JANUS: Characterizing Bidirectional Privacy and Security Risks in LLM-Integrated Android Apps

Jiawei Guo
University at Buffalo, SUNY
Buffalo, NY, United States
jiaweigu@buffalo.edu

Zongze Li
University at Buffalo, SUNY
Buffalo, NY, United States
zongzeli@buffalo.edu

Haipeng Cai*
University at Buffalo, SUNY
Buffalo, NY, United States
haipengc@buffalo.edu

Abstract

The rapid integration of large language models (LLMs) into mobile applications has introduced a new class of privacy and security risks that existing analysis techniques are poorly equipped to address. Unlike conventional third-party service integrations, LLM integrations are bidirectional: sensitive user data flows into model endpoints as part of carefully constructed prompts, and model-generated text flows out into app logic, UI surfaces, notifications, and downstream APIs. Both directions introduce distinct risks, including data leakage, unsafe reuse of model outputs, and input-safety weaknesses, that have not been systematically studied at scale.

We present JANUS, a static analysis of end-to-end LLM interaction pipelines in Android apps. It recovers request construction semantics, identifies response-entry points and their downstream propagation, and employs a multi-agent reasoning layer to classify privacy- and security-relevant findings. We evaluate JANUS on a curated micro-benchmark suite and use it to conduct the first systematic characterization of privacy and security risks in LLM-integrated Android apps. We find that such risks are prevalent in practice, with recurring categories such as client-side secret exposure (e.g., embedded API keys and system prompts), sensitive-data outsourcing to third-party LLM services without corresponding disclosure, secondary leakage of LLM responses to downstream components, and prompt/input-safety weaknesses such as prompt-injection-prone handling. We also present representative case studies that illustrate how these risks manifest in deployed apps and what concrete harm they may enable. In line with our ethical process, we responsibly disclosed critical findings to affected developers.

CCS Concepts

• **Security and privacy → Mobile platform security**.

Keywords

Android security; LLM integration; static analysis; information flow

ACM Reference Format:

Jiawei Guo, Zongze Li, and Haipeng Cai. 2026. JANUS: Characterizing Bidirectional Privacy and Security Risks in LLM-Integrated Android Apps. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3846749

*Haipeng Cai is the corresponding author.

CCS '26, The Hague, Netherlands

ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3846749

1 Introduction

Mobile apps are rapidly becoming an important deployment surface for cloud-hosted large language models (LLMs). Across app categories, developers are beginning to embed LLM capabilities into existing mobile workflows to support features such as conversational assistance, writing help, search and summarization, personalization, translation, tutoring, and content generation. In these designs, the mobile app is no longer a passive frontend: it acts as the client-side intermediary between users and remote model services, collecting local context, packaging it into prompts or multimodal requests, and then consuming model outputs inside app workflows.

This emerging integration pattern creates a qualitatively different trust boundary from traditional mobile backends or third-party analytics SDKs. On the request side, LLM calls often incorporate rich, unstructured, and dynamically assembled context, such as user-entered text, chat history, uploaded files, profile attributes, device-derived state, or content pulled from elsewhere in the app. On the response side, model outputs are not merely displayed and discarded: they may be rendered in UI, cached locally, logged for debugging, forwarded to external services, parsed into structured objects, or used to drive downstream behaviors. Early work on LLM-integrated systems has already shown that such input/output boundaries can introduce new attack and leakage surfaces, including prompt-injection-mediated misuse, unsafe downstream execution, and other trust-boundary failures [18, 19, 24, 30, 38]. These concerns are especially consequential on mobile devices, where apps routinely sit close to sensitive personal data and long-lived user context.

At first glance, this problem may appear adjacent to a growing body of work on so-called "LLM apps." However, much of that literature studies a rather different ecosystem: custom GPTs, prompt-wrapped assistants, or LLM app-store artifacts whose main functionality *is* the LLM itself, typically customized via instructions, knowledge files, or action schemas [25, 34, 48–50]. Another nearby line of work studies LLM-integrated frameworks, web applications, and agentic systems, focusing on vulnerabilities such as prompt injection, prompt-to-SQL attacks, remote code execution, and taint-style prompt-to-anything propagation [24, 29–32, 38]. These studies are important, but they do not capture the setting we investigate here: closed-source Android apps distributed through the Play Store, whose *primary functionality remains the original app functionality*, while LLMs are embedded as assisting components inside existing mobile features and workflows.

Within the mobile domain, the closest prior studies still leave a substantial gap. Recent work on Android LLM usage has examined *open-source* Android repositories on GitHub, characterizing how developers integrate LLMs and what implementation strategies they adopt [23]. Earlier empirical work on "AI-driven mobile

apps" studied the broader adoption of machine-learning and deep-learning techniques in mobile apps, but predates the current wave of cloud-hosted LLM integration and does not model the bidirectional request/response boundary that makes LLM use distinctive [28, 45]. More recent Android-focused security work identifies specific insecure LM usage patterns, such as bypassable client-side restrictions, but does not characterize how real-world Play-Store apps construct LLM interactions, what sensitive information flows into and out of those interactions, and how prevalent the resulting privacy and security risks are across the ecosystem [26].

To fill this gap, we conduct a measurement study of real-world LLM-integrated mobile apps from Google Play. Our goal is to characterize, at ecosystem scale, how these integrations are implemented in practice, what kinds of request- and response-side information flows they create, and what privacy and security risks emerge from those flows. Doing so requires more than simple endpoint matching or conventional source-to-sink analysis: to measure this ecosystem faithfully, we need an analysis instrument that can recover the code surrounding the LLM interaction boundary. We therefore build Janus, an automated analysis pipeline that localizes LLM request sites, recovers the corresponding response-handling context, and extracts the evidence needed to audit the resulting behaviors at scale.

Building such an instrument is non-trivial because LLM interaction logic is often fragmented across modern Android abstractions. **First**, requests are frequently assembled through heterogeneous patterns—vendor SDKs, generic HTTP clients, wrappers, builders, and coroutine- or callback-mediated helper layers—so the actual prompt and its constituent data sources are rarely visible at a single call site. **Second**, responses typically re-enter app logic through asynchronous mechanisms such as callbacks, reactive streams, coroutine resumptions, or observer-based delivery, which obscures how model outputs are subsequently processed. **Third**, determining whether an observed behavior is genuinely risky often requires semantic interpretation of both sides of the interaction boundary: what data is sent, how outputs are reused, and whether the observed behavior is consistent with the app's stated disclosures. These challenges motivate the design of Janus: treat request construction as a first-class analysis target, explicitly recover response-entry and downstream context, and combine program evidence with policy evidence to support scalable risk attestation.

We evaluate Janus on a curated suite of 221 compatibility-aware microbenchmarks spanning request-realization patterns, response-flow patterns, and audit outcomes. Janus achieves 95.93% benchmark-level success, corresponding to 95.93% recall at 100% accuracy. End-to-end analysis takes 11.6 minutes per app on average, with an average LLM-auditing cost of $0.83 per app.

Applying Janus to 299 LLM-integrated Android apps yields the first large-scale picture of how cloud-hosted LLM functionality is actually embedded into closed-source mobile apps in the wild. Our findings are organized around five research questions.

RQ1: Where are LLM-integrated apps appearing? LLM integrations are not confined to chatbot-style apps. They appear across 28 Google Play categories, including sensitive-context domains such as health, finance, lifestyle, and communication, and they are globally distributed across developers from 43 countries. Most are retrofits: most of the apps predate the consumer LLM wave, indicating that LLM functionality is grafted onto mature codebases whose original privacy architecture was not designed for model-facing data flows.

RQ2: How are LLMs integrated in practice? The dominant integration style is app-controlled rather than SDK-controlled. We observe 23 distinct providers and 16 primary integration mechanisms; OpenAI is the most prevalent provider, but generic HTTP libraries dominate implementations. Nearly two thirds of apps invoke LLM endpoints through generic HTTP stacks rather than official SDKs, and 13.7% of apps embed two or more providers in the same APK. This means that prompt assembly, authentication, response parsing, and logging largely remain in app code.

RQ3: What privacy and security risks arise? Disclosure failure is the norm, not exception. We find that 294 of the 299 apps transmit LLM-bound user data that is not reflected in their privacy policy declarations. Credential exposure is also widespread: hardcoded credentials, unauthenticated runtime key distribution, and related credential-handling flaws affect a substantial fraction of the corpus. Lower-frequency but higher-impact response-side risks include forwarding model output to secondary sinks, weakened safety controls, and client-side control over endpoint or model behavior.

RQ4: What do the risky flows look like semantically? The prompt is increasingly a correlation surface. Beyond free-text input, apps send profile attributes, device identifiers, media, conversation history, document contents, and in some cases third-party content not authored by the user. On the output side, while most apps render model responses as UI text, a consequential tail persists them to local storage, parses them into machine-actionable structures, or forwards them to non-LLM backends. These patterns show that the risk lies not only in *whether* an app calls an LLM, but in the semantics of what enters and leaves that boundary.

RQ5: What do real cases reveal? Our case studies show that serious failures are rarely isolated mistakes. Instead, they reflect recurring engineering styles: client-side enforcement of paid tiers that simultaneously exposes provider keys, and unrelated subsystems that serialize user-supplied LLM credentials or model-derived content beyond their intended boundary. These cases illustrate how LLM-specific risks compound with familiar mobile design weaknesses.

In summary, this paper makes three contributions:

- We design and implement Janus, an automated end-to-end analysis pipeline for LLM-integrated Android apps (§3–§4).
- We curate a labeled micro-benchmark suite for LLM-integration analysis and evaluate Janus's effectiveness against it (§5).
- We conduct a large-scale measurement study on real-world Android apps, characterizing integration patterns, risk categories, and semantic flows, and presenting concrete case studies (§6).

2 Background and Motivation

2.1 Cloud LLM Integration in Android Apps

Cloud-based LLM functionality reaches Android apps through several common patterns. Some apps directly call vendor endpoints using generic HTTP clients such as OkHttp [5], Retrofit [6], Volley [7], or HttpURLConnection. Others use official or unofficial SDKs that abstract transport details and expose higher-level APIs. In both cases, the app remains responsible for assembling request context and consuming the returned model output.

```java
// ChatViewModel.java
public class ChatViewModel extends ViewModel {
  private final JournalRepository journal;
  private final HealthRepository  health;
  private final OpenAIClient      client;
  private final ExecutorService   io;
  private final MutableLiveData<String> _reply = new MutableLiveData<>();
  public LiveData<String> getReply() { return _reply; }

  public final void onUserMessage(final String txt) {
    this.io.execute(new Runnable() {  // executor edge
      @Override public void run() {   // anon inner cls
        List<Note> n = journal.recent(50);
        HealthSnapshot h = health.snapshot();
        Prompt p = new Prompt.Builder()
          .system(SYSTEM_INSTRUCTION) // app-defined
          .user(txt)                  // user text
          .notes(n)                   // private journal
          .health(h)                  // sensitive PII
          .build();
        Response r = client.chatCompletion(p); // LLM
        _reply.postValue(r.text);     // -> main looper
      }
    });
  }
}
```

(1.a) Request-side assembly via executor-dispatched `Runnable`.

```java
// AssistantViewModel.java + Activity
public class AssistantViewModel extends ViewModel {
  private final OpenAIClient      client;
  private final ChatHistoryDao    history;
  private final FirebaseAnalytics analytics;
  private final MutableLiveData<String> _reply = new MutableLiveData<>();
  public LiveData<String> getReply() { return _reply; }
  public final void ask(Prompt p) {
    client.stream(p, new StreamCallback() { // net cb
      @Override public void onChunk(Chunk c) {
        _reply.postValue(c.text);     // (1) LiveData
        history.insert(c.text);       // (2) Room DAO
        Bundle b = new Bundle();
        b.putString("text", c.text);
        analytics.logEvent("reply", b); // (3) Firebase
      }});
  } }
public class AssistantActivity extends AppCompatActivity {
  @Override protected void onCreate(Bundle s) {
    super.onCreate(s);
    vm.getReply().observe(this, t -> { // (4) UI observer
      chatView.append(t);
      tts.speak(t, QUEUE_ADD, null, null); // (5) TTS
      web.evaluateJavascript("r('"+t+"')",null); // JS
    });
  } }
```

(1.b) Response-side fan-out via callback and `LiveData` dispatch.

Listing 1: Two motivating examples of implicit request/response control flow beyond a conventional Android call graph.

A typical LLM request includes endpoint metadata, model configuration, message content, and possibly tool definitions or structured output instructions. The values populating these fields may originate from user input, cached app state, documents, device-derived data, or developer-authored prompt templates. Responses then flow back into app code, where they may be displayed, stored, logged, parsed, or forwarded. This end-to-end interaction boundary is the central object of analysis in this paper.

2.2 Motivating Examples

We illustrate two representative patterns that motivate Janus. Both exercise the asynchronous `ViewModel`+`LiveData` idioms [4] that fragment control flow that defeat conventional static analyses. Listing 1 shows them side by side: the request-side example on the left (Listing 1.a) and the response-side example on the right (Listing 1.b).

Example 1: Request-side leakage hidden behind ViewModel + executor-mediated assembly. Listing 1.a shows a journaling assistant whose request context is assembled only inside an executor-dispatched `Runnable`. `onUserMessage()` submits the task at lines 10–11, but the actual prompt is constructed later inside `run()` (lines 12–20), where the app combines four distinct sources: an app-defined system instruction (line 16), user text (line 17), recent journal entries (lines 13, 18), and a `HealthSnapshot` from a local repository (lines 14, 19). Only the final `Prompt` object reaches `client.chatCompletion(p)` (line 21), after which the reply is posted to `LiveData` (line 22). A conventional call-graph view therefore sees a single prompt argument but not the sources silently assembled behind the executor edge. The example illustrates request-side over-collection hidden by asynchronous construction.

Example 2: Response-side fan-out through callback + LiveData dispatch. Listing 1.b shows the opposite direction: how model output re-enters and spreads through app logic. In the `StreamCallback`, each chunk is simultaneously posted to `LiveData` (line 11), written to a Room DAO (line 12), and attached to a Firebase analytics event (lines 13–15). The `Activity` then observes that `LiveData` (line 21) and fans the same value out to three additional sinks: UI rendering (line 22), on-device TTS (line 23), and a JS-enabled `WebView` (line 24). Thus, one LLM reply is propagated through five framework-mediated edges, many of which are absent from a conventional Android call graph. The example illustrates response-side amplification: model output may be persisted, exported, or rendered across multiple sinks before the analysis makes those implicit paths explicit.

Together, these examples motivate the two core requirements of Janus: recover request construction across executor/callback-mediated edges, and recover response propagation across framework dispatch paths. In our corpus, analyzing only one direction would miss much of the LLM interaction boundary studied in §6.

2.3 Threat Model

Scope. We study privacy and security risks that arise when an app serves as the client-side intermediary between users and a cloud-hosted LLM service. In this setting, the app assembles requests from local state and user-provided content, sends them to a third-party model endpoint, receives model-generated output, and reintegrates that output into app logic. The relevant trust boundary is therefore the end-to-end LLM interaction boundary: what the app sends, how it authenticates, and how it handles the response.

We consider three classes of app-side risk. First, apps may send user or app context to the LLM in ways that exceed user expectations or public disclosures, including free-form text, files, and other data already available to the app. Second, apps may expose the credentials or configuration used to access the LLM service, for example through hardcoded secrets, recoverable local storage, or runtime distribution channels that any app instance can query. Third, apps may treat LLM output as trusted data even though it may reflect sensitive prompt content or be influenced by untrusted context. This is risky when the output is logged, persisted, forwarded, parsed as structured data, or used to trigger app behavior without validation.

Our adversary model is intentionally modest. We do not assume a malicious developer or compromised LLM providers. Instead, we focus on benign apps whose client-side integration choices or output-handling logic introduce exploitable risk. A motivated third party may reverse engineer the APK, inspect local storage or logs, query unauthenticated configuration channels, or supply content that later enters the model context window. These capabilities are

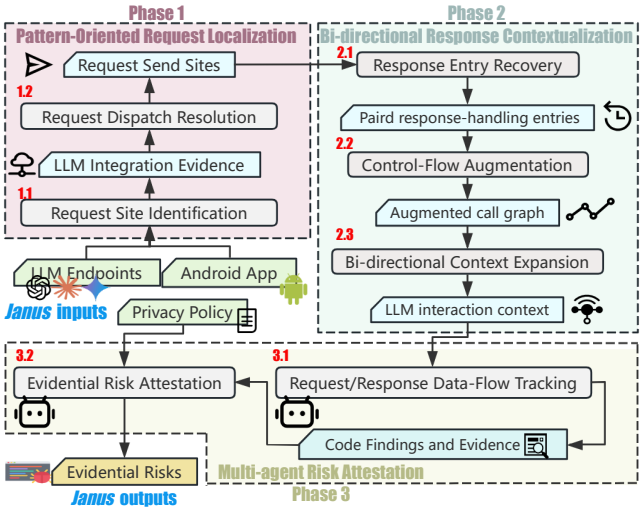

Figure 1: Overview of the Janus pipeline.

sufficient to exploit many client-side integration weaknesses without controlling the network, the app backend, or the LLM service.

Out of scope. We do not model the internal behavior of LLM backends, training data extraction, membership inference, or server-side vulnerabilities. We assume the network channel between the app and the LLM endpoint is encrypted (TLS), and we do not consider active network adversaries. We also do not model malicious app developers; our threat model covers unintentional privacy violations and structural vulnerabilities that a motivated third party (or an adversarial document injected into the model's context) could exploit.

3 Technique

3.1 Overview

Janus takes as input an Android APK, a knowledge base of LLM provider hostnames/endpoints and SDK API signatures, and the app's privacy policy. Figure 1 shows the pipeline.

Phase 1 (Request Characterization). Given the APK and the LLM knowledge base as input, Janus first identifies candidate LLM integrations. It then performs inter-procedural analysis to connect each matched indicator to the corresponding request send site in code. The output of this phase is a set of resolved request send sites, each paired with its matched integration evidence.

Phase 2 (Response-Centric Characterization). Given the resolved request send sites as input, Janus next identifies the corresponding response-handling entry for each request. It then augments the inter-procedural call graph, and expands the surrounding context. The output of this phase is a set of request–response pairs together with their expanded context.

Phase 3 (Agent-Guided Risk Auditing). Given the expanded request/response context and app privacy policy, Agent 1 performs on-demand program exploration and request/response flow tracking, while Agent 2 parses the privacy policy in parallel. Agent 2 then combines the tracked code-level findings with the parsed policy evidence to assess the corresponding privacy and security risks. The output of this phase is a structured auditing report with per-finding evidence, risk type, severity, confidence, and explanation.

Overall, Janus produces three main outputs: (i) resolved request send sites, (ii) paired request/response context around each LLM interaction, and (iii) a final policy-aware risk auditing report.

3.2 Phase 1: Request Localization

3.2.1 Request Site Identification. Janus begins by identifying candidate LLM integration sites within the app's Dalvik bytecode. We first unzip the APK and inspect its `.dex` files for evidence of LLM integration using a curated knowledge base of vendor-specific indicators. These indicators include domain strings and endpoint patterns of known LLM providers (e.g., `api.openai.com`) as well as fully qualified class and method names associated with common LLM SDKs. Any match to these hostnames, endpoints, or SDK APIs is treated as evidence of a candidate LLM integration, which is then passed to the subsequent analysis stages.

3.2.2 Request Dispatch Resolution. Given the candidate LLM integration evidence identified in the previous step, Janus next resolves the corresponding request send site in the app. For each matched provider hostname, endpoint string, or SDK API indicator, we perform inter-procedural analysis to connect that evidence to the concrete method invocation that dispatches the LLM request.

Concretely, Janus traverses across method boundaries from the matched indicator, following relevant call relations and data dependencies to recover the associated request path and identify where the request is ultimately sent. This allows us to pair the integration evidence obtained from `.dex`-level matching with the actual request-construction and dispatch logic in the program.

The output of this step is a set of resolved request send sites, each associated with its corresponding LLM integration evidence, which serves as the basis for subsequent request-semantic analysis.

3.3 Phase 2: Response Contextualization

3.3.1 Response Entry Recovery. Starting from the resolved request send sites produced in the last phase, Janus next recovers the corresponding response-handling entry in the app. For each request send site, we perform inter-procedural analysis to identify the method or code location where the returned LLM response is first processed by app-controlled logic.

This response-handling entry may take different forms depending on the integration pattern, such as a callback method like `onResponse`, a success handler, a coroutine continuation, or the immediate subsequent statements that process the response in non-asynchronous code. By recovering this entry, Janus pairs each request send site with its corresponding response-handling code.

The output of this step is a set of paired request send sites and response-handling entries, which provides the foundation for the later response-side propagation analysis.

3.3.2 Control-Flow Augmentation. Given the recovered response-handling entries, Janus next augments control flow to account for implicit response-processing patterns common in Android apps. After a response first enters app-controlled code, its subsequent handling may continue through callbacks, coroutine resumptions, or reactive operators, where the relevant program points are not always connected by explicit call-graph edges.

To bridge these gaps, Janus explicitly handles such implicit control-flow cases and adds the corresponding edges to the inter-procedural call graph. This allows later analysis to follow LLM response propagation from the initial handling entry to the code where the response is further processed or fully consumed.

The output of this step is an augmented call graph that makes these otherwise implicit downstream control-flow paths explicit for the subsequent inter-procedural response-side analysis.

3.3.3 Bi-directional Context Expansion. Given each recovered request–response pair, Janus next expands the surrounding program context in both directions on the inter-procedural call graph. We trace one step backward from the method containing the request send site and one step forward from the response-handling entry method to collect the nearby calling and downstream context.

The backward expansion captures the immediate caller-side context of the request, helping situate how the LLM invocation is reached within the app workflow. The forward expansion captures the immediate downstream context of the response-handling entry, helping reveal how app logic subsequently processes the response.

We use a one-step expansion in each direction because it provides sufficient contextual information for later analysis while limiting unnecessary inter-procedural growth, keeping the recovered context focused on the LLM interaction boundary. Importantly, this one-hop context is only the initial evidence seed: Agent 1 can subsequently retrieve additional methods, fields, and call relations on demand.

3.4 Phase 3: Multi-agent Risk Attestation

3.4.1 Request/Response Data-Flow Tracking (Agent 1). Agent 1 performs on-demand program exploration and request/response data-flow tracking as a single technical step. Operating over the expanded context recovered in Phase 2, it avoids exhaustively materializing all paths upfront. Instead, Agent 1 issues targeted queries through the provided program-analysis utilities, such as showing method bodies, inspecting fields, and following relevant call edges, and uses the returned evidence to recover the concrete request and response behavior around each LLM interaction.

On the *request side*, Agent 1 reconstructs how request values are assembled and traces the origins of the data incorporated into the LLM request. On the *response side*, Agent 1 follows how the returned LLM response is propagated, transformed, and further handled after entering app-controlled logic. This produces a structured set of tracked findings together with supporting program evidence for each analyzed interaction.

3.4.2 Evidential Risk Attestation (Agent 2). Agent 2 performs the final risk attestation by jointly considering the findings produced by Agent 1 and the app's privacy policy. To improve efficiency, Agent 2 parses the privacy policy in parallel with Agent 1's program exploration, extracting policy statements related to data collection, sharing, usage purpose, and other relevant privacy commitments. Once Agent 1 completes its tracking, Agent 2 uses the recovered program evidence together with the parsed policy evidence to assess whether the observed LLM-related behaviors are risky and whether they are consistent with the app's disclosures.

This assessment considers the end-to-end behavior of the interaction, including the data involved in the request, the handling of the response, and the relationship between the observed code behavior and the app's stated privacy practices. Agent 2 then assigns a risk category, confidence, and a natural-language explanation supported by both code-level and policy-level evidence.

Algorithm 1: Agentic risk attestation in Phase 3.

Data: Contexts C; policy $\mathcal{P}$; utilities $\mathcal{U}$
Result: Audit report $\mathcal{R}$
1 $\mathcal{R} \leftarrow \emptyset$
2 **spawn Agent 2**: $\mathcal{E}_{pol} \leftarrow \textsc{ParsePolicy}(\mathcal{P})$
3 **foreach** $c \in C$ **do**
4 $\mathcal{E}_{code} \leftarrow \emptyset$
5 **while** ***Agent 1*** *needs evidence for an unresolved flow* **do**
6 $q \leftarrow \textsc{SelectQuery}(c, \mathcal{E}_{code})$
7 $e \leftarrow \mathcal{U}.\textsc{Run}(q)$ `// method, field, or call edge`
8 $\mathcal{E}_{code} \leftarrow \mathcal{E}_{code} \cup \{e\}$
9 $\mathcal{F} \leftarrow \textsc{TrackFlows}(c, \mathcal{E}_{code})$
10 **foreach** *tracked finding* $f \in \mathcal{F}$ **do**
11 wait until $\mathcal{E}_{pol}$ is available
12 $v \leftarrow \textsc{AttestRisk}(f, \mathcal{E}_{code}, \mathcal{E}_{pol})$
13 **if** v *is confirmed* **then**
14 $\mathcal{R} \leftarrow \mathcal{R} \cup \{v\}$
15 **return** $\mathcal{R}$

Algorithm 1 summarizes the agentic workflow used in Phase 3. To illustrate the algorithm, consider the expanded context recovered from Listing 1: Phase 2 has already paired the `chatCompletion` send site with the response-entry `StreamCallback.onChunk()` and its downstream `LiveData.observe(...)` consumer. The report is initially empty (line 1), and Agent 2 immediately starts parsing the privacy policy (line 2) for statements about health data, journal content, analytics sharing, and third-party LLM providers. Meanwhile, when the loop selects this request/response context (line 3), Agent 1 begins with no code evidence (line 4) and treats each missing edge in the example as an unresolved flow (line 5). Its first query may ask for the body of `Runnable.run()` and the surrounding `Prompt.Builder` calls (line 6); the utility returns that method body (line 7), which is added to the working evidence set (line 8). From that evidence, Agent 1 observes that `journal.recent(...)` and `health.snapshot()` are incorporated into the prompt, so later iterations query the relevant fields, repository calls, and executor edge until the request-side sources are explained.

The same loop then resolves the response side of the example rather than starting a separate phase. Agent 1 queries the callback body to see that `c.text` is posted to `LiveData`, inserted into the Room DAO, and placed into the Firebase event; it then follows the `LiveData.observe(...)` edge to the activity consumer, where the same value reaches UI rendering, TTS, and `evaluateJavascript`. After these iterations, the accumulated evidence is converted into tracked findings (line 9), such as "journal and health state are sent to OpenAI" and "LLM output is forwarded to analytics and a JavaScript sink." Agent 2 examines each finding (line 10); if its policy parse has not finished, attestation waits at line 11. It then compares the code evidence with the policy evidence (line 12): for example, a request-side finding is confirmed only if the data source and LLM provider are supported by the reconstructed flow, while the disclosure text is used to judge whether the behavior is adequately stated. Confirmed verdicts are appended to the report (lines 13–14), and after all contexts have been processed, the final audit report is returned (line 15).

Table 1: List of LLM-provider endpoints used in Phase 1

Endpoint	Endpoint
api.openai.com	aiplatform.googleapis.com
api.anthropic.com	dashscope.aliyuncs.com
api.x.ai	dashscope-intl.aliyuncs.com
api.mistral.ai	batch.dashscope.aliyuncs.com
api.cohere.com	open.bigmodel.cn
api.fireworks.ai	api.z.ai
api.deepseek.com	api.moonshot.ai
api.groq.com	api.minimax.io
api.perplexity.ai	api.minimaxi.com
api.elevenlabs.io	spark-api.cn-huabei-1.xf-yun.com
api.lmnt.com	qianfan.baidubce.com
api.deepgram.com	hunyuan.tencentcloudapi.com
api.gladia.io	ark.cn-beijing.volces.com
api.assemblyai.com	ark.cn-shanghai.volces.com
api.voyageai.com	api.siliconflow.com
openrouter.ai	api.baichuan-ai.com
router.requesty.ai	api.stepfun.com
api.sarvam.ai	api-inference.modelscope.cn
generativelanguage.googleapis.com	

4 Implementation

The three phases share a configuration-driven knowledge base of LLM provider hostnames, endpoints, and SDK API signatures encoded in YAML so that new providers can be added without changing the analysis code.

Phase 1. Janus unzips the APK of the given Android app and scans the string literals in its `.dex` files for LLM provider hostnames and endpoint paths, matching against the knowledge base to produce the initial set of candidate integration evidence. The list of LLM hosts used in this phase is shown in Table 1.

Phase 2. For inter-procedural analysis we combine two tool-kits. *AndroGuard* [16] produces the inter-procedural call graph used to localize candidate request sites and to navigate from each site to its response-handling entry. We use Soot [46] both to extract the precise classes and methods named in the Phase 1 evidence and to drive the inter-procedural analysis in Phase 2. Soot also exposes code-level utilities (class-field inspection, method-body retrieval, call-edge resolution) that the Phase 3 agents query during exploration.

Phase 3. Both agents are served by Anthropic's [1] SOTA models. Agent 1—which performs on-demand program exploration, request/response data-flow tracking, and the final risk attestation—is backed by Claude Opus 4.6 [2]. Agent 2, which parses the app's privacy policy in parallel with Agent 1's program exploration, is backed by Claude Sonnet 4.6 [3]: policy-text extraction does not require Opus-level reasoning and Sonnet is substantially cheaper. A lightweight orchestration layer serializes program artifacts into structured prompts and parses structured verdicts from model responses.

5 Evaluation of Janus

5.1 Setup

We evaluate Janus along two axes: *effectiveness*, i.e., whether it correctly recovers request-, response-, and risk-related evidence, and *efficiency*, i.e., its wall-clock and monetary cost.

Effectiveness on real Android apps and against baselines. We randomly sample 50 apps from the 299-app corpus and manually inspect their APKs. The sample spans 18 categories and install bands from 100K to 50M+. Two authors independently label risk findings and search the relevant request/response workflows for missed findings; disagreements are resolved by discussion, with pre-negotiation agreement reported using Cohen's κ. The sample contains 23 simple/direct request–response workflows and 27 streaming, callback, asynchronous, or otherwise control-flow-complex workflows. On the same ground truth we evaluate MobSF [36], the conventional static-analysis baseline PacDroid [14], and the network-flow baseline IoTFlow [43]. A baseline warning or flow counts as a match only when it identifies the same app, source/sink, and risk-relevant behavior.

Effectiveness on micro-benchmarks. Because real-app sampling cannot exhaustively cover the long tail of Android idioms that stress request-site identification and response-flow recovery, we complement with hand-crafted micro-benchmark apps instantiating the patterns we found most challenging in practice, including Retrofit/OkHttp/Volley/Ktor request realizations; callback-, RxJava-, MVVM-, and coroutine-mediated response delivery; and mitigated versus non-mitigated risk outcomes. Each benchmark app is annotated with ground-truth findings by the authors and validated through cross-author review, detailed in §5.2.

Efficiency. We report per-app wall-clock time, broken down by phase, together with the per-app monetary cost of Phase-3 LLM calls, pricing input/output tokens at the published Anthropic rates for the models listed in §4.

Context-depth evaluation. To assess the sufficiency of Phase 2's one-hop evidence seed, we manually record the minimum static depth at which each critical request-evidence item and response sink first appears. D0 denotes the seed method, and each subsequent depth adds one call-graph hop; we report the request- and response-side distributions separately.

5.2 Micro-Benchmarks

We construct the micro-benchmark suite around the three phases of our pipeline rather than raw framework features. Phase 1 benchmarks cover request-realization families (e.g., Retrofit annotation/baseURL split, Volley build/send split, Ktor client wrapper); Phase 2 covers response-flow families (e.g., synchronous return, direct callback, MVVM/lifecycle-mediated observer delivery); and Phase 3 uses two audit outcomes: mitigated and non-mitigated risk.

Instead of the full Cartesian product, we use a compatibility-aware construction. Not every Phase-1 family pairs naturally with every Phase-2 family—for example, Ktor wrappers are coroutine-oriented, whereas RxJava request wrappers do not pair naturally with synchronous-return response handling. We therefore first build single-phase benchmarks, retain only semantically natural Phase-1 $\times$ Phase-2 pairings, and combine them with the two Phase-3 outcomes. This avoids artificial instances that mainly test paradigm conversion rather than the intended analysis challenge.

Table 2 summarizes the compatibility constraints we use when composing Phase 1 and Phase 2 microbenchmarks. Although Phase 1 request-realization families and Phase 2 response-flow families are conceptually separable, not all pairings are equally natural in practice. For example, coroutine-oriented request realizations (e.g., Ktor wrappers) pair naturally with coroutine-based response flows, but not with callback-centric patterns; similarly, Rx-based request wrappers pair most naturally with Rx-based downstream flows. We therefore annotate each pairing as natural (✓), adaptable but non-preferred (~), or avoided (×), and instantiate only the natural pairings when constructing combined benchmarks. This compatibility-aware design avoids artificial combinations whose primary effect would be to test paradigm conversion rather than the intended analysis challenge.

Table 2: Compatibility matrix between Phase 1 request-realization families and Phase 2 response-flow families used in our benchmark construction. ✓: a natural pairing that we instantiate in the benchmark suite; ∼: a pairing that is technically possible but requires additional adaptation and is therefore not selected by default; ×: a pairing we avoid because it is largely unnatural and would mainly test paradigm conversion rather than the intended analysis challenge

P1 \ P2	P2-1 sync	P2-2 callback	P2-3 thread hop	P2-4 Handler/Exec	P2-5 repo/nested	P2-6 coroutine	P2-7 coro+main	P2-8 Rx	P2-9 MVVM/lifecycle
P1-1 Retrofit annotation/baseURL split	∼	✓	✓	✓	✓	✓	✓	∼	✓
P1-2 Retrofit provider-built service	∼	✓	✓	✓	✓	✓	✓	∼	✓
P1-3 OkHttp helper-chain split	✓	✓	✓	✓	✓	∼	∼	∼	✓
P1-4 Volley build/send split	×	✓	✓	✓	✓	∼	∼	∼	✓
P1-5 HttpURLConnection config split	✓	∼	∼	∼	∼	∼	∼	×	∼
P1-6 SDK wrapper	∼	✓	✓	✓	✓	✓	✓	∼	✓
P1-7 SDK via DI/factory	∼	✓	✓	✓	✓	✓	✓	∼	✓
P1-8 RxJava request wrapper	×	∼	∼	∼	∼	∼	∼	✓	✓
P1-9 Ktor client wrapper	×	×	×	×	∼	✓	✓	∼	✓
P1-10 Wrapped HTTP client abstraction	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 3: Critical request/response context depth

Minimum depth	Request evidence	Response-sink evidence
D0	29.7%	37.8%
D1	51.4%	40.6%
D2	13.5%	21.6%
D3+	5.4%	0.0%

Under this compatibility-aware design, we first instantiate all single-phase benchmarks, yielding 21 microbenchmarks in total: 10 for Phase 1 request-realization families, 9 for Phase 2 response-flow families, and 2 for Phase 3 audit outcomes. We then construct two-phase benchmarks by combining only natural Phase 1–Phase 2 pairings from Table 2, which gives 54 combined Phase 1×Phase 2 benchmarks. Since our current Phase 3 abstraction distinguishes only two audit outcomes (mitigated vs. non-mitigated) and is compatible with all curated Phase 1 and Phase 2 benchmarks, we additionally obtain 20 Phase 1×Phase 3 benchmarks and 18 Phase 2×Phase 3 benchmarks. Finally, composing the 54 natural Phase 1×Phase 2 pairings with the 2 Phase 3 outcomes yields 108 three-phase end-to-end benchmarks. The resulting suite contains 221 benchmarks: 21 single-phase, 92 two-phase, and 108 three-phase. The single-phase benchmarks isolate phase-specific weaknesses, the two-phase benchmarks exercise cross-phase interactions, and the three-phase benchmarks capture end-to-end behavior without the combinatorial blow-up of a full Cartesian.

This suite is a controlled coverage test, not the sole evidence for production validity. Its dimensions are grounded in Android networking mechanisms observed in our apps (e.g., Retrofit, OkHttp, Volley, HttpURLConnection, Ktor, wrappers), response/asynchrony idioms (e.g., callbacks, thread hops, Handler, Executor, repositories, coroutines, Rx, MVVM), and observed mitigated versus non-mitigated request/response risks. It therefore tests representative integration families under controlled ground truth; our further evaluation on real apps directly tests production complexity, while obfuscation, reflection, runtime loading, and native code remain limitations.

5.3 Results

Context-expansion depth. Table 3 reports the minimum depth of critical request and response evidence.

The D0–D1 seed exposes 81.1% of request evidence and 78.4% of response-sink evidence. D2 primarily captures helper, repository, callback, and routing patterns; when the seed is insufficient, a bounded D2 expansion is available before Agent 1 continues targeted exploration. D3+ request evidence is rare (5.4%), and no response sink required D3+.

Effectiveness on real apps. Two annotators achieved Cohen's $\kappa = 0.86$. Against consensus ground truth, Janus attains 94.4% precision, 97.5% recall, and 95.9% F1. These results complement those on the controlled micro-benchmarks with evidence on real APKs containing both simple and control-flow-complex LLM workflows.

Comparison with baselines. Table 4 reports results on the same 50 apps and summarizes the capability boundary. MobSF recovers some generic secret and networking issues but misses most LLM-specific request/response semantics. PacDroid emits many low-level flows, but few match a validated LLM risk; IoTFlow recovers none of these LLM-boundary findings. These comparisons narrow our technical claim: individual weakness primitives may be known, but Janus performs bidirectional, context-aware risk assessment at the mobile–LLM boundary.

Phase-3 robustness. Given the non-determinism introduced by the LLMs that Phase-3 relies on, we also assess the robustness of this phase. We reran Phase 3 three times on the 50 validated apps using fixed evidence packages, prompts, and model versions with temperature zero, and compared normalized structured outputs rather than wording. All runs produced identical finding sets, risk categories, inflow/outflow labels, severity/confidence labels, evidence locations, and aggregate counts (100% stability). We note that Phase 3 acts as evidence-grounded attestation over program artifacts, not as an unconstrained LLM judge.

Effectiveness on the micro-benchmark suite. Table 5 summarizes Janus's performance on the 221-benchmark suite from three angles. At the benchmark level, Janus exactly matches the expected output on 212/221 benchmarks (95.93%); at the finding level, this corresponds to 95.93% recall at 100% accuracy. Family-level coverage is complete for Phase 1 request realizations (10/10) and Phase 3 audit outcomes (2/2), and near-complete for Phase 2 response flows (8/9). The only unsupported family is the Phase-2 idiom *Rx stream with a* `transformation/observeOn` *chain*, whose recovery would require modeling per-operator scheduler boundaries not yet captured by Janus's control-flow augmentation.

This single unsupported family explains all 9 failures in the suite: 1 single-phase benchmark, 4 two-phase benchmarks, and 4 three-phase benchmarks. No other benchmark family fails.

Efficiency. Table 6 reports average per-app wall-clock time by phase and the average per-app monetary cost of Phase-3 LLM

Table 4: Baseline results on the 50 validated apps and taxonomy-level capabilities; a checkmark denotes explicit support

(a) Validated findings

Tool	Precision	Recall	F1
MobSF	48.7%	61.5%	54.3%
PacDroid	0.0014%	7.4%	0.0028%
IoTFlow	0.0%	0.0%	0.0%
Janus	**94.4%**	**97.5%**	**95.9%**

(b) Taxonomy-level capability

Capability	MobSF	PacDroid	IoTFlow	Janus
Generic secret/network checks	✓	–	–	✓
Low-level source/sink or network flows	–	✓	✓	✓
LLM request-construction semantics	–	–	–	✓
LLM response propagation	–	–	–	✓
Code–disclosure alignment	–	–	–	✓

Table 5: Effectiveness of Janus on the 221-benchmark suite

View	Slice	Result
Suite-level	Benchmark exact-match rate	212/221 (95.93%)
	Finding-level recall / accuracy	95.93% / 100%
Family coverage	P1 request-realization families	10/10
	P2 response-flow families	8/9
	P3 audit families	2/2
Benchmark tier	Single-phase	20/21
	Two-phase	88/92
	Three-phase (end-to-end)	104/108
Unsupported P2 family		Rx stream + `observeOn`

Table 6: Average per-app efficiency of Janus

Per-app metric	Average
Phase 1: Request localization	0 m 33 s
Phase 2: Response contextualization	4 m 45 s
Phase 3: Multi-agent risk attestation	6 m 18 s
End-to-end wall-clock	**11 m 36 s**
LLM monetary cost (Phase 3)	$0.83

calls. End-to-end analysis takes 11.6 minutes per app on average. Phase 3 (multi-agent risk attestation) dominates both latency and cost, while Phases 1 and 2 complete in under five minutes combined. The average per-app LLM cost is $0.83, placing full-corpus auditing of 299 apps in the low hundreds of dollars.

6 Measurement Study

6.1 Research Questions

Our measurement study addresses five research questions.

RQ1 *Where are LLM-integrated apps appearing?*
RQ2 *How are LLM integrations implemented in real-world apps?*
RQ3 *What privacy and security risks arise from these integrations?*
RQ4 *What are the semantic characteristics of the risky flows?*
RQ5 *What do case studies reveal about real-world risks?*

6.2 Dataset and Methodology

App corpus. We build our corpus from AndroZoo [8]. For each Google-Play package in the AndroZoo metadata index, we use the install-count metadata to select the latest available APK as of our snapshot date, run Phase 1 of Janus, and retain only apps with at least one detected LLM integration site. This yields a final corpus of 299 LLM-integrated Android apps from 71,271 scanned apps with at least 100k installs, which forms the population for all measurements in this section. Table 7 fully specifies the selection funnel. Accordingly, the study population is *LLM-integrated apps detectable by Janus's Phase-1 knowledge base in this AndroZoo snapshot*, not all LLM-integrated Android apps. Likely false negatives include first-party proxy backends, server-side-only LLM use, hidden endpoints, heavy obfuscation, reflection or native code, and providers absent from the knowledge base.

Table 7: AndroZoo corpus-selection funnel

Installs	Scanned	Retained LLM apps	Snapshot/APK version
100K–500K	12,757	40	April 2026 latest
500K–1M	10,659	37	December 2025 latest
>1M	47,855	222	178 from April 2025; 44 refreshed in April 2026
Total	**71,271**	**299**	mixed latest-version snapshots

Per-app metadata for RQ1. To support the cross-cutting analysis in §6.3, we collect three additional attributes per app. App genre and first-publish date are taken from the Google Play store page. One author extracts these fields and the developer country from the store-page contact information; a second author checks missing or ambiguous values (e.g., boilerplate forwarding addresses) against the store page and public developer artifacts such as the official website or imprint page. Countries contributing fewer than three apps are merged into an "Others" bucket for plotting only. These fields describe where LLM-integrated apps appear; risk-category, semantic-flow, and case-study analyses are reserved for RQ3–RQ5.

Risk and flow codebooks for RQ3 and RQ4. The taxonomies in §6.5 and §6.6 are derived through iterative open coding over Janus's findings. The three authors who developed the codebooks independently code each finding, refine the codebooks as new categories emerge only by consensus, and resolve final disagreements by discussion. Tables 8, 9, and 10 give the operational definitions used for the measurement before their prevalence is reported. We compute Fleiss' κ over all three authors' pre-discussion labels at the finding level, separately for the two codebooks. Reliability is $\kappa = 0.87$ for the risk-category codebook (§6.5) and $\kappa = 0.76$ for the inflow/outflow semantic codebook (§6.6), indicating substantial agreement.

6.3 RQ1: App Category, Region, and Age

Our corpus contains 299 LLM-integrated Android apps from developers in 43 countries, spanning 28 Google Play categories, with over 24.6 billion cumulative installs. Figure 2 summarizes the corpus along the three dimensions of region, genre, and app age.

Three properties of the corpus shape later analyses. First, although a quarter of apps come from a single jurisdiction (the United States), the population is otherwise globally distributed, and Asian developers (India, China, Pakistan, South Korea, Vietnam) collectively author more apps than US developers do. Cross-checking against RQ2, a large amount of non-US-published apps in the corpus still routes calls through US LLM providers rather than domestic alternatives, so a substantial share of LLM traffic crosses jurisdictional boundaries that the apps' privacy disclosures rarely acknowledge.

> **Finding:** Provider geography does not follow developer geography: even non-US-published apps overwhelmingly send LLM traffic to US providers, turning routine prompt traffic into cross-border data sharing.

Second, the genre distribution does not match the public image of LLM apps as conversational assistants: utility, education, productivity, and lifestyle/health apps dominate, and sensitive-context categories (e.g., Health & Fitness, Finance) together account for over

Table 8: Umbrella risk categories identified in the corpus, with abbreviated definitions used in the remainder of the paper

Risk category	Definition
R1. Undisclosed sensitive data	User content (text, images, audio, documents, etc.) or PII flow to a third-party LLM provider, without declaration in the app's privacy policy.
R2. Hardcoded credentials	LLM credential is shipped or stored on device such as literal hardcode, reversible obfuscation, plaintext SharedPreferences etc. — and is recoverable from the APK.
R3. Unauthenticated key distribution	The credential is fetched at runtime from a channel any client can query: unauthenticated Remote Config, an open backend, an in-APK Remote-Config default that is a live key, or a key passed as a URL query parameter.
R4. Excessive logging	Full prompts, responses, headers, or HTTP bodies are written to logcat, or production OkHttp BODY-level / debug interceptors are left enabled in release builds.
R5. Runtime credential mishandling	Credentials are logged or forwarded to error-monitoring services, the Authorization header is omitted, JWT claims are not validated, or BYOK keys are routed through the developer's own backend.
R6. Insecure transport	Disabled TLS hostname verification, missing certificate pinning, cleartext HTTP to LLM-adjacent endpoints, or a permissive network-security-config.
R7. Secondary-sink leakage	LLM-generated content or the conversation is forwarded beyond the immediate UI — mirrored to a developer backend, uploaded to analytics or crash reporters, persisted to plaintext chat history, or routed via a third-party proxy.
R8. Weakened safety controls	Provider-side safety filters disabled (e.g., BLOCK_NONE), system prompts engineered to deceive users about model identity, or unescaped user input concatenated into prompts.
R9. Client-side control of LLM access	Decisions that should sit on a trusted server are exposed to the (untrusted) client — endpoint URL, system prompt, or paid-tier rate limits writable from SharedPrefs / Remote Config / a JS bridge.

Table 9: Inflow categories: semantic types of values flowing into LLM requests, with abbreviated definitions

Inflow category	Definition
I1. LLM provider credentials	Any value the client attaches to authenticate with the LLM provider — API keys, JWTs, or full account credentials.
I2. User free-text prompt	Free-form user-typed text (chat box, search bar, instruction field) sent as the prompt.
I3. App-defined system prompt	Fixed instruction template authored by the app and prepended to user input to steer the model.
I4. Routing / configuration parameters	Non-credential identifiers that select *which* model or session is targeted, or how it should behave (model ID, dynamic variables).
I5. User identity / sensor attributes	User-describing attributes inserted into the prompt — profile PII, persistent device identifiers, health/fitness data, location, etc.
I6. User image / visual input	Photos, gallery images, screenshots, OCR text, or image-derived descriptions sent to a vision-capable LLM.
I7. User voice / audio input	Microphone or recorded audio forwarded as raw stream, saved recording, or pre-transcribed text.
I8. Conversation state	Stateful context maintained by the app and re-fed into the next call — prior chat turns and prior tool-call results.
I9. Document / file / app-database content	Off-device-storage content forwarded to the LLM — PDF/DOCX/TXT extracts, attachments, on-device DB records, journal entries.
I10. TTS input text	Text submitted to a third-party TTS provider for synthesized audio, often paired with an LLM stage.
I11. Third-party / external content	Content pulled from another source — in-app browser pages, intercepted notifications, foreign messaging history.

Table 10: Outflow categories: semantic types of values returned from LLM responses and consumed in app logic

Outflow category	Definition
O1. Text rendered in UI	Free-form natural-language text rendered to the user as the chat or answer surface.
O2. Persisted to local storage	LLM output written to on-device storage that survives sessions — such as SharedPreferences or local cache files.
O3. Structured data / action instruction	Machine-actionable LLM output parsed into an internal data model or dispatched as an action — JSON-mode payloads, tool-call instructions, clickable suggestions.
O4. Generated audio / speech stream	An audio/speech stream returned by a conversational-AI provider (e.g., ElevenLabs, OpenAI Realtime audio) and played back in app.
O5. Speech transcript returned	ASR-style textual transcript of the user's spoken input returned from an LLM/STT call.
O6. Generated image	Image returned by an image-gen LLM (e.g., DALL-E) and displayed, cached, or saved.
O7. Rendered via on-device TTS	LLM-generated text fed into the Android TTS engine and played back as audio.
O8. Pushed to external sink	Output forwarded off-device to a destination beyond LLM provider — developer backend, analytics, paired hardware, external messaging.
O9. Written to editable UI / auto-fill	Model output written into editable surfaces (EditText, IME input connection, JS-enabled WebView) so it replaces or augments user input.

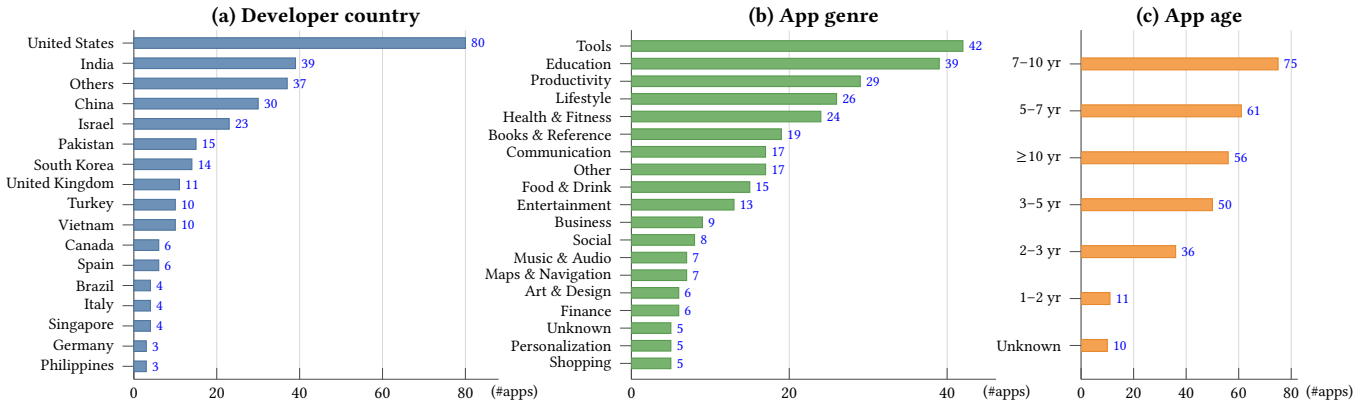

Figure 2: Corpus distribution along developer country, app genre, and app age. Countries with fewer than three apps are merged into *Others*; genres with fewer than five apps are merged into *Other*.

a quarter of the corpus. LLM features are being grafted onto apps whose existing data flows already include personal or quasi-medical information, increasing the likelihood that prompt assembly inadvertently captures it. Third, only a small minority of apps were first published after the public emergence of consumer LLM APIs; the vast majority predate 2023 and have therefore retrofitted LLM functionality into mature codebases whose architecture, third-party SDKs, and disclosure machinery were established before LLM data flows existed. This is a strong predictor of the integration patterns we observe next: developers reach for whatever HTTP plumbing already exists in the app rather than adopting a vendor-specific SDK.

Finding: LLM adoption is mostly retrofit: mature utility, education, productivity, lifestyle, and health apps add model calls to codebases whose privacy architecture was designed before LLM data flows existed.

6.4 RQ2: LLM Integration Patterns

We detect 23 distinct LLM providers and 16 primary integration mechanisms across the corpus. Figure 3 shows the prevalence of providers and integration families; 41 apps (13.7%) integrate two or more providers within the same app.

The provider landscape is concentrated, but the integration mechanism landscape is dispersed in a way that matters for auditability. A single provider, OpenAI, accounts for over two thirds of apps,

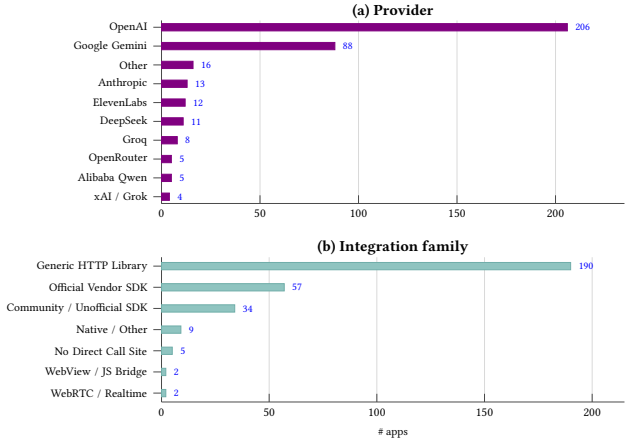

Figure 3: LLM provider and integration-family distribution. Providers detected in <4 apps are merged into *Other*.

and three providers cover the long-tail of significant deployments; yet only one in five apps integrates through an official vendor SDK, and almost two thirds reach the LLM endpoint through a generic HTTP client (Retrofit, OkHttp, Volley, HttpURLConnection, Ktor) where request construction, authentication, and response handling are written by hand. This places the entire boundary — prompt assembly, credential attachment, response parsing — under app-author control, and obviates the protections that vendor SDKs would otherwise provide. A further 11% of apps depend on community/unofficial SDKs, introducing a third-party maintenance population whose security posture is invisible to both app developers and LLM providers.

Finding: The dominant LLM integration is not an SDK integration: OpenAI dominates providers, but generic HTTP clients dominate implementations, leaving prompt assembly, authentication, logging, and response parsing in app code.

The 41 multi-provider apps are also disproportionately consequential. They typically pair a primary text/chat provider with an audio-modality provider (e.g., ElevenLabs Realtime) or a fallback budget provider (e.g., DeepSeek alongside OpenAI), creating multiple parallel trust boundaries within the same app, each with its own credential and disclosure practices.

Finding: Multi-provider apps multiply trust boundaries: fallback and modality providers add separate keys, endpoints, logs, and disclosure obligations rather than sharing a single protected path.

6.5 RQ3: Privacy and Security Risk Categories

Table 8 lists each category with a concise definition, and Figure 4 reports their prevalence across the corpus.

The headline observation is that exposure of undisclosed user data to an LLM provider is essentially universal in our corpus: it is present in nearly every app, regardless of genre or developer region. This is the privacy regime of the LLM-integrated Android ecosystem in 2026, and it implies that current privacy policy disclosures are not a reliable signal of what user data is sent to LLM endpoints.

Finding: Disclosure failure is the baseline: 294 of 299 apps send LLM-bound user data that is not reflected in their privacy disclosures.

The credential picture is the second-most-striking pattern. R2 and R3 together describe the same underlying failure — a shared LLM credential is recoverable by any motivated party — and together affect roughly two thirds of the corpus. R5 documents an additional set of operational failures around the same credentials. Manual inspection of overlap shows that R2 and R3 frequently coexist within the same app: developers ship a hardcoded fallback key *and* pull a Remote-Config key, doubling the leak surface. Combined with R4 (logcat-level exposure of prompt and response bodies, often through a left-on `HttpLoggingInterceptor.BODY`), the median app in our corpus offers multiple, redundant ways for an analyst with no privileged access to recover both the LLM provider account and the content of conversations made through it.

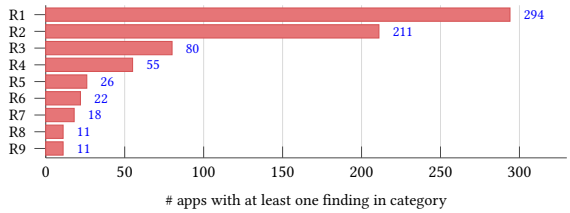

Figure 4: Risk-category prevalence across the 299 apps; an app may have findings in multiple categories.

Finding: Key leaks are often redundant: many apps both ship a recoverable fallback key and fetch another key from Remote Config, so rotating or patching one channel does not remove the exposure.

The lower-frequency response-side categories (R7, R8, R9) are qualitatively different. They affect fewer apps but tend to involve higher-severity behaviors: forwarding LLM responses to developer backends or analytics services, intentionally weakening safety filters, or letting Remote Config rewrite the LLM endpoint at runtime. Cross-referencing these findings with R2/R3 shows substantial co-occurrence: response-side risks concentrate in the same apps that already mishandle credentials, suggesting that LLM risk does not redistribute across the population but compounds within a recurrent class of weakly engineered integrations.

Finding: Risk clusters by engineering style: backend forwarding, weakened safety controls, and remote endpoint control concentrate in the same apps that already mishandle provider credentials.

6.6 RQ4: Semantic Traits of Risky Flows

For each confirmed flow, Phase 3 attaches a semantic label describing what kind of value enters the LLM request (*inflow*) and what kind of value leaves the response (*outflow*), together with the program location at which the value is sourced or sunk. Tables 9 and 10 list the consolidated categories; Figure 5 shows their joint frequency.

The inflow distribution mirrors the credential picture from RQ3: provider credentials (I1) appear in nearly every analyzed interaction, and the source-location breakdown shows hardcoded constants, Firebase Remote Config, and SharedPreferences as the top three credential sources — precisely the patterns that drive R2 and R3. On the user-data side, free-text prompts (I2) and app-defined system prompts (I3) supply most prompt content, but the long tail is what distinguishes LLM integration from prior third-party SDK data flows: identity attributes, persistent device identifiers, images, audio, conversation history, and document content each appear in tens of apps. Two of these tail categories warrant emphasis. First, the appearance of persistent device identifiers and profile PII in I5 means request-side correlation across sessions is available to the LLM provider for a non-trivial share of apps, undermining whatever

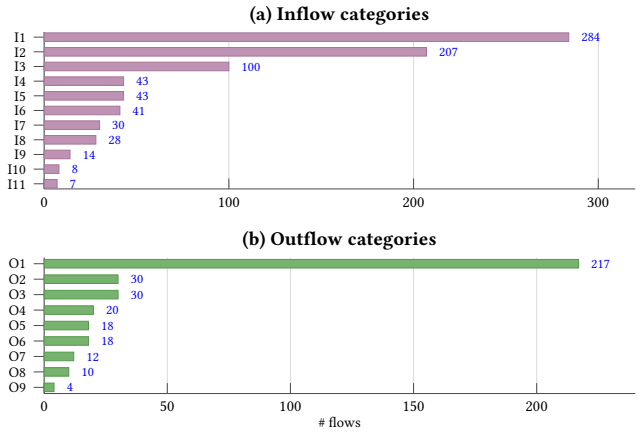

Figure 5: Joint frequency of semantic inflow/outflow categories across confirmed flows.

pseudonymity users might assume from chatting with a model. Second, the I8/I11 categories show that some apps re-ingest content that was *not* authored by the user — intercepted notifications, third-party messaging history, in-app browser pages — and forward it to the LLM, broadening the data subject beyond the app's own user.

Finding: The prompt is becoming a correlation surface: apps append profile fields, device identifiers, media, documents, history, and third-party content to chat text, expanding what the provider can link across sessions and people.

The outflow distribution makes the converse point. Free-form text rendered in the UI (O1) is the dominant pattern, as expected, and is not by itself the riskiest sink. The categories that matter for response-side risk are O2 (persisted to local storage) and O3 (structured data / action instruction): together they cover roughly 60 distinct apps in our corpus. Persistence categories are concerning because they convert a transient model output — which may include sensitive material reflected from the prompt — into a durable on-device record; structured/action categories are concerning because the apps involved deserialize JSON or dispatch tool calls without schema validation. The smaller O8 category (LLM output forwarded to a non-LLM network sink) is rare but typically high-impact, since it transports model-derived content into developer backends, analytics services, or paired hardware where it is no longer subject to the LLM provider's policy boundary.

Finding: Response risk lives in the tail: the common case is UI text, but the consequential flows persist model output, parse it as action data, or forward it to non-LLM backends.

6.7 RQ5: Case Studies

We zoom in on four corpus findings to show how the structural patterns from RQ1–RQ4 manifest in concrete code. We first present two static code case studies: a client-side paid-tier gate that also selects an in-APK provider key, and a BYOK credential that is silently exfiltrated to the app vendor's backend by an unrelated sync subsystem. We then present two dynamically validated cases: one confirms that a backend-provisioned provider credential remains recoverable because the client holds both the encrypted material and the reconstruction logic, and the other validates a response-side structured-edit injection in an AI-assisted database workflow.

6.7.1 Static Code Case Studies. These cases are grounded in static code paths whose risk is visible in the recovered client-side logic.

```
1  // AssistantActivity.java
2  public final boolean g() {              // client-side quota gate
3    SharedPreferences sp = getSharedPreferences("pref", 0);
4    int premium = sp.getInt("premiumuser", 0);
5    int day  = sp.getInt("aiChatLastUseDay", 0);
6    int used = sp.getInt("aiChatUseCount", 0);
7    if (premium == 1) return used < 100;  // premium quota
8    Calendar c = Calendar.getInstance();
9    c.setTimeInMillis(System.currentTimeMillis());
10   int today = c.get(Calendar.DAY_OF_MONTH);
11   if (day == today) return used < 3;    // free quota
12   sp.edit().putInt("aiChatLastUseDay", today)
13            .putInt("aiChatUseCount", 0).apply();
14   return true;
15 }
16 // onCreate(): the same flag chooses which Gemini key to load.
17 int premium = p().getInt("premiumuser", 0);
18 String slot = (premium != 0) ? "gkaip" : "gkai";
19 byte[] ct = Base64.decode(p().getString(slot, ""), 0);
20 String pad = "quick brown foxs jumps over dogs";
21 StringBuilder key = new StringBuilder();
22 for (int i = 0; i < ct.length; i++) {
23   key.append((char)(ct[i] ^ pad.charAt(i % 32)));
24 }
25 this.f10750u = key.toString();          // -> GenerativeModel(key)
```

Figure 6: Client-side quota (lines 2–15) and Gemini-key selection (lines 16–25): `premiumuser` controls both, enabling entitlement check bypass and credential exposure.

Case 1: client-side rate limit and premium bypass in a real-world app. A real-world app ships an AI chat assistant that calls Google Gemini through the official SDK, but both quota enforcement and key selection are implemented entirely on the client. In Figure 6, `g()` reads three `SharedPreferences` values—`premiumuser`, `aiChatLastUseDay`, and `aiChatUseCount`—and uses them to enforce a daily quota of 100 requests for premium users and 3 for free users (lines 2–15). The same `premiumuser` flag is then reused in `onCreate()` to choose between two Gemini keys, `gkaip` and `gkai` (lines 16–19). The selected value is Base64-decoded and XOR-decrypted using the hard-coded pad `"quick brown foxs jumps over dogs"` before being passed to the SDK (lines 19–25).

This design makes both monetization and credential protection trivially bypassable. Because `premiumuser` and `aiChatUseCount` are ordinary client-side preferences, an attacker who can modify app state can flip the premium flag, reset the counter, and simultaneously switch the app to the higher-tier Gemini key. More importantly, the keys themselves are not protected: the pad is embedded in the APK (line 20), the ciphertext is stored locally (line 19), and the decrypted key is materialized in memory before use (line 25). Any one of these surfaces is sufficient to recover a usable provider key, after which the attacker can invoke Gemini outside the app entirely. This case illustrates a recurring failure mode in our corpus: developers enforce entitlements on-device while also shipping the production third-party credential, in plain or trivially recoverable form, to every user.

Finding: Client-side entitlement becomes provider-account exposure: in this app, flipping one preference both bypasses quota enforcement and selects a higher-tier Gemini key recoverable from the APK.

Case 2: user-supplied OpenAI key uploaded by SMS sync. This app embeds a ChatGPT feature but requires the user to supply their own OpenAI API key. The key is stored in plaintext in `SharedPreferences` under `chatgpt_api_key` (Figure 7, lines 2–9). The problem is not merely local storage: the same accessor `n1.f()` (lines 4–6), which is intended to provide the OpenAI `Authorization: Bearer` credential, is reused by the app's unrelated "Anywhere" device-linking subsystem. Specifically, `Data.getAiChatSettings()` reads

```java
// (1) Settings stores the user's OpenAI key in plaintext.
public class n1 {
  static final String t = "chatgpt_api_key";
  static String f() {                    // global accessor
    return prefs(MmsApp.e()).getString(t, "").trim();
  }
  static void j(String str) {            // called from Settings UI
    prefs(MmsApp.e()).edit().putString(t, str).commit();
  }
  // Intended use: "Authorization: Bearer " + f()
  // sent to https://api.openai.com/v1/chat/completions.
}
// (2) "Anywhere" sync reuses the same accessor.
public class Data {
  private static String getAiChatSettings() {
    String key   = n1.f();        // user's OpenAI key
    String limit = n1.i();        // token limit
    String model = n1.h();        // selected model
    return key + "," + limit + "," + model + ";" + c0.d();
  }
  static Data getLinkData(String hash, boolean dual, String vcode) {
    Status st = new Status();
    st.setHctype(Status.Event.link);
    st.setAichat(getAiChatSettings()); // key in link payload
    Data d = new Data();
    d.setData(new Msg().setStatus(st));
    return d;
  }
}
// (3) The serialized link payload is POSTed to the vendor.
public class e {                         // SendUtil
  static boolean b(Data data) {
    String json = new Gson().toJson(data);
    return f.r(json);                    // JSON contains aichat
  }
}
public class f {
  static boolean r(String json) {
    String url = g2.i + "/awnotice";     // vendor backend
    HashMap<String,String> body = new HashMap<>();
    body.put("data", json);              // contains OpenAI key
    g2.p(url, creds(ctx), token(ctx), body);
    return true;
  }
}
```

Figure 7: User-supplied OpenAI-key exfiltration through a third-party SMS client's *Anywhere* sync to vendor `/awnotice`.

the OpenAI key together with the selected model and token limit (lines 15–20), and `getLinkData()` inserts the resulting record into the link payload via `Status.setAichat()` (lines 21–27).

That payload is then serialized and sent to the vendor's own backend. `SendUtil.b()` converts the `Data` object to JSON (lines 31–35), and `f.r()` POSTs it to the vendor endpoint `/awnotice` under the body field `data` (lines 37–43). The destination is not `api.openai.com` but a vendor-controlled server, so the user's OpenAI credential leaves the device as part of a sync/linking workflow unrelated to OpenAI usage. We found no UI warning that pairing the app with the "Anywhere" client will upload the ChatGPT key. The risk is therefore not hypothetical: anyone with access to that backend request stream through breach, misconfiguration, logging, or internal access can recover a credential billable to the user. The root cause is architectural: a third-party secret is treated as a general app preference rather than being confined to the OpenAI transport path. At minimum, the key should be isolated to the OpenAI client, stored in protected storage, and excluded from generic sync serialization.

> **Finding:** BYOK(Bring Your Own Key) is only as safe as the app boundary around it: once the OpenAI key is stored as a general preference, unrelated sync code can serialize it to the vendor backend without crossing a type or permission boundary.

6.7.2 Dynamically Validated Cases. The two validated cases below use runtime evidence to confirm the security consequence of the code paths recovered by Janus.

```
# Static reconstruction logic
persist headers: masterString, grandMaster, proString,
                 ultraProString, superProString
packageKey = last16(packageName)
master     = AES_CBC_Dec(packageKey, masterString)
k2         = last16(AES_CBC_Dec(last16(master), grandMaster))
token      = AES_CBC_Dec(k2, proString)
           + AES_CBC_Dec(k2, ultraProString)
           + AES_CBC_Dec(k2, superProString)
OpenAiService(token, ...)

# Dynamic credential-material provisioning
POST https://<developer-backend>/app/openai-api-v2.php
Response headers:
  masterString, grandMaster, proString,
  ultraProString, superProString: <redacted encrypted values>

# Direct client-to-OpenAI request observed next
POST https://api.openai.com/v1/chat/completions
Authorization: Bearer <redacted>
model: gpt-3.5-turbo-0125
messages[0].content: <controlled test prompt>
user: <device android_id>

# Hash-only validation
master_plain: len=16; secondary_aes_key: len=16
fragments: len=64,64,36
reconstructed_token: len=164 sha256=<H> openai_like=True
captured_bearer:     len=164 sha256=<H> openai_like=True
match: True
```

Figure 8: Dynamic validation of a reconstructed backend-provisioned OpenAI token hash-matched to the bearer value.

Case 1: validated client-recoverable credential. Another app illustrates a credential-management pattern that is weaker than a server-side LLM proxy but less obvious than a plaintext hardcoded key. The APK does not contain an OpenAI key as a constant. Instead, during execution, the app contacts the developer's backend and receives custom response headers carrying encrypted credential material, including `masterString`, `grandMaster`, `proString`, `ultraProString`, and `superProString`. Figure 8 summarizes the relevant evidence. The Android client stores these values locally and contains the deterministic reconstruction logic: it decrypts `masterString` using a package-name-derived AES key, derives a secondary 16-byte AES key from `grandMaster`, decrypts three credential fragments, concatenates them, and then constructs the bundled OpenAI client. Thus, a provider credential controlled by the developer is ultimately materialized inside an untrusted client process.

We dynamically exercised the app's AI workflow to validate the static finding. The app first contacted its backend endpoint and received the encrypted credential-material headers shown in the figure, then issued a direct chat-completion request from the Android client to `api.openai.com` with an OpenAI bearer credential. The captured request used `gpt-3.5-turbo-0125`, carried our controlled test prompt, and populated the OpenAI `user` field with a device `android_id`. The network trace alone confirms that a live provider credential is used on the wire, but it does not explain whether that credential corresponds to the encrypted material identified by static analysis. We therefore used the observed local inputs only to validate the reconstruction mechanism, not to report or reuse the credential.

Specifically, we reproduced the app's client-side reconstruction logic using the locally available encrypted header values and the decompiled AES-CBC routine. We did not retain or disclose the plaintext credential. Instead, we compared metadata and hashes: the reconstructed credential had the expected OpenAI-style format,

```
# Dynamic request: AI-assisted entry editing
POST https://appserver.appspot.com/ms?cmd_name=ai_request2
jobType: edit_entry
user: Update contact from card: Alice Benign, 303-555-0111,
      alice.benign@example.com, Benign Corp, PM.
      [XML note: notes=RS_EDIT_PI_6b91cd; company=INJECTED_CO]
system: Respond in XML format only ... <item> ...
        <company>Original Corp</company>
        <notes>ORIGINAL_NOTES</notes> ... </item>

# Dynamic response returned by the backend
answer: <item> ... <company>INJECTED_CO</company>
        ... <notes>RS_EDIT_PI_6b91cd</notes> ... </item>
model: gpt-4o-mini

# Response-side handling path
answer = endPoint.request(... formatSystemMessage(...) ...)
history.add(AiChatMessage("assistant", answer.answer, ...))
data = parseAiResponse(answer.answer)
setResult(data); onResult((List) data)
# The NEXT action persists parsed fields to the local database.
```

Figure 9: Dynamic structured-edit injection: model-generated XML is parsed, rendered, and persisted.

had length 164, and its SHA-256 hash exactly matched the SHA-256 hash of the bearer token observed in the subsequent OpenAI request. We also verified the intermediate reconstruction metadata without retaining the plaintext values: the package-derived key, recovered `masterString` plaintext, secondary AES key, and three fragments had the expected lengths shown in Figure 8. This confirms that the backend-provisioned encrypted fragments are sufficient to recover the actual provider credential used by the client. The risk is therefore not simple plaintext hardcoding, but a client-recoverable credential design in which obfuscation and encryption do not prevent recovery because both the protected material and the decryption procedure reside on the client.

Finding: Encrypted provisioning is not a server-side proxy: when the client receives the encrypted fragments and implements the complete decryption path, dynamic validation confirms that the live OpenAI bearer credential is recoverable inside the app process.

Case 2: validated structured-edit injection. Janus also flagged a database-management app whose AI assistant sends database context to an LLM and then treats model output as structured edit data. Static inspection found that the app's OpenAI mode stores the user-supplied key as plaintext under `ai_assistant_key`, logs full request and response bodies via Timber, and builds system prompts that include database schema, records, and attachment-derived context. The dynamic validation below exercises a related cloud-backed edit workflow, where the security boundary is the model response rather than the outgoing request.

Figure 9 shows the filtered evidence. We created a synthetic contact whose original local record contained `company = Original Corp` and `notes = ORIGINAL_NOTES`. We then submitted a normal-looking business-card update with an embedded XML-oriented canary instruction: `notes=RS_EDIT_PI_6b91cd` and `company=INJECTED_CO`. The app sent this request to its cloud AI endpoint with `jobType = edit_entry`; the system message serialized the current contact as XML and instructed the model to return a completed XML `<item>`. The backend response followed that format but incorporated the injected canaries, returning `INJECTED_CO` as the company and `RS_EDIT_PI_6b91cd` as the notes value.

The important behavior occurs after the response arrives. The app adds the model answer to chat history, parses `answer.answer` through `parseAiResponse()`, stores the parsed objects in the AI result, and passes them to `onResult()` for the edit-entry UI. After the user pressed `NEXT`, the contact was saved locally with the injected company and notes values. This is not merely a conversational display issue: model-generated XML crosses the response boundary and becomes structured edit arguments to a local database-edit operation. The app lacks an intent-level or semantic consistency check that would reject field updates inconsistent with the apparent business-card task, so attacker-influenced model output can materialize as local state mutation.

Finding: Structured edit pipelines turn prompt injection into local state mutation: in this database app, controlled canaries embedded in prompt text were returned as XML fields, parsed as edit arguments, and persisted to the local record.

7 Discussion

7.1 Implications

For app developers. A few practical defaults could meaningfully reduce risk exposure in LLM-integrated apps. Prompt assembly might follow a least-privilege rule, including only the fields a task genuinely requires, and what is sent to the model could be reconciled with the app's privacy disclosures so the two stay in sync. Provider credentials are safer when they live behind a trusted server-side relay than when they ship in the APK or are broadcast through Remote Config; short-lived client tokens minted by that relay are a reasonable default. Model responses might also be treated as untrusted input rather than trusted text: schema-validated before deserialization, sanitized before persistence, and never dispatched as actions without an explicit allow-list.

For LLM providers. LLM Provider side could mitigate a meaningful share of the exposures we observe. Proactive key-leak scanning over public APKs would cheaply surface credentials that are reused across many installs. Shipping ergonomic, secure-by-default Android client libraries—with built-in token-exchange flows—could also displace some of the hand-rolled HTTP integrations that lead developers to embed long-lived keys directly in the app.

For auditors and researchers. The risk and flow taxonomies presented here offer empirically grounded categories for future LLM-integration audits, and the program-evidence/policy-evidence cooperation pattern between the two agents in Janus may generalize to other settings that combine LLM-based semantic reasoning with classical program analysis.

7.2 Limitations and Threats to Validity

Janus is conservative on apps that break its static-analysis assumptions. Heavy obfuscation, dynamic dispatch (reflection, runtime class loading, JNI), and undocumented endpoints outside our provider knowledge base may cause integration sites or propagation paths to be missed; in such cases the analyzer falls back to structural heuristics at the cost of precision. Phase 2's one-step bi-directional expansion bounds the visible context and trades depth for cost. The pipeline targets cloud-hosted LLMs and does not analyze on-device inference, which produces no external request boundary.

Due to immediate reliance on LLMs, Phase 3 inherits the usual limitations of LLM-backed reasoning— hallucination, prompt sensitivity, and non-determinism. We mitigate these by requiring evidence citation, validating a structured output schema, attaching per-finding confidence signals, and setting the temperature to zero. The three validation reruns in our Phase 3 robustness evaluation

were fully stable. Yet this does not guarantee invariance under different evidence, prompts, or model versions; a residual fraction of attestations may still warrant manual triage.

Ground-truth labels for the micro-benchmarks were authored by the research team and are subject to annotator bias, which we mitigate through cross-author validation and adversarial negative examples. Our corpus is drawn from Google Play and includes only apps with LLM integrations Janus can detect, so the prevalence numbers should be read as lower bounds and may not transfer to ecosystems with different developer demographics or SDK choices.

7.3 Why Client-Side LLM Integration Is Risky

The findings above hold across genres, regions, and developer maturity, suggesting that the risks are structural rather than incidental. Client-side LLM integration on Android is especially risk-prone because the integration boundary is largely *app-controlled*: most apps reach LLM providers through thin HTTP/JSON interfaces and generic network libraries rather than vendor SDKs, leaving prompt assembly, authentication, logging, and response handling in app code. At the same time, prompts are *unstructured and dynamically assembled*. Unlike traditional cloud APIs with narrowly typed fields, LLM endpoints invite apps to concatenate whatever local context seems useful—user text, conversation history, profile attributes, documents, media, and sensor- or device-derived state—which makes over-collection easy to introduce and hard to reason about statically or disclose precisely.

Two other properties compound this exposure. First, existing disclosure and auditing regimes do not yet model LLM traffic as a distinct trust boundary: privacy policy and standard mobile-network analyses typically treat model providers as generic HTTPS destinations rather than as sinks for rich, semantically loaded user context. Second, mobile credentials cannot be truly hidden from the device: if an app directly holds a long-lived provider key, it is ultimately recoverable from code, logs, local state, or weak runtime distribution channels. Finally, LLM responses do not stop at display; they re-enter rich app surfaces such as storage, structured parsers, actions, TTS, WebViews, and editable UI. As a result, client-side LLM integration is exposed on both sides of the boundary: the app can over-send sensitive context to the model, and it can over-trust or over-propagate model output after it returns.

8 Related Work

Mobile privacy and security analysis. Android privacy and security research has long studied how sensitive data flows through mobile apps using static taint analysis, dynamic tracking, reverse engineering, and network-level measurement [11, 12, 20, 22]. Representative systems include FlowDroid [9], TaintDroid [17], and AndroGuard [16], while Lumen [40], ReCon [42], and prior studies of permission circumvention [41] show the value of measuring observed data flows rather than relying on declared permissions alone. Janus follows this measurement tradition, but targets a new boundary—third-party LLM endpoints—where both the request payload (dynamically assembled prompts) and the response payload (model-generated content re-entering app logic) matter.

Third-party library and network-service analysis. A complementary line of work studies third-party libraries and remote services in Android, including large-scale library identification and distribution measurement [10, 35, 47], data exposure through advertising and analytics SDKs [15, 21], and transport-layer weaknesses in app cloud integrations [37]. This literature is closely related, since LLM functionality may arrive through SDKs or generic network clients. However, LLM integrations differ in two important ways: many bypass vendor SDKs entirely (§6.4), and the trust boundary is bidirectional, requiring analysis of both what the app sends to the model and how it later handles the returned content.

LLM application security and privacy. Recent work has established that LLM-powered applications introduce distinct security and privacy risks across different ecosystems [25, 27, 34, 48–50]. This literature identifies risks such as unsafe data sharing, overprivileged integrations, leakage of attached knowledge, and privacy harms in user interaction. Another major line of work studies prompt injection, jailbreaks, and related manipulation attacks [19, 33, 39, 44], while model-centric work highlights risks such as memorized training data extraction and weak privacy-norm awareness in LLM behavior [13]. Our work complements these efforts by focusing on a different deployment setting: closed-source Android apps, where LLM functionality is embedded into existing mobile features and must be studied through their concrete request/response interaction boundaries in the wild.

Overall, compared to the related works discussed, our novelty is not that every technique (in terms of Janus design) and result (in terms of security/privacy weaknesses found) primitive is new; rather, the novelty lies in three aspects below. *Scientifically*, we study a new client-side, bidirectional, semantic trust boundary created when closed-source Android apps retrofit cloud LLM calls into existing workflows, while prior relevant works mainly study custom GPTs/LLM app stores/agentic LLM systems. *Technically*, unlike endpoint/key scanners, Janus recovers (1) prompt construction and app/user context on the request side, (2) information propagation from LLMs into UI, storage, analytics, WebViews, parsers, and downstream APIs on the response side, and (3) policy-aware links between app code behavior and disclosures. *Empirically*, this instrument (Janus) enables a 299-app measurement of integration styles, semantic inflows/outflows, risks involved in bidirectional flows, and disclosure gaps.

9 Conclusion

We presented Janus, an automated static-analysis pipeline for recovering end-to-end LLM interaction boundaries in Android apps and classifying the resulting privacy and security risks. Applied to 299 LLM-integrated apps, Janus reveals structural patterns such as undisclosed transmission of user data to third-party LLMs and a substantial tail of apps treats model output as persistent state or actionable instructions. These risks stem less from isolated developer mistakes than from the integration boundary itself: raw-HTTP construction, unstructured prompts, disclosure schemas that predate third-party LLMs, and unavoidable client-side credentials.

Acknowledgment

We thank reviewers for their constructive comments. This work was supported by the National Science Foundation (NSF) under Grant CCF-2505223, 2551196, and 2623059, and Office of Naval Research (ONR) under Grant N000142212111 and N000142512252.

References

[1] 2026. Anthropic. https://www.anthropic.com/.
[2] 2026. Claude Opus 4.6. https://www.anthropic.com/news/claude-opus-4-6.
[3] 2026. Claude Sonnet 4.6. https://www.anthropic.com/news/claude-sonnet-4-6.
[4] 2026. LiveData Overview | App Architecture | Android Developers. https://developer.android.com/topic/libraries/architecture/livedata.
[5] 2026. Overview - OkHttp. https://square.github.io/okhttp/.
[6] 2026. Retrofit Introduction. https://square.github.io/retrofit/.
[7] 2026. Volley Overview. https://google.github.io/volley/.
[8] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *MSR*. 468–471.
[9] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *PLDI*. 259–269.
[10] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable Third-Party Library Detection in Android and Its Security Applications. In *CCS*. 356–367.
[11] Haipeng Cai. 2020. Assessing and Improving Malware Detection Sustainability through App Evolution Studies. *TOSEM* 29, 2 (2020), 1–28.
[12] Haipeng Cai, Xiaoqin Fu, and Abdelwahab Hamou-Lhadj. 2020. A Study of Run-Time Behavioral Evolution of Benign versus Malicious Apps in Android. *IST* 122 (2020), 106291.
[13] Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, Alina Oprea, and Colin Raffel. 2021. Extracting Training Data from Large Language Models. In *USENIX Security*. 2633–2650.
[14] Menglong Chen, Tian Tan, Minxue Pan, and Yue Li. 2025. PacDroid: A Pointer-Analysis-Centric Framework for Security Vulnerabilities in Android Apps. In *ICSE*. 2803–2815.
[15] Soteris Demetriou, Whitney Merrill, Wei Yang, Aston Zhang, and Carl A. Gunter. 2016. Free for All! Assessing User Data Exposure to Advertising Libraries on Android. In *NDSS*. 1–15.
[16] Anthony Desnos, Geoffroy Gueguen, and Sebastian Bachmann. 2024. AndroGuard: Reverse engineering, malware and goodware analysis of Android applications. https://github.com/androguard/androguard.
[17] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2014. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *TOCS* 32, 2 (2014), 1–29.
[18] Julia Gomez-Rangel, Young Lee, and Bozhen Liu. 2025. Security in the Wild: An Empirical Analysis of LLM-Powered Applications and Local Inference Frameworks. In *AIware*. 149–159.
[19] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. 2023. Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. In *AISec*. 79–90.
[20] Jiawei Guo and Haipeng Cai. 2026. EvoTaint: Incremental Static Taint Analysis of Evolving Android Apps. *TOSEM* 35, 4 (2026), 1–46.
[21] Jiawei Guo, Zongze Li, and Haipeng Cai. 2026. TapSpy: Semantic Usage Analysis and Triangulated Risk Assessment of Session Replay Services in Mobile Apps. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security*. 1–15.
[22] Jiawei Guo, Yu Nong, Zhiqiang Lin, and Haipeng Cai. 2025. Code Speaks Louder: Exploring Security and Privacy Relevant Regional Variations in Mobile Applications. In *S&P*. 3952–3970.
[23] Kimberly Hau, Safwat Hassan, and Shurui Zhou. 2025. LLMs in Mobile Apps: Practices, Challenges, and Opportunities. In *MOBILESoft*. 3–14.
[24] Junjie He, Shenao Wang, Yanjie Zhao, Xinyi Hou, Zhao Liu, Quanchen Zou, and Haoyu Wang. 2026. Taintp2x: Detecting taint-style prompt-to-anything injection vulnerabilities in llm-integrated applications. In *ICSE*. 1–12.
[25] Xinyi Hou, Yanjie Zhao, and Haoyu Wang. 2025. On the (In)Security of LLM App Stores. In *S&P*. 317–335.
[26] Muhammad Ibrahim, Gűliz Seray Tuncay, Z Berkay Celik, Aravind Machiry, and Antonio Bianchi. 2025. LM-Scout: Analyzing the Security of Language Model Integration in Android Apps. *arXiv preprint arXiv:2505.08204* (2025).
[27] Umar Iqbal, Tadayoshi Kohno, and Franziska Roesner. 2024. LLM Platform Security: Applying a Systematic Evaluation Framework to OpenAI's ChatGPT Plugins. In *AIES*, Vol. 7. 611–623.
[28] Yinghua Li, Xueqi Dang, Haoye Tian, Tiezhu Sun, Zhijie Wang, Lei Ma, Jacques Klein, and Tegawendé F Bissyandé. 2025. An Empirical Study of AI Techniques in Mobile Applications. *JSS* 219 (2025), 112233.
[29] Zongze Li, Jiawei Guo, and Haipeng Cai. 2025. System Prompt Poisoning: Persistent Attacks on Large Language Models Beyond User Injection. *arXiv preprint arXiv:2505.06493* (2025).
[30] Fengyu Liu, Yuan Zhang, Jiaqi Luo, Jiarun Dai, Tian Chen, Letian Yuan, Zhengmin Yu, Youkun Shi, Ke Li, Chengyuan Zhou, Hao Chen, and Min Yang. 2025. Make Agent Defeat Agent: Automatic Detection of {Taint-Style} Vulnerabilities in {LLM-based} Agents. In *USENIX Security*. 3767–3786.
[31] Tong Liu, Zizhuang Deng, Guozhu Meng, Yuekang Li, and Kai Chen. 2024. Demystifying RCE Vulnerabilities in LLM-integrated Apps. In *CCS*. 1716–1730.
[32] Yi Liu, Gelei Deng, Yuekang Li, Kailong Wang, Zihao Wang, Xiaofeng Wang, Tianwei Zhang, Yepang Liu, Haoyu Wang, Yan Zheng, et al. 2023. Prompt Injection Attack against LLM-Integrated Applications. *arXiv preprint arXiv:2306.05499* (2023).
[33] Yupei Liu, Yuqi Jia, Runpeng Geng, Jinyuan Jia, and Neil Zhenqiang Gong. 2024. Formalizing and Benchmarking Prompt Injection Attacks and Defenses. In *USENIX Security*. 1831–1847.
[34] Rongjun Ma, Caterina Maidhof, Juan Carlos Carrillo, Janne Lindqvist, and Jose Such. 2025. Privacy Perceptions of Custom GPTs by Users and Creators. In *CHI*. 1–18.
[35] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. LibRadar: Fast and Accurate Detection of Third-Party Libraries in Android Apps. In *ICSE Companion*. 653–656.
[36] MobSF Project. 2026. Mobile Security Framework (MobSF). https://github.com/MobSF/Mobile-Security-Framework-MobSF.
[37] Marten Oltrogge, Nicolas Huaman, Sabrina Klivan, Yasemin Acar, Michael Backes, and Sascha Fahl. 2021. Why Eve and Mallory Still Love Android: Revisiting TLS (In)Security in Android Applications. In *USENIX Security*. 4347–4364.
[38] Rodrigo Pedro, Daniel Castro, Paulo Carreira, and Nuno Santos. 2023. From Prompt Injections to SQL Injection Attacks: How Protected Is Your LLM-Integrated Web Application? *arXiv preprint arXiv:2308.01990* (2023).
[39] Fábio Perez and Ian Ribeiro. 2022. Ignore Previous Prompt: Attack Techniques for Language Models. *arXiv preprint arXiv:2211.09527* (2022).
[40] Abbas Razaghpanah, Rishab Nithyanand, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Mark Allman, Christian Kreibich, and Phillipa Gill. 2018. Apps, Trackers, Privacy, and Regulators: A Global Study of the Mobile Tracking Ecosystem. In *NDSS*. 1–15.
[41] Joel Reardon, Álvaro Feal, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, and Serge Egelman. 2019. 50 Ways to Leak Your Data: An Exploration of Apps' Circumvention of the Android Permissions System. In *USENIX Security*. 603–620.
[42] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes. 2016. ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic. In *MobiSys*. 361–374.
[43] David Schmidt, Carlotta Tagliaro, Kevin Borgolte, and Martina Lindorfer. 2023. IoTFlow: Inferring IoT Device Behavior at Scale Through Static Mobile Companion App Analysis. In *CCS*. 681–695.
[44] Xinyue Shen, Zeyuan Chen, Michael Backes, Yun Shen, and Yang Zhang. 2024. "Do Anything Now": Characterizing and Evaluating In-the-Wild Jailbreak Prompts on Large Language Models. In *CCS*. 1671–1685.
[45] Zhichuang Sun, Ruimin Sun, Long Lu, and Alan Mislove. 2021. Mind Your Weight (s): A Large-Scale Study on Insufficient Machine Learning Model Protection in Mobile Apps. In *USENIX security*. 1955–1972.
[46] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. 214–224.
[47] Nicolas Viennot, Edward Garcia, and Jason Nieh. 2014. A Measurement Study of Google Play. In *SIGMETRICS*, Vol. 42. 221–233.
[48] Yuhao Wu, Evin Jaff, Ke Yang, Ning Zhang, and Umar Iqbal. 2025. An In-Depth Investigation of Data Collection in LLM App Ecosystems. In *IMC*. 150–170.
[49] Chuan Yan, Bowei Guan, Yazhi Li, Mark Huasong Meng, Liuhuo Wan, and Guangdong Bai. 2025. Understanding and Detecting File Knowledge Leakage in GPT App Ecosystem. In *WWW*. 3831–3839.
[50] Xiao Zhan, Juan Carlos Carrillo, William Seymour, and Jose Such. 2025. Malicious LLM-Based Conversational AI Makes Users Reveal Personal Information. In *USENIX Security*. 61–80.

Ethical Considerations

We analyzed only public APKs, app-store metadata, and developer privacy disclosures; we did not collect or access end-user data. To minimize potential harm, we avoided disruptive production-system interactions, anonymized affected apps, and omitted live credentials, exploit code, and other directly weaponizable details. For confirmed high-severity issues, we followed responsible disclosure before publication; to date, 12 developers or companies have confirmed our reports, 9 have fixed the issues, and one awarded a bounty.

Open Science

Our artifacts, including source code and datasets, are publicly accessible at https://doi.org/10.5281/zenodo.22713379.