Mobile OS Memory Management System

Project Report

1. Introduction

1.1 Problem Statement

Android devices run dozens of applications simultaneously, each consuming a share of the limited physical RAM. The stock Android Low Memory Killer (LMK) operates as a hidden kernel-level process that automatically terminates apps when memory pressure is detected. While effective, it provides:

- **No visibility** — users cannot see which apps consume the most memory or why certain apps are killed.
- **No control** — there is no way to selectively choose which apps to keep or kill.
- **No adaptiveness** — LMK uses static OOM-adjustment thresholds baked into the kernel and does not adapt to real-time usage patterns.

1.2 Objective

Design and implement a Real-time Adaptive Memory Management System for Android that:

1. Monitors live memory statistics from a physical Android device via USB.
2. Displays per-process memory consumption with OOM priority classification.
3. Provides intelligent, threshold-aware recommendations for memory optimisation.
4. Allows selective or batch force-stop of low-priority apps.
5. Compares our model's approach against stock Android's LMK for academic presentation.

1.3 Scope

- **Platform:** Android 10–15 (tested on Android 15, Moto G34 5G)
- **Interface:** Web-based dashboard (Streamlit + Plotly)
- **Data Source:** Android Debug Bridge (ADB) over USB
- **Language:** Python 3.14

2. System Architecture

2.1 High-Level Architecture Diagram

```
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
■                       Streamlit Dashboard (app.py)                ■
■  ■■■■■■■■■■■■■■ ■■■■■■■■■■■■■■ ■■■■■■■■■■■ ■■■■■■■■■ ■■■■■■■■■■ ■
■  ■  Memory    ■ ■  Running   ■ ■ History ■ ■  Smart ■ ■Compare ■ ■
■  ■  Overview  ■ ■  Processes ■ ■  Graphs ■ ■  Reco. ■ ■ Table  ■ ■
■  ■■■■■■■■■■■■■■ ■■■■■■■■■■■■■■ ■■■■■■■■■■■ ■■■■■■■■■ ■■■■■■■■■■ ■
■        ■               ■              ■            ■         ■      ■
■  ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ ■
■  ■                    Data Collection Layer                    ■ ■
■  ■          collect_data() — cached with 25s TTL              ■ ■
■  ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ ■
■                  ■                          ■                      ■
■        ■■■■■■■■■■■■■■■■■■■■■■■■       ■■■■■■■■■■■■■■■■■■■■■■■■■■■          ■
■        ■  Live Path     ■        ■  Demo Path          ■          ■
■        ■  (ADB connected)■        ■  (No device)        ■          ■
■        ■■■■■■■■■■■■■■■■■■■■■■■■       ■■■■■■■■■■■■■■■■■■■■■■■■■■■          ■
■                  ■                          ■                      ■
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
                   ■                          ■
    ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■  ■■■■■■■■■■■■■■■■■■■■■■■■■■■
    ■     ADB Layer             ■  ■    Demo Data        ■
    ■  ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■  ■  ■   demo_data.py      ■
    ■  ■   adb_utils.py       ■  ■  ■  (Gaussian noise)  ■
    ■  ■   • run_adb()        ■  ■  ■■■■■■■■■■■■■■■■■■■■■■■■■■■■
    ■  ■   • run_adb_host()   ■  ■
    ■  ■   • force_stop_app() ■  ■
    ■  ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■  ■
    ■  ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■  ■
    ■  ■  memory_reader.py    ■  ■
    ■  ■  • dumpsys meminfo   ■  ■
    ■  ■  • /proc/meminfo     ■  ■
    ■  ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■  ■
    ■  ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■  ■
    ■  ■  process_reader.py   ■  ■
    ■  ■  • dumpsys meminfo   ■  ■
    ■  ■  • dumpsys activity  ■  ■
    ■  ■    processes         ■  ■
    ■  ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■  ■
    ■  ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■  ■
    ■  ■  smart_manager.py    ■  ■
    ■  ■  (Pure logic, no ADB)■  ■
    ■  ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■  ■
    ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
                   ■
                   ▼
    ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
    ■   Android Device (USB)     ■
    ■   Moto G34 5G, Android 15  ■
    ■   USB Debugging Enabled    ■
    ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
```

2.2 Module Summary

| File | Lines | Purpose |

|------|-------|---------|

| `config.py` | ~90 | All tuneable constants: thresholds, OOM priority map, blocklist, demo defaults |

| `modules/adb_utils.py` | ~150 | ADB binary auto-discovery, subprocess wrappers, device detection, force-stop |

| `modules/memory_reader.py` | ~100 | Parse `dumpsys meminfo` and `/proc/meminfo` into `MemoryInfo` dataclass |

| `modules/process_reader.py` | ~140 | Parse per-process PSS and OOM levels into `ProcessInfo` dataclass list |

| `modules/smart_manager.py` | ~130 | Decision engine: usage %, recommendations, kill-candidate selection, comparison |

`modules/demo_data.py`	~95	Gaussian-noise fake data generator for offline demonstrations
`app.py`	~842	Streamlit dashboard: 5 tabs, glassmorphism theme, Plotly charts, kill controls
`requirements.txt`	4	Python package dependencies

3. How It Works — Detailed Walkthrough

3.1 ADB Connection & Device Discovery

When the dashboard starts, it needs to communicate with a physical Android phone. This is done through Android Debug Bridge (ADB), a command-line tool that lets a computer send commands to an Android device over USB.

Step-by-step:

1. Auto-discover ADB binary — The `_find_adb()` function in `adb_utils.py` searches for the `adb.exe` executable:

- First checks if `adb` is already on the system PATH (via `shutil.which`).
- If not found, it probes common Windows installation directories:
- `%LOCALAPPDATA%\Android\platform-tools\`
- `%USERPROFILE%\Android\platform-tools\`
- `%PROGRAMFILES%\Android\platform-tools\`
- Various Android SDK paths
- The resolved path is cached at module import time so it's only computed once.

2. Check device connectivity — `is_device_connected()` runs `adb devices` and parses the output. A device is considered connected only if its line contains `\tdevice` (meaning it's authorised).

3. Fetch device info — `get_device_info()` retrieves the phone model (`ro.product.model`) and Android version (`ro.build.version.release`) via ADB shell property queries.

4. Fallback to demo mode — If no device is detected (ADB not installed, USB not connected, or debugging not authorised), the system seamlessly falls back to `demo_data.py` which generates realistic fake data using Gaussian distributions.

3.2 Memory Data Collection

The system extracts RAM statistics using two complementary ADB commands:

#### Primary Source: `adb shell dumpsys meminfo`

This Android system service produces a comprehensive memory report. The `memory_reader.py` module parses it with regex patterns:

Regex Pattern	Extracts	Example Match

`Total RAM:\s+([\d,]+)\s*K`	Total physical RAM	`Total RAM: 7,643,264K`
`Used RAM:\s+([\d,]+)\s*K`	Currently used RAM	`Used RAM: 5,123,456K`
`Free RAM:\s+([\d,]+)\s*K`	Available RAM	`Free RAM: 2,519,808K`
`Lost RAM:\s+([\d,]+)\s*K`	Unaccounted ("lost") RAM	`Lost RAM: 412,032K`
`status\s+(\w+)`	Kernel memory status	`(status normal)`

All values are stored in a `MemoryInfo` dataclass with fields in kilobytes.

#### Fallback Source: `adb shell cat /proc/meminfo`

A lighter, faster alternative that reads the kernel's memory file directly. Used when `dumpsys meminfo` output cannot be parsed. Provides `MemTotal`, `MemFree`, and `MemAvailable`.

3.3 Process Data Collection

Per-app memory and priority data comes from two separate ADB dumps, merged by `process_reader.py`:

#### Source 1: Per-Process PSS from `dumpsys meminfo`

The module isolates the "Total PSS by process" section (ignoring the "by OOM adjustment" and "by category" sections that would cause false matches) and parses lines like:

```
    310,245K: com.android.chrome (pid 12345 / activities)
    248,671K: com.google.android.gms (pid 1234)
```

Regex: `^\s+([\d,]+)\s*K:\s+(\S+)\s+\(pid\s+(\d+)`

This extracts: PSS (KB), package name, and PID.

> What is PSS? Proportional Set Size — the amount of physical memory a process uses, with shared pages divided proportionally among all processes sharing them. It's the most accurate single metric for "how much RAM does this app use?"

#### Source 2: OOM Priority from `dumpsys activity processes`

This dump reveals each process's OOM adjustment level — the priority category Android assigns for its Low Memory Killer. The regex handles both Android <15 and Android 15 formats:

```
  Android <15:  Proc #42: fore  T/A/FGS  trm: 0 3456:com.whatsapp/u0a123 (service)
  Android 15:   Proc # 0: fg    T/A/TOP  LCMNFUA  t: 0 9993:com.android.settings/1000 (top-activity)
```

Regex: `(?:Proc|PERS)\s+#\s*\d+:\s+(\w+)\s+\S+\s+\S+\s+\S*\s*(?:trm|t):\s+\d+\s+(\d+):(\S+?)(?:/(\S+))?\s+\((.+?)\)`

Each OOM code maps to a human-readable label and a kill priority score (0–5):

OOM Code	Label	Kill Score	Meaning
`fore` / `fg`	Foreground	0	User is actively using this app — never kill
`vis`	Visible	1	App is visible on screen (e.g., widget)
`percep`	Perceptible	2	App is doing something the user can perceive (e.g., playing music)
`svc`	Service	2	Running a background service
`prev`	Previous	3	The app the user most recently left
`svcb`	Service-B	3	Lower-priority background service
`bak`	Background	4	App is in the background, not doing active work
`cch`	Cached	5	Fully cached, safest to kill
`pers` / `psvc`	Persistent	0	System-critical persistent process
`sys`	System	0	Core Android system process
`home`	Home	1	The launcher / home screen

#### Merging: PSS + OOM

The `get_running_processes()` function merges both maps by package name:

1. For each package found in the PSS map, look up its OOM code from the activity dump.
2. If not found in the OOM map, default to `"bak"` (background).
3. Attach the kill score from `config.OOM_PRIORITY`.
4. Sort all processes by PSS descending (biggest memory consumers first).

3.4 Smart Memory Management Engine

`smart_manager.py` is a pure logic module — it never calls ADB. It receives data from the readers and produces decisions.

#### System-Level Health Assessment

```
usage_pct = (used_kb / total_kb) * 100

if usage_pct >= 80%  →  "critical"  (red alert, recommend immediate action)
if usage_pct >= 60%  →  "warning"   (yellow, suggest cleanup)
if usage_pct <  60%  →  "healthy"   (green, no action needed)
```

#### Kill-Candidate Selection

A process is flagged as a kill candidate if both conditions are met:

1. `kill_score >= 3` — only background (4), cached (5), previous (3), or service-B (3) apps
2. `package_name ∉ KILL_BLOCKLIST` — never suggest killing system-critical packages like `com.android.systemui`, `android`, `com.android.phone`, `com.android.settings`, etc.

Candidates are sorted by PSS descending so killing them frees the most RAM first.

#### RAM Freed Estimation

```
estimated_freed_mb = sum(candidate.pss_kb for all candidates) / 1024
```

This gives the user a concrete number for how much memory they'd reclaim.

3.5 Dashboard (app.py)

The Streamlit app provides 5 tabs for different views of the data:

#### Tab 1: Memory Overview

- **4 metric cards:** Total RAM, Used RAM, Free RAM, Usage %
- **Gauge chart:** Plotly indicator with colour-coded ranges (green/yellow/red)
- **Progress bar:** Visual memory fill level
- **System status alert:** Colour-coded recommendation (healthy/warning/critical)
- **Breakdown table:** Used, Free, Lost memory in MB

#### Tab 2: Running Processes

- **Sortable data table:** All detected processes with PSS, Priority, Kill Score, PID
- **Horizontal bar chart:** Top 10 memory consumers with gradient colouring (cyan → purple → pink)
- **Force-stop buttons:** Per-app kill buttons for each killable background process (live mode only)

#### Tab 3: Memory History

- **Used vs Free line chart:** Dual-trace time series (pink = used, green = free)
- **Usage % area chart:** Single-trace percentage over time with fill
- History stores up to 120 data points (~60 minutes at 30-second refresh)

#### Tab 4: Smart Recommendations

- **Candidate metrics:** Number of kill candidates and estimated freeable MB
- **Candidates table:** Package name, PSS, priority, kill score (heat-mapped)
- **Per-app kill buttons:** Individual stop buttons for each candidate
- **Donut pie chart:** Memory distribution among candidates
- **Optimize Now button:** One-click batch kill of all candidates (primary action)

#### Tab 5: Android vs Our Model

- **Comparison table:** Side-by-side feature comparison (5 aspects)
- **Key Innovations list:** 6 claimed innovations for academic presentation

3.6 Caching & Refresh Strategy

- **Cache TTL = 25 seconds** — ADB data is cached to avoid hammering the device with subprocess calls on every Streamlit widget interaction.
- **Auto-refresh = OFF by default** — Users can enable a 30-second auto-refresh toggle in the sidebar.
- **Manual refresh** — "■ Refresh Now" button in sidebar clears cache and reruns.
- This design avoids the sluggishness of aggressive polling while still supporting live monitoring.

3.7 Glassmorphism UI Theme

The dashboard uses a custom CSS injection (~300 lines) for a modern frosted-glass dark theme:

- **Dark gradient background:** `#0a0a1a → #0d1b2a → #1b1040` with radial glow overlays
- **Glass-effect containers:** `backdrop-filter: blur(20px)` with semi-transparent borders and shadows
- **Accent palette:** Cyan (`#00d4ff`), Purple (`#a855f7`), Pink (`#ec4899`), Green (`#10b981`)
- **Gradient title text:** Main heading uses cyan-to-purple gradient fill
- **Themed Plotly charts:** All charts use transparent backgrounds with light-coloured axes and labels
- **Inter font family** imported from Google Fonts
- **Custom scrollbars,** styled alerts, glass tabs, gradient buttons, and hover animations

4. Data Flow Diagram

```
■■■■■■■■■■■■■■■■     USB + ADB     ■■■■■■■■■■■■■■■■■■■■
■  Android     ■ ■■■■■■■■■■■■■■■■■■■■■  Python Process   ■
■  Device      ■  adb shell dumpsys  ■                  ■
■              ■  meminfo / activity  ■  memory_reader   ■
■  Moto G34 5G ■  processes          ■  process_reader  ■
■  Android 15  ■                     ■                  ■
■■■■■■■■■■■■■■■■                      ■■■■■■■■■■■■■■■■■■■■
                                              ■
                                 MemoryInfo + List[ProcessInfo]
                                              ■
                                              ▼
                                 ■■■■■■■■■■■■■■■■■■■■■■■■■
                                 ■    smart_manager.py     ■
                                 ■                        ■
                                 ■   • calculate_usage_%  ■
                                 ■   • get_recommendation ■
                                 ■   • get_kill_candidates■
                                 ■   • estimate_freed_mb  ■
                                 ■■■■■■■■■■■■■■■■■■■■■■■■■
                                              ■
                                 Recommendations + Candidates
                                              ■
                                              ▼
                                 ■■■■■■■■■■■■■■■■■■■■■■■■■
                                 ■     Streamlit (app.py)  ■
                                 ■                        ■
                                 ■   Metrics, Charts,     ■
                                 ■   Tables, Kill Buttons ■
                                 ■   Glassmorphism Theme  ■
                                 ■■■■■■■■■■■■■■■■■■■■■■■■■
                                              ■
                                              ▼
```

```
                                   ■■■■■■■■■■■■■■■■■■■■■■■■■■
                                   ■   User's Web Browser   ■
                                   ■   http://localhost:8501■
                                   ■■■■■■■■■■■■■■■■■■■■■■■■■■
```

5. Key Algorithms

5.1 Kill-Priority Scoring Algorithm

```
Input:  Process P with OOM code from Android
Output: Kill score (0–5)

1. Look up P.oom_code in OOM_PRIORITY dictionary
2. If found → score = OOM_PRIORITY[code]["score"]
3. If not found → score = 3 (default, assume background)

Score Interpretation:
  0 = NEVER kill (foreground, system, persistent)
  1 = Avoid killing (visible, home)
  2 = Kill only under extreme pressure (perceptible, service)
  3 = Safe to kill (previous app, service-B)
  4 = Recommended to kill (background)
  5 = Ideal kill target (cached, not in use)
```

5.2 Adaptive Recommendation Algorithm

```
Input:  usage_pct (float, 0–100)
Output: (severity, title, detail)

if usage_pct >= 80:
    return CRITICAL
    → "Recommend killing low-priority background apps immediately"
elif usage_pct >= 60:
    return WARNING
    → "Consider closing unused background apps"
else:
    return HEALTHY
    → "The device has adequate free memory. No action required."
```

5.3 Kill-Candidate Filtering Algorithm

```
Input:  List of all running processes, MemoryInfo
Output: Sorted list of safe-to-kill processes

candidates = []
for each process P:
    if P.kill_score >= MIN_KILL_SCORE (3)
       AND P.package_name NOT IN KILL_BLOCKLIST:
        candidates.append(P)

Sort candidates by PSS descending (largest memory hog first)
return candidates
```

6. Innovation Comparison: Stock Android LMK vs Our Model

Aspect	Stock Android (LMK)	Our Smart Model
Kill Strategy	Kills lowest OOM-adj process blindly	Ranks by PSS + priority + user-recency
User Awareness	No user notification	Dashboard shows candidates before action
Adaptiveness	Static OOM thresholds	Threshold-aware recommendations (60%/80%)
Visibility	Hidden kernel process	Real-time GUI with history & gauges
Control	Fully automatic, no user choice	User can review & selectively kill

7. Technology Stack

Component	Technology	Version	Purpose
Language	Python	3.14	Core application logic
Dashboard	Streamlit	1.53.1	Web-based interactive UI
Charts	Plotly	6.5.2	Interactive gauges, bars, lines, pies
Data	Pandas	2.3.3	DataFrames for tables
Auto-Refresh	streamlit-autorefresh	1.0.1	Periodic page refresh
Device Bridge	ADB (Android Debug Bridge)	Platform Tools	USB communication with phone
Device	Moto G34 5G	Android 15	Test hardware
OS	Windows	10/11	Development environment

8. Project Structure

```
Os_2/
■
■■■ app.py                  # Main Streamlit dashboard (842 lines)
■■■ config.py               # All configuration constants
■■■ requirements.txt        # Python dependencies
■■■ PROJECT_REPORT.md       # This report
■
■■■ modules/
    ■■■ __init__.py         # Package marker
    ■■■ adb_utils.py        # ADB subprocess wrappers
    ■■■ memory_reader.py    # System memory parser
    ■■■ process_reader.py   # Per-process PSS + OOM parser
    ■■■ smart_manager.py    # Decision engine (pure logic)
    ■■■ demo_data.py        # Fake data generator for demos
```

9. How to Run

Prerequisites

1. Python 3.10+ installed

2. ADB (Android Platform Tools) installed

3. An Android phone with USB Debugging enabled (Settings → Developer Options → USB Debugging)

Steps

```
# 1. Install dependencies
pip install -r requirements.txt

# 2. Connect your Android phone via USB and authorise debugging

# 3. Launch the dashboard
streamlit run app.py

# 4. Open http://localhost:8501 in your browser
```

If no phone is connected, the dashboard automatically runs in demo mode with simulated data.

10. Screenshots Description

The dashboard contains 5 tabs:

1. Memory Overview — Four metric cards showing Total/Used/Free RAM and Usage %, a Plotly gauge, progress bar, and system health status indicator.

2. Running Processes — Scrollable table of all processes with heat-mapped PSS column, horizontal bar chart of top 10 memory consumers, and per-app force-stop buttons.

3. History — Two time-series charts tracking Used vs Free memory and Usage % over time.

4. Smart Recommendations — Candidate table, per-app kill buttons, donut chart of memory distribution, and one-click "Optimize Now" batch kill button.

5. Android vs Our Model — Academic comparison table and list of 6 key innovations.

11. Limitations & Future Work

Current Limitations

- Requires USB connection (no wireless ADB support implemented yet)
- ADB commands add ~2-5 seconds latency per refresh cycle
- Cannot intercept Android's own LMK kills in real-time
- Kill history is lost on page reload (session-state only)

Future Enhancements

- Wireless ADB support (ADB over WiFi)
- Machine learning model to predict memory pressure before it occurs
- Process usage pattern tracking over multiple sessions
- Integration with Android's `ActivityManager` API for finer-grained control

- Export reports as PDF
- Push notifications when memory reaches critical levels

12. Conclusion

This project demonstrates a working adaptive memory management system for Android that improves upon stock Android's LMK by providing:

1. Transparency — Users can see exactly which apps consume how much memory.
2. Control — Users can selectively kill specific apps rather than relying on blind automatic termination.
3. Intelligence — The system uses threshold-aware recommendations and priority-based scoring to suggest optimal cleanup actions.
4. Real-time Monitoring — Live data from an actual Android device is visualised in an interactive dashboard.
5. Academic Value — The comparison framework clearly articulates the innovations over stock Android for course presentation.

The system successfully bridges the gap between Android's low-level memory management and user-facing visibility, creating an educational and practical tool for understanding OS memory management concepts.

Report generated for OS Course Project — Mobile OS Memory Management System