanitizedEvent` → `StructuredContext` → `DecisionRouter` → `PolicyEngine` → `ActionLifecycle` → `AuditRecord`。这条链路

中的每一步都有明确输入输出，因此项目可以通过分阶段开发方式逐步实现：先跑通离线 replay 和本地规则，再接入 Android 公共 API trace，最后补充云端模型、Android bridge 和更复杂的预测后端。

2 项目目标与最小闭环

本项目的首要目标是完成一个最小可运行闭环，具体包括以下步骤：

(1) 在用户授权下采集 Android 设备上的用户行为和基本上下文信息；

(2) 在本地对采集到的信息进行整理，去除或弱化其中的隐私内容；

(3) 将脱敏后的结构化信息发送给云端远程大模型，由大模型调用 skills 进行判断；

(4) 根据模型和 skills 返回的结果，在本地执行低风险的预加载或缓存优化；

(5) 记录从原始意图到云端判定再到本地执行的完整逻辑流转轨迹，使得系统可对特定轨迹进行状态回放。

该闭环的设计原则是：云端只输出结构化建议，本地保留最终执行权；模型能力用于场景理解与 skill 编排，而隐私边界、策略审查和动作治理由本地 Rust 核心确定。这样既能利用远程大模型的语义理解能力，又能避免让大模型直接控制设备。

3 技术可行性分析

3.1 用户行为信息采集可行性

Android 平台已经提供了多类官方能力，可用于构建行为采集层：

- `UsageStatsManager` 可观察应用使用情况和前后台切换趋势[2]；
- `NotificationListenerService` 可在用户授权后监听通知事件，例如通知来源、类别、出现时间和撤销时间[3]；
- `MediaStore`、`ContentObserver` 或文件监听机制可用于观察媒体和部分文件变化；
- `AccessibilityService` 可在明确授权下补充界面级交互信息，例如窗口变化、点击目标、页面切换或文本变化[4]；

- 前台服务、WorkManager 或本地数据库可用于维持采集任务、缓存事件和等待网络可用时上传结构化上下文。

这些能力都属于 Android 官方开放接口，适合在 Android Studio Emulator 或 AOSP Emulator 等虚拟环境中先行验证，再根据需要补充物理设备测试。DiPECS 当前把公开 API 作为默认主路径，把 Accessibility 等高权限来源作为后续增强方向，以降低部署和合规风险[5]。

3.2 时间、地点与设备状态采集可行性

除用户操作事件外，DiPECS 还需要获取部分基本上下文，用于帮助云端大模型判断用户意图。这部分信息同样可以通过 Android 官方能力获取：

- **时间信息**：系统时间、星期、时区、节假日标记、当前时间段等，可直接由系统 API 获取；
- **地点信息**：在用户授权下，可通过 Android Location API 获取经纬度，或进一步转换为家、学校、公司、通勤途中等粗粒度地点类别；
- **网络状态**：通过 `ConnectivityManager` 获取 Wi-Fi、蜂窝网络、离线状态等信息；
- **电量与充电状态**：通过 `BatteryManager` 获取当前电量、是否充电、是否低电量等状态[6]；
- **设备运动状态**：可选用加速度计、陀螺仪或 Activity Recognition API 判断静止、步行、乘车等粗粒度状态；
- **系统环境状态**：可采集屏幕开关、耳机连接、蓝牙连接、音量模式、勿扰模式等与用户行为有关的上下文。

为降低隐私风险，项目展示阶段可以只使用粗粒度位置或模拟位置，精确经纬度不上传。这些上下文与行为事件共同构成 `StructuredContext` 的输入，使云端判断具备足够但不过量的信息。

3.3 本地脱敏与上下文整理可行性

云端远程大模型需要足够上下文才能判断用户意图，但原始用户信息中可能包含隐私内容。DiPECS 采用本地确定性脱敏：`RawEvent` 只存在于 collector 到 core 的短路径中，经过 `PrivacyAirGap` 后，后续模块只处理 `SanitizedEvent`。

具体脱敏策略包括：

- 将原始事件转化为统一 schema，例如 `event_type`、`timestamp`、`source_app`、`foreground_app`、`notification_category`、`location_type`、`network_state` 等；
- 对通知正文、界面文本、文件名等敏感内容默认不上传，必要时只上传摘要类别或脱敏标签；
- 将精确经纬度转化为粗粒度地点类别；
- 对连续事件按时间窗口聚合，减少上传频率和网络消耗。

这部分逻辑由 Rust 实现，能够提供稳定的数据结构、严格的错误处理和较好的性能，也便于后续做确定性回放和安全审计。这种设计牺牲一部分语义完整性，但换来清晰的隐私边界和可测试的泄漏防护。

3.4 云端大模型与 skills 判断可行性

在 DiPECS 中，本地端采集并整理上下文，云端远程大模型负责选择和调用合适的 skills，由 skills 对用户下一步可能行为进行候选判断、概率评估或优先级排序。这里的 skill 可以理解为一个可解释、可复用、可迭代的行为判断单元；在工程实现上，skill 可被设计为一组具有明确输入输出接口的判断模块，其内部可采用规则、统计特征或轻量逻辑模型，对特定场景下的候选行为进行评分或判定[7]。

云端大模型适合承担以下任务：

- 根据短时间窗口内的用户行为事件和上下文识别当前场景；
- 选择一个或多个候选 skills，并为它们进行迭代；
- 调用 skills 完成具体判断；
- 对多个 skill 的结果进行排序、合并或裁决，形成最终建议；

- 返回结构化判断结果，包括被调用的 skill、置信度、理由摘要、建议预加载对象和风险等级。

当前实现中，`CloudLlmBackend` 是可选后端：只有配置完整且 router 判断上下文复杂度足够时才调用；失败后回退到 `RuleBased`，并由 circuit breaker 防止连续错误放大。本地端只接受结构化输出，不直接执行自然语言指令；`PolicyEngine` 负责校验模型和 skill 输出是否合法、是否超过风险等级、是否满足权限条件和冷却时间。这样既能利用远程大模型的理解与编排能力，又能避免让大模型直接控制设备[8;9]。

3.5 Skills 自迭代可行性

DiPECS 中的 skills 不应是一次性写死的规则，而应能根据用户历史行为和系统反馈持续迭代。可行的自迭代方式包括：

- 根据用户偏好更新 skill 参数，例如常用应用包名、常用时间段、固定地点类别、常见操作链路；
- 基于历史 trace 统计某个 skill 的触发成功率，即预测正确率；
- 将低置信度或冲突场景沉淀为待优化样本，后续由大模型总结成新的 skill 草案，经过校验再更新到本地。

为了保证安全，自迭代不应直接生成可执行系统动作，而是优先更新 skill 的描述、触发条件、上下文字段权重和候选预加载对象。所有新增或修改后的 skill 仍需经过白名单、风险等级和本地策略校验，才能进入在线判断链路。当前系统通过 `AuditRecord` 和 replay 机制为后续 skill 迭代保留了反馈入口，但自迭代本身属于后续阶段目标，不作为最小闭环的前提。

3.6 Rust 核心逻辑可行性

项目核心逻辑适合用 Rust 实现，原因主要包括：

- 行为事件流、上下文窗口、模型请求、skill 调用结果和优化策略都适合用强类型结构表达，Rust 能减少隐式状态错误；

- 本地端虽不运行模型，但需要低开销地完成事件聚合、脱敏、网络请求封装、策略校验和日志回放，Rust 适合移动端性能敏感路径；
- 策略执行需要严格区分 “模型或 skill 建议” 和 “本地可执行动作”，Rust 可以通过类型和模块边界限制高风险动作进入执行链路；
- 日志回放、确定性测试和安全审计适合以 Rust 核心库形式沉淀，方便在 PC 端和 Android 端复用。

工程上，Kotlin 负责采集 Android 事件和上下文，序列化后传入 Rust；Rust 完成聚合、脱敏和请求构造；云端大模型调用 skills 并返回判断结果后，Rust 再进行策略校验；最终 Kotlin 根据 Rust 给出的安全动作调用实际 Android API。这种分工使核心逻辑可以脱离 Android UI 进行确定性测试，即使物理设备能力尚未完全接入，系统仍可通过 replay 验证隐私、路由、策略和审计链路。

4 技术栈、工程实现与验证环境

4.1 工程实现可行性

DiPECS 的架构可以拆为六层，表 1 给出了各层的代表模块与职责。

表 1: DiPECS 分层设计

层级	代表模块	主要职责
采集层	Android collector, aios-collector	采集应用切换、通知、屏幕、电量、进程资源和 JSONL trace。
隐私层	PrivacyAirGap	将 raw event 转为 sanitized event，丢弃通知正文、文件路径等敏感信息。
上下文层	WindowAggregator	按时间窗口构造 StructuredContext 和 ContextSummary。
决策层	DecisionRouter, RuleBased, CloudLlm	根据复杂度、隐私分数和后端状态选择本地或云端决策。
治理层	PolicyEngine, ActionLifecycle	检查 capability、风险、置信度、target 和 action schema。
执行/验证层	aios-action, OfflineAdapter, replay	执行低风险动作或离线模拟，生成 audit record 和 hash。

这种分层的可行性来自两个事实。第一，Android 和 Linux 已经能提供足够的低粒度事件，支持最小原型；第二，Rust 中的核心模块不依赖 Android UI，可以用离线 trace 进行确定性测试。即使 Android 物理设备能力尚未完全接入，系统仍可通过 replay 验证隐私、路由、策略和审计链路。

工程实现按四个部分推进，每一部分都有相对明确的平台支持或实现边界，因此具备逐步落地的条件：

(1) **本地采集部分**：由 Kotlin 接入 Android 官方能力，采集用户行为和基本上下文信息。该部分不需要绕过系统权限或修改系统底层。

(2) **本地整理与脱敏部分**：由 Rust 统一事件格式，对采集到的信息进行聚合、过滤和隐私脱敏。该部分输入输出边界清晰，适合用 Rust 实现并通过 trace 回放验证结果是否稳定。

(3) **云端判断部分**：将结构化上下文发送给远程大模型，由大模型选择并调用 skills，返回候选动作、置信度、风险等级和建议优化内容。该部分本地只需要提供结构化上下文，复杂的意图理解放在云端大模型上完成，降低了移动端计算压力。

(4) **本地优化与反馈部分**：由 Rust 校验云端返回结果，再由 Kotlin 执行低风险优化，同时记录日志、trace 和用户后续行为。该部分只涉及预加载、缓存等低风险动作，不直接替用户执行敏感操作，因此实现和验证成本相对可控。

4.2 开发环境与验证平台可行性

本项目可采用以下开发与验证环境：

- **Android Studio**：用于 Kotlin 应用开发、权限调试、模拟器运行和 UI 展示[10]；
- **AOSP Emulator**：用于更接近原生 Android 行为的系统级验证，适合减少厂商 ROM 干扰；
- **Rust Android target**：例如 `aarch64-linux-android`，用于构建 Android 端 Rust 核心库；
- **adb / logcat**：用于部署、日志采集和行为验证；
- **云端大模型 API**：用于接收结构化上下文、结合 skills 并返回结构化判断结果[11]。

DiPECS 采用离线 replay 作为主要验证手段。Replay 不访问真实 Android、不访问网络，也不执行真实系统动作，而是读取 JSONL trace，经过相同的 Rust pipeline，生成 intent、policy decision、execute record 和 audit hash。这使项目可以在没有物理设备或云端的情况下持续验证核心逻辑。

5 决策与动作执行可行性

5.1 RuleBased 基线

RuleBased 是项目第一阶段后端的首选方案。它不依赖网络、不需要训练数据，行为可解释，适合作为默认降级路径。当前合理边界是：RuleBased 只生成低风险、能力范围内的动作，例如 `NoOp`、`KeepAlive` 和 `ReleaseMemory`。它不应直接生成 `PreWarmProcess` 或 `PrefetchFile`，因为这些动作需要更明确的 target 语义和更高能力等级。

RuleBased 的价值不在于预测能力最强，而在于建立系统基线：给定同一 `StructuredContext`，它应产生稳定的 `IntentBatch`，并且不应持续触发 capability denial。这个基线为后续云端模型和统计模型提供对照。

5.2 统计预测模型

在 RuleBased 之后，最自然的增强是统计转移模型。移动行为预测研究表明，用户应用切换存在明显时序规律[12–15]。项目可以从 replay trace 或真实 Android trace 中统计 foreground app 之间的转移概率，形成 $P(a_{t+1} \mid a_t)$ 或 $P(a_{t+1} \mid a_{t-k}, \ldots, a_t)$。

统计模型实现成本低，推理开销小，且容易解释。它可以作为 LocalEvaluator 的第一版：在上下文中输出 top-k 候选 app 和置信度，再由 `PolicyEngine` 判断是否允许 `KeepAlive` 或更高级动作。若统计模型不能明显优于 RuleBased，复杂序列模型也不应过早投入。

5.3 云端 LLM 后端

云端 LLM 的可行性体现在语义理解和规则编排上。对于包含多个 semantic hint 的复杂上下文，LLM 可以基于结构化 JSON 生成 `IntentBatch`，并给出风险等级、置信度和 rationale tags。当前设计中，CloudLlm 是可选后端：只有配置完整且 router 判断上下文复杂度足够时才调用；失败后回退到 RuleBased，并由 circuit breaker 防止连续错误放大。

这种设计降低了云端依赖风险。网络不可用、API 失败或模型输出格式错误都不会阻塞本地链路。但同时也说明，云端模型不能作为唯一控制面；所有云端输出仍必须经过本地策略和动作生命周期。

5.4 动作执行可行性

动作执行是项目风险最高的部分。相比预测 “用户可能打开某个应用”，执行 “预热、保活、预取、释放内存” 等动作会直接影响系统资源。为了降低风险，DiPECS 将动作分为建议、授权和执行三个阶段。表 2 总结了各动作类型的可行性与限制。

表 2: 动作类型的可行性与限制

动作	当前可行语义	主要限制
NoOp	不做动作，只记录系统观察结果	无直接收益，但适合安全降级。
KeepAlive	对已出现 app 或系统任务做轻量保活提示	Android 普通应用能做的真实保活能力有限。
ReleaseMemory	释放系统或缓存类资源，降低压力	package 级释放需与 Android bridge 能力对齐。
PreWarmProcess	预热应用或本系统资源	第三方 app 预热受平台限制，不能默认承诺。
PrefetchFile	对 url/uri 或可访问内容做预取	需要明确 target 权限和 policy 放行规则。

因此，最可行的执行路线是先用 OfflineAdapter 验证 action lifecycle，再逐步接入 Android bridge。真实执行层应优先支持 Android-safe 的动作，例如本应用资源预热、公开 content URI 预取、cache trim 或用户可见通知提示。对第三方进程强制预热、系统级内存调度等能力，应明确标记为 root/AOSP 阶段目标。

6 验证与评估可行性

DiPECS 采用离线 replay 作为主要验证手段。Replay 不访问真实 Android、不访问网络，也不执行真实系统动作，而是读取 JSONL trace，经过相同的 Rust pipeline，生成 intent、policy decision、execute record 和 audit hash。这使项目可以在没有物理设备或云端的情况下持续验证核心逻辑。

移动端预加载系统的评估不能只看预测是否命中，还要看启动延迟、能耗和误预加载成本。FALCON 对 app prelaunch 的评估方式可以作为后续实验设计参考[12]。评估指标应分为三组：

(1) **预测指标**：top-1/top-k 命中率、置信度校准、不同后端相对 RuleBased 的提升；

(2) **系统指标**：动作授权数、拒绝数、noop rate、policy denial reason、执行失败数、replay hash 稳定性；

(3) **资源指标**：动作触发后的启动延迟变化、CPU/内存开销、误预测导致的资源浪费率。

第一阶段应优先使用系统指标和 replay 稳定性，因为它们直接验证系统边界是否正确。只有在动作桥接更完整后，再重点评估启动延迟和资源收益。

7 进度与风险分析

7.1 阶段计划

从当前设计来看，项目可以按最小闭环逐步推进，先保证链路跑通，再扩展采集范围和 skill 类型。表 3 给出了分阶段开发路线，表中日期为报告撰写时已知的计划节点，实际执行以代码仓库迭代为准。

表 3: 阶段性开发路线

阶段	目标	可验证结果
Phase 1	打通离线 replay 和 RuleBased 基线	同一 trace 产生稳定 intent、policy 和 audit hash。
Phase 2	接入 Android public-API trace	模拟器或物理设备 JSONL 能进入 Rust pipeline。
Phase 3	完善 cloud/rule/local 后端路由	云端失败可回退，privacy-sensitive context 不上传。
Phase 4	增加统计预测后端	与 RuleBased 比较 top-k 命中率和 noop rate。
Phase 5	强化 Android-safe action bridge	真实动作有审计记录，并能量化收益与开销。

7.2 风险分析

7.2.1 权限与隐私风险

应用需要一些敏感权限，比如访问通知、位置和辅助功能，这些都必须用户主动允许。同时，收集的数据可能包含隐私信息。应对措施包括：只在官方授权入口启用采集，权限缺失时降级运行；上传云端的数据只保留必要信息，不传正文、精确位置等敏感内容；在界面上明确告诉用户哪些数据在用。

7.2.2 网络与云端服务风险

调用远程大模型依赖网络，可能会延迟或失败，影响实时体验。应对措施包括：设置请求超时和降级策略，缓存最近一次结果；只在上下文变化明显时请求模型；网络不可用时先记录事件，不阻塞用户操作。

7.2.3 Skills 自迭代风险

Skills 在根据历史行为和反馈自我迭代时，可能出现参数偏差或逻辑异常，导致行为结果不符合预期。应对措施包括：skills 自动迭代只更新非高风险参数，如描述、触发条件、参数权重和候选预加载对象，不增加高风险操作；保留旧版本的 skills，便于追踪；新版本先在观察模式下运行，确认行为稳定后再进入在线链路。

7.2.4 动作收益不确定风险

即使预测正确，动作也未必带来可感知收益。例如某些应用启动本来很快，预加载反而浪费资源。应对措施是引入 confidence threshold、预算限制和 wasted action rate，用实验数据决定哪些动作值得保留。

7.2.5 系统复杂性和项目范围风险

Kotlin 与 Rust 协作会增加接口设计和调试成本；同时，一次实现所有复杂功能容易导致项目失控。应对措施包括：核心逻辑集中在 Rust，数据传输用稳定格式，关键接口加测试；阶段目标清晰，只做意图判断、skill 调用、自迭代记录和低风险预加载；云端模型只负责选择技能和操作内容，不直接控制设备；整个执行链路可审计、可关闭。

8 局限性与展望

本可行性分析存在以下局限：

(1) 分析以公开 API 能力和项目内部实现为主，尚未在真实物理设备上完成端到端长时间运行验证；

(2) 高权限数据源（Accessibility、Binder、eBPF）的可行性判断以能力研究为主，缺少稳定部署方案；

(3) 动作收益评估依赖后续实验，当前阶段无法给出启动延迟或能耗的量化结论；

(4) 云端 LLM 和 skills 自迭代的可行性结论基于接口设计和假设场景，真实场景下的失败模式需进一步覆盖。

后续工作应在守住数据链路、隐私边界和动作治理边界的前提下，逐步扩展：稳定 Android 到 Rust 的生产入口、建立 golden trace 回归基线、接入真实云端后端、验证 Android-safe 动作桥接，并在更长周期内评估预测模型和动作执行的真实收益。

9 结论

综合分析，DiPECS 的最小闭环具备可行性。Android 官方 API 足以提供第一阶段上下文；Kotlin + Rust 技术栈可以清晰划分采集与核心逻辑；Rust 核心可以稳定承担脱敏、聚合、策略和 replay；RuleBased 可以作为本地安全基线；CloudLlm 和统计模型可以作为后续增强；skills 设计为云端判断提供了可解释、可迭代的单元；ActionLifecycle 和 audit record 能限制动作风险并提供可验证记录。

项目当前需要避免过度承诺高权限采集、系统级资源调度和 skills 自迭代的即时收益。可行的发展策略是先保证数据链路、隐私边界和动作治理正确，再逐步证明预测模型、skill 迭代和真实动作是否带来性能收益。这样既能形成可演示原型，也能保留继续扩展到 root/AOSP 环境的空间。

参考文献

[1] Kai Mei, Shiyao Zhang, Jiawei Liu, et al. Aios: LLM agent operating system. arXiv:2403.16971, 2024.

[2] Android Developers. UsageStatsManager. Android Developers, 2024.

[3] Android Developers. NotificationListenerService. Android Developers, 2024.

[4] Android Developers. AccessibilityService. Android Developers, 2024.

[5] Google Play Console Help. AccessibilityService API policy. Google Play Console Help, 2024.

[6] Android Developers. BatteryManager. Android Developers, 2024.

[7] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, Zhangyue Yin, Shihan Dou, Rongxiang Weng, Wensen Cheng, Qi Zhang, Wenjuan Qin, Yongyan Zheng, Xipeng Qiu, Xuanjing Huang, and Tao Gui. The rise and potential of large language model based agents: A survey. arXiv:2309.07864, 2023.

[8] Zhiyong Wu, Chengcheng Han, Zichen Ding, et al. OS-Copilot: Towards generalist computer agents with self-improvement. arXiv:2402.07456, 2024.

[9] OS-Copilot project page. Project Page, 2024.

[10] Android Developers. Run apps on the Android Emulator. Android Developers, 2024.

[11] DeepSeek. Deepseek API documentation. DeepSeek API Docs, 2025.

[12] Tingxin Yan, David Chu, Deepak Ganesan, et al. FALCON: Fast app launching via predictive user context. Microsoft Research, 2012.

[13] Sha Zhao, Julian Ramos, Ziwen Jiang, et al. AppUsage2Vec: Modeling smartphone app usage for prediction. IEEE Xplore, 2019.

[14] Zhihao Shen, Zhefu Yan, Zhongmin Chen, et al. DeepAPP: A deep reinforcement learning framework for mobile application usage prediction. ACM Digital Library, 2019.

[15] Tong Li, Tong Xu, Huangdong Wang, Zhen Tu, Sasu Tarkoma, Zhu Han, and Pan Hui. Smartphone app usage analysis: Datasets, methods, and applications. *IEEE Communications Surveys & Tutorials*, 24(2):937–966, 2022.