

Performance Analysis and Optimization of Augmented Reality Applications on Mobile Platforms Using Flutter

Harsh Sharma¹, Dr. Akshya Bhardwaj²

M. Tech Scholar, Department of Computer Science And Engineering, University Institute Of Technology, Himachal Pradesh University, Shimla, India.

Harshsharma8049@Gmail.Com¹

Department of Computer Science And Engineering, University Institute Of Technology, Himachal Pradesh University, Shimla, India.²

ABSTRACT – Delivering responsive augmented reality (AR) experiences on mobile devices requires maintaining high rendering performance while operating within the constraints of limited processing power, memory, and battery resources. This study introduces a benchmarking framework developed with Flutter that evaluates the runtime performance of AR applications using Google ARCore on a physical Android device. The proposed framework continuously records important runtime metrics, including frame rate, frame time, CPU utilisation, memory consumption, estimated power usage, and motion-to-photon (MTP) latency during live AR sessions. Six optimisation techniques—distance-based level of detail, visibility control for distant objects, adaptive rendering quality, lazy asset loading, background-isolate data export, and batched platform metric collection—can be enabled independently to examine their individual and combined impact through controlled A/B experiments. Performance evaluation was conducted on a Realme RMX5070 smartphone running Android 16 under identical environmental conditions using a Release build. Experimental results demonstrate that enabling all optimisation strategies increased the average frame rate from 92.94 FPS to 103.27 FPS, representing an improvement of approximately 11.1%. Average memory usage decreased from 286.5 MB to 267.7 MB, while mean UI frame time was reduced by 9.4%. Among the evaluated techniques, adaptive quality adjustment produced the largest individual improvement in rendering performance. CPU utilisation remained consistently close to full capacity throughout all experiments, whereas GPU utilisation metrics were unavailable because of platform restrictions. The proposed benchmarking framework offers a reproducible methodology for analysing Flutter-based mobile AR applications and provides a practical foundation for future optimisation research across different devices and AR platforms.

Key words: *Augmented Reality, Flutter, Google ARCore, Mobile Performance Analysis Benchmarking, Rendering Optimisation, Level of Detail, Adaptive Quality, Mobile Computing*

I. INTRODUCTION

Augmented Reality (AR) has rapidly emerged as an important technology for enhancing user interaction by combining digital information with the physical environment in real time. Applications built on AR are now widely adopted across education, healthcare, retail, industrial maintenance, navigation, and entertainment because they provide immersive and intuitive user experiences. Modern mobile platforms make AR accessible through software development kits such as Google ARCore and Apple ARKit, enabling smartphones to perform motion tracking, environmental understanding, plane detection, and light estimation without specialised hardware.

Despite these advancements, delivering smooth AR experiences on smartphones remains challenging. Mobile

devices have limited computational resources, memory capacity, battery power, and thermal headroom. During an AR session, the camera subsystem, sensor processing, rendering engine, tracking algorithms, and application logic execute simultaneously, placing continuous demand on system resources. As the workload increases, applications may experience lower frame rates, longer rendering times, higher memory consumption, and increased latency, all of which negatively influence user experience.

Performance evaluation is therefore a fundamental step in the development of efficient AR applications. Although various profiling tools are available for game engines such as Unity and Unreal Engine, they are not designed specifically for Flutter applications that integrate native AR functionality through platform channels. In addition, many published studies evaluate optimisation techniques without maintaining consistent experimental conditions,

making it difficult to compare the effectiveness of different approaches or reproduce reported results.

To address these challenges, this research presents a Flutter-based benchmarking framework that enables systematic performance analysis of mobile AR applications. The proposed system executes live ARCore sessions on a physical Android device while continuously recording runtime metrics including frame rate, frame time, CPU utilisation, memory consumption, estimated power usage, and motion-to-photon latency. The framework also supports controlled A/B experimentation by allowing optimisation strategies to be enabled or disabled individually, making it possible to evaluate their independent and combined effects under identical operating conditions.

Unlike conventional benchmarking approaches that focus primarily on rendering performance, the proposed framework integrates runtime monitoring, automated data logging, CSV export, and comparative visualisation within a single application. This integrated design simplifies performance evaluation while improving experimental repeatability and allowing researchers to analyse optimisation techniques using identical workloads across multiple test sessions.

Research Contributions

The primary contributions of this research are summarised as follows:

- Development of a Flutter-based benchmarking application capable of evaluating real-time AR performance using Google ARCore.
- Design of multiple benchmark scenes that represent different rendering workloads for performance evaluation.
- Implementation of an integrated telemetry framework that combines Flutter frame statistics with native CPU, memory, and power information through Android platform channels.
- Development of six independent optimisation techniques that can be evaluated individually or collectively using controlled A/B experiments.
- Experimental validation of the proposed framework on a physical Android device together with a detailed statistical analysis of runtime performance and an objective discussion of the observed limitations.

II. RELATED WORK

Mobile Augmented Reality Frameworks

Recent advances in mobile computing have enabled smartphones to support sophisticated Augmented Reality (AR) applications without requiring specialised hardware. Google's ARCore and Apple's ARKit are currently the two most widely adopted frameworks for mobile AR development. These platforms provide essential capabilities such as environmental understanding, motion tracking, plane detection, light estimation, and image recognition, allowing developers to create interactive AR experiences on consumer devices. Although both frameworks offer similar core functionalities, the overall application performance is highly dependent on device specifications, sensor quality, and environmental conditions rather than the framework alone.

Flutter has become an increasingly popular crossplatform development framework because it allows developers to build Android and iOS applications from a single codebase. However, Flutter-based AR applications typically access native AR functionality through platform plugins, introducing additional communication between the Flutter engine and native platform layers. This integration makes performance analysis more challenging than in applications developed directly using native SDKs or game engines. The benchmarking framework proposed in this work is designed specifically to evaluate this integration while maintaining a reproducible testing environment.

Performance Evaluation In Mobile Ar

Evaluating the quality of an AR application requires more than measuring rendering speed alone. Previous studies commonly assess runtime behaviour using metrics such as frames per second (FPS), frame time, processing latency, CPU utilisation, and memory consumption because these parameters directly influence application responsiveness and user experience. Stable frame pacing is particularly important in mobile AR, where fluctuations in rendering performance may reduce tracking quality and increase perceived latency.

Modern smartphones also operate under strict thermal and power constraints. Continuous camera processing, sensor fusion, and real-time rendering generate sustained

computational load that may trigger thermal throttling during extended AR sessions. Consequently, monitoring CPU activity and memory usage alongside rendering performance provides a more comprehensive understanding of application behaviour than FPS measurements alone. Certain hardware statistics, including GPU utilisation, are unavailable to standard Android applications because of operating system restrictions. Therefore, such metrics should be reported transparently rather than estimated using indirect methods

Optimisation Techniques for Mobile Ar Applications

Researchers have proposed numerous techniques to improve rendering efficiency in graphics-intensive applications. Level of Detail (LOD) methods reduce rendering complexity by displaying simplified versions of 3D models when objects are located farther from the camera. Visibility-based culling techniques avoid processing objects that contribute little to the current scene, thereby reducing unnecessary rendering operations. Likewise, lazy asset loading distributes resource loading over time instead of loading all assets simultaneously, reducing memory spikes and startup delays.

Adaptive rendering strategies dynamically modify visual quality according to current system performance, allowing applications to maintain smoother frame rates during periods of heavy workload. In addition, background execution mechanisms, such as isolates in Flutter, move computationally expensive tasks away from the main UI thread, helping preserve interface responsiveness during data processing and file operations. These optimisation techniques have been widely recommended in mobile application development; however, their effectiveness varies according to scene complexity, device capability, and application architecture.

Research Gap

Although previous studies have investigated various optimisation strategies for mobile AR, comparatively little attention has been given to systematic benchmarking frameworks developed specifically for Flutter-based AR applications. Existing profiling tools are generally integrated with game engines such as Unity or Unreal Engine and provide limited support for applications that communicate with native AR libraries through Flutter platform channels. Moreover, many published evaluations focus on isolated performance metrics without

maintaining identical experimental conditions for direct comparison between baseline and optimised implementations.

The present research addresses this gap by introducing a reproducible benchmarking framework that combines runtime telemetry, configurable optimisation strategies, automated data collection, and controlled A/B experimentation within a single Flutter application. This approach enables consistent evaluation of optimisation techniques while supporting transparent and repeatable performance analysis across different experimental configurations

III. SYATEM DESIGN

The proposed benchmarking framework was developed to provide a structured environment for analysing the runtime behaviour of Flutter-based Augmented Reality applications. Rather than serving solely as an AR demonstration application, the framework integrates scene management, performance monitoring, optimisation control, data logging, and result visualisation within a single system. This architecture enables researchers to evaluate different optimisation strategies under identical experimental conditions while ensuring that performance data are collected consistently throughout each benchmark session

System Architecture

The application adopts a layered software architecture that separates user interaction, application logic, performance monitoring, and native platform communication into independent components. This modular organisation improves maintainability while allowing individual modules to evolve without affecting the overall system.

The presentation layer contains all user-facing interfaces, including the home screen, AR session interface, optimisation configuration screen, and performance visualisation dashboard. These interfaces allow users to launch benchmark scenes, modify optimisation settings, monitor runtime statistics, and compare recorded sessions. Application state is managed using Riverpod, which maintains the current optimisation configuration, stores runtime performance samples, tracks the active Level of

Detail (LOD) state, and preserves historical benchmark sessions for comparative analysis.

The service layer performs the core benchmarking operations. It is responsible for collecting Flutter frame timing information, sampling device performance metrics, recording benchmark sessions, exporting experimental data, and coordinating communication between Flutter and the native Android implementation.

At the platform layer, the application integrates Google ARCore through the **ar_flutter_plugin_plus** package. Native Android code implemented in **MainActivity.kt** retrieves CPU utilisation, memory usage, and estimated power information while also managing CSV file generation through the Android MediaStore framework. The overall relationship between these software components is illustrated in **Figure 1**, which demonstrates how the presentation, application, service, and native platform layers cooperate during benchmark execution.

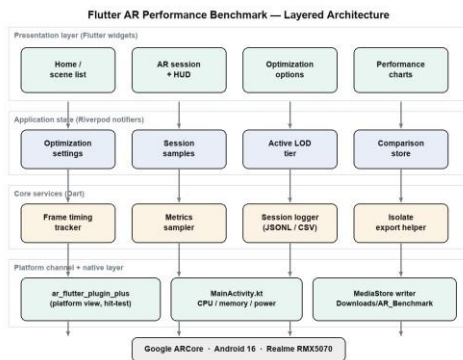


Fig. 1. Layered architecture of the Flutter AR benchmarking application

Performance Monitoring Pipeline

During each benchmarking session, runtime information is continuously collected from multiple sources. Flutter provides frame timing statistics, while native Android services supply CPU utilisation, memory consumption, and power-related measurements. These independent data streams are combined by a central metrics aggregator to produce a unified performance record for the current session.

The aggregated information is simultaneously utilised for three purposes:

- displaying live statistics through the on-screen performance dashboard,
- storing benchmark samples in temporary memory throughout the experiment,
- maintaining historical datasets for subsequent A/B comparison.

Once the benchmark session is completed, the collected samples are serialised into CSV format. Depending on the selected optimisation settings, the export operation may be executed on a background isolate to minimise interference with the user interface. The resulting dataset is then stored within the **Downloads/AR_Benchmark** directory using Android's MediaStore API.

Figure 2 illustrates the complete telemetry workflow from runtime data collection to permanent storage, while also identifying the performance attributes recorded within each exported sample..

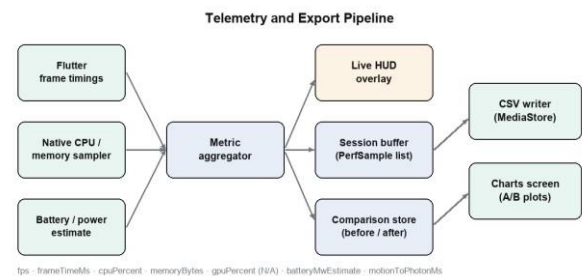


Fig. 2. Telemetry and export pipeline; each CSV row stores nine measured fields.

User Interface Design

The benchmarking application provides a dedicated interface for each stage of the experimental workflow.

The home screen presents the available benchmark scenarios from which users can select the desired workload. After launching an experiment, the AR session interface displays real-time performance statistics through an integrated HeadsUp Display (HUD), allowing users to observe runtime behaviour without interrupting the benchmark.

A separate optimisation configuration screen enables each optimisation technique to be activated or deactivated independently. This functionality supports controlled A/B experimentation by allowing identical benchmark scenes to be executed under different optimisation configurations. The performance comparison screen subsequently visualises the collected benchmark results using statistical summaries and graphical analysis.

Representative screenshots of these interfaces are presented in **Figures 3 and 4**, illustrating both the baseline configuration and the fully optimised configuration employed during the experimental evaluation



Fig. 3. Application UI: (a) home, (b) scene list, (c) all optimisations off (baseline), (d) all optimisations on (optimised).

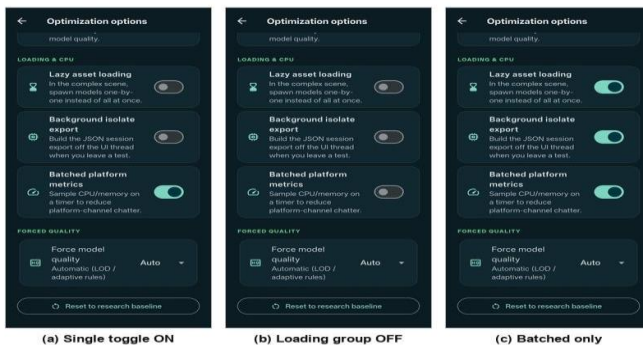


Fig. 4. Optimisation-options detail: (a) a single technique enabled, (b) loading/CPU group disabled, (c) batched-metrics enabled with forced quality control.

Benchmark Scenarios

To evaluate optimisation techniques under different rendering workloads, the framework implements several

benchmark scenarios with varying levels of computational complexity.

The **Single Object Plane** scenario provides a lightweight workload by placing a single virtual object on a detected surface. This configuration serves as a simple baseline for evaluating rendering performance.

The **ChairCompare** scenario, consisting of a life-sized chair positioned beside a small duck model, was selected as the primary experimental workload reported in this study. Its controlled scene composition allows consistent comparison between baseline and optimised executions.

A **Multi-Object Stress** scenario introduces ten simultaneously rendered models arranged in a grid to increase rendering complexity and impose greater demand on system resources. Although implemented within the application, this scenario was not included in the reported quantitative evaluation.

The framework also includes an **Image Tracking** benchmark that anchors virtual content to a printed image marker. While this functionality has been successfully implemented, its performance measurements are outside the scope of the present investigation.

The implemented benchmark scenarios are summarised in **Table 1**, demonstrating how different workloads can be employed to evaluate optimisation techniques under varying rendering conditions.

Table 1. Benchmark scenes implemented in the application.

Scene	Workload description
Single-object plane	One model auto-placed on the floor; light baseline load.
Chair vs duck (chairCompare)	Life-size chair (1:1 m) beside a small duck; used for all reported measurements.
Multi-object stress	Ten models in a grid to stress CPU, GPU, and memory (available in the app; not part of the reported CSV set).
Image tracking	Model anchored to a printed marker (implemented; not claimed in the results).

IV. OPTIMISATION STRATEGIES

The benchmarking framework incorporates six independent optimisation mechanisms that can be enabled or disabled individually. This design supports controlled A/B experimentation by allowing the performance contribution of each technique to be evaluated in isolation as well as in combination. During baseline experiments, all optimisation mechanisms remain disabled, whereas the optimised configuration activates every technique simultaneously. A summary of these techniques is presented in **Table II**.

Table 2. Optimisation techniques exposed for A/B evaluation.

Technique	Intended effect
Distance-based LOD	Swap to a lighter mesh as the camera moves farther away.
Hide far-away objects	Shrink or hide distant nodes to cut rendering work.
Adaptive quality	Force a lower quality tier when FPS drops below ~30.
Lazy asset loading	Spawn models one-by-one in the multi-object scene.
Background isolate export	Serialise the session off the UI thread.
Batched platform metrics	Sample CPU/memory on a timer to cut channel chatter.

Distance-Based LOD and Visibility Control

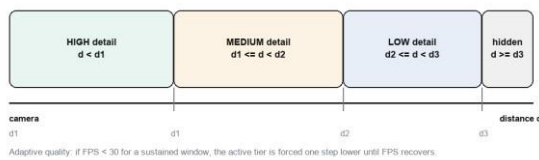


Fig. 5. Distance-based LOD tiers with an adaptive-quality fallback when FPS is low.

V. HARDWARE AND SOFTWARE ENVIRONMENT

Experimental evaluation was carried out on a **Realme RMX5070** smartphone equipped with **8 GB of RAM** and running **Android 16**. The device supports **Google ARCore**, allowing execution of native mobile AR applications without additional hardware. All experiments

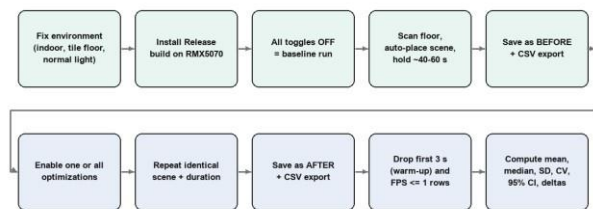
employed the Release version of the application to eliminate debugging overhead and more accurately represent deployment conditions.

Testing was performed indoors under consistent lighting on a tiled floor, providing reliable plane detection throughout the evaluation. The **ChairCompare** benchmark scene, consisting of a life-sized chair positioned alongside a duck model, served as the primary workload for all reported measurements. To remove initial tracking instability, only samples collected after a three-second warm-up period with FPS values greater than one were included in the statistical analysis. The complete experimental configuration is summarised in **Table III**.

Table 3. Device and measurement environment.

Item	Value
Device	Realme RMX5070
Operating system	Android 16
Memory	8 GB RAM
AR runtime	Google ARCore (supported)
Application build	Release
Environment	Indoor, normal lighting, tiled floor
Scene	chairCompare (chair vs duck), automatic placement
Analysis window	Samples after ≥ 3 s warm-up with FPS > 1

A/B Experimental Protocol



Reported metrics: mean/median FPS, frame time, CPU %, memory (MB), MTP estimate, share of samples < 60 FPS.
GPU utilisation counters are not exposed to the unprivileged application and are reported as N/A.

Fig. 6. Controlled A/B protocol; the warm-up transient is discarded before analysis.

Benchmark Dataset

Each optimisation configuration was executed once under identical environmental conditions and exported as an independent CSV dataset. File names were assigned according to the applied optimisation strategy, ensuring an unambiguous correspondence between recorded data and experimental configuration.

During inspection of the exported datasets, two observations were noted. First, the automatically generated "last run" files contained duplicate information corresponding to the immediately preceding benchmark session. Second, the datasets obtained from the backgroundisolate configuration and the batched-platform-metrics configuration produced identical sample sequences and were therefore treated as representing the same experimental observation. The complete mapping between optimisation configurations and exported CSV files is presented in **Table IV**.

Table 4. Mapping of CSV files to optimisation configurations.

Configuration	Source file	Samples
Baseline (all off)	AR_Benchmark_baseline_before_...00-18-42.csv	177
All optimisations on	AR_Benchmark_optimized_after_...00-22-22.csv	198
LOD only	lod_only.csv	133
Hide far-away only	hide_far_away.csv	148
Adaptive quality only	adaptive_qulaity.csv	180
Lazy loading only	lazy_asset.csv	79
Isolate export only	background isolate.csv	114
Batched metrics only	batchplatfor.csv	114

Statistical Analysis

To ensure consistent comparison across all benchmark sessions, measurements recorded during the initial three-second warm-up period were excluded from analysis. Samples with FPS values less than or equal to one, corresponding to the object-placement phase, were also discarded.

The remaining steady-state observations were analysed using descriptive statistical measures, including the

arithmetic mean, median, standard deviation, coefficient of variation, 5th percentile, 95th percentile, and the 95% confidence interval of the mean. Relative improvements produced by each optimisation strategy were calculated with respect to the baseline configuration, allowing quantitative assessment of their individual and combined effects on application performance.

VI. RESULTS

Overall Performance Comparison

The primary objective of the experimental evaluation was to quantify the cumulative effect of enabling all six optimisation strategies within the proposed benchmarking framework. Table V presents the comparative results obtained from the baseline configuration and the fully optimised implementation.

The experimental observations indicate a clear improvement in rendering performance after enabling all optimisation techniques. The average frame rate increased from **92.94 FPS** under the baseline configuration to **103.27 FPS** in the optimised configuration, corresponding to an overall improvement of approximately **11.1%**. A similar trend is reflected in the median frame rate, which increased from **90.59 FPS** to **99.14 FPS**, demonstrating a more stable rendering experience throughout the benchmark session.

Frame rendering efficiency also improved following optimisation. The mean UI frame time decreased from **10.98 ms** to **9.94 ms**, representing a reduction of approximately **1. 9.4%**. Lower frame times indicate that the application required less time to generate each rendered frame, thereby contributing to smoother visual performance.

Memory utilisation showed a measurable reduction during the optimised benchmark. Average resident memory consumption decreased from **286.5 MB** to **267.7 MB**, indicating that the optimisation strategies collectively reduced the application's memory footprint. In contrast, CPU utilisation remained close to full capacity in both experimental configurations, with average utilisation remaining near 100%. This observation suggests that the achieved performance improvements primarily resulted from more efficient rendering and resource management

rather than reduced processor activity. Detailed numerical comparisons are presented in **Table V**, while **Figures 7 and 8** illustrate the corresponding frame-rate behaviour and distribution.

Table 5. Steady-state A/B comparison (chairCompare, Release build, RMX5070).

Metric	Baseline	Optimised	Change
Mean FPS	92.94	103.27	+11.1%
Median FPS	90.59	99.14	—
FPS SD	14.38	19.93	—
FPS 5th pct	74.37	84.84	—
Frame time (ms)	10.98	9.94	-9.4%
Mean CPU (%)	99.9	99.3	≈ flat
Mean memory (MB)	286.5	267.7	-6.6%
Mean MTP est. (ms)	23.05	22.19	—
Samples (n)	177	198	—

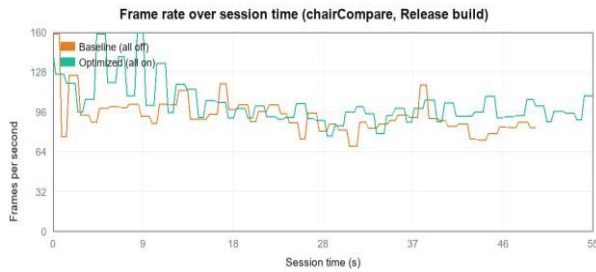


Fig. 7. Per-sample frame rate over the session for the baseline and optimised runs.

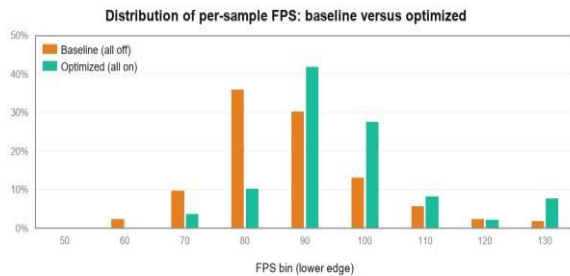


Fig. 8. Distribution of per-sample FPS; the optimised run shifts toward higher bins.

Evaluation of Individual Optimisation Techniques

To better understand the contribution of each optimisation strategy, every technique was evaluated independently

while all remaining optimisations were disabled. The resulting performance measurements are summarised in **Table VI**.

Among the evaluated techniques, **adaptive quality control** produced the largest individual improvement, increasing the average frame rate by approximately **4.6%** relative to the baseline. This behaviour demonstrates the effectiveness of dynamically adjusting rendering quality according to current workload conditions.

Both the **background isolate** and **batched platform metrics** configurations produced small but measurable improvements over the baseline. Although these techniques had only a limited effect on rendering speed, they reduced overhead associated with background processing and communication between Flutter and the native platform.

The **distance-based Level of Detail** and **lazy asset loading** strategies produced performance values comparable to the baseline configuration, indicating that their impact was relatively small under the selected experimental workload. In contrast, **far-object hiding** produced a slight reduction in average frame rate. Because the ChairCompare scene contains only two nearby virtual objects, distant-object culling is rarely activated, meaning that the additional management overhead outweighs any potential rendering savings. Under more complex scenes containing numerous distant objects, the effectiveness of this technique is expected to be considerably greater. The comparative performance of all optimisation strategies is illustrated in **Figure 9**.

Table 6. Mean steady-state metrics by configuration (vs. baseline).

Configuration	Mean FPS	Δ FPS	Mem (MB)	CV %
Baseline (all off)	92.94	—	286.5	15.5
All optimisations on	103.27	+11.1%	267.7	19.3
Adaptive quality	97.25	+4.6%	266.6	18.0
Isolate export	93.19	+0.3%	284.7	23.8
Batched metrics	93.19	+0.3%	284.7	23.8
LOD only	92.41	-0.6%	284.6	13.6
Lazy loading	92.30	-0.7%	283.7	18.5
Hide far-away	89.21	-4.0%	277.4	12.9

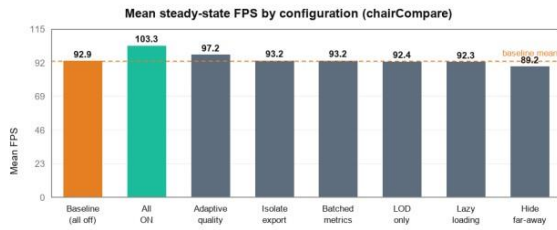


Fig. 9. Mean steady-state FPS by configuration; the dashed line marks the baseline.

Cpu, Memory, And Frame Time Behaviour

Time-series analysis provides additional insight into the runtime behaviour of the application throughout each benchmark session. As illustrated in **Figures 10 and 11**, CPU utilisation rapidly approaches full capacity shortly after the AR session begins and remains close to saturation throughout execution. This behaviour reflects the continuous computational requirements associated with camera processing, motion tracking, rendering, and user-interface updates.

Although processor utilisation remains largely unchanged between configurations, memory consumption exhibits a consistently lower trend after optimisation. The reduced memory footprint is likely attributable to improved resource management, including simplified model representation and more efficient loading strategies.

Because processor utilisation remains nearly constant throughout both experiments, the observed increase in rendering performance cannot be attributed to reduced CPU workload alone. Instead, the results suggest that the optimisation techniques improve the efficiency of rendering operations, enabling more frames to be processed within the available computational budget.

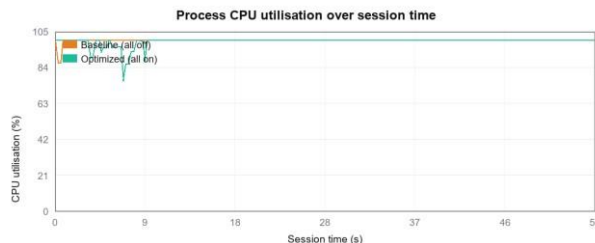


Fig. 10. Process CPU utilisation over the session (both runs saturate near 100%).

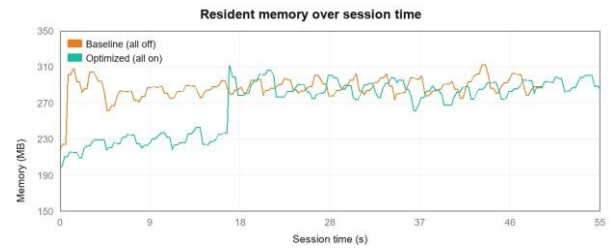


Fig. 11. Resident memory over the session for the baseline and optimised runs.

On-Device Hud Confirmation

Fig. 12 shows the live HUD during chairCompare sessions. The reported FPS (mid-80s to low-100s), memory (~290–310 MB), 100% CPU, N/A GPU, and an MTP estimate near 22–24 ms are consistent with the CSV summaries above.

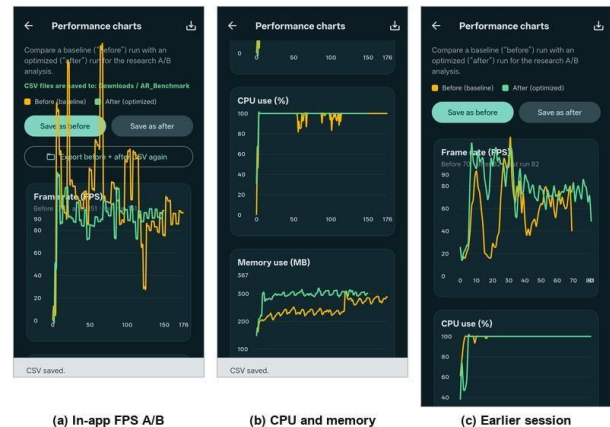
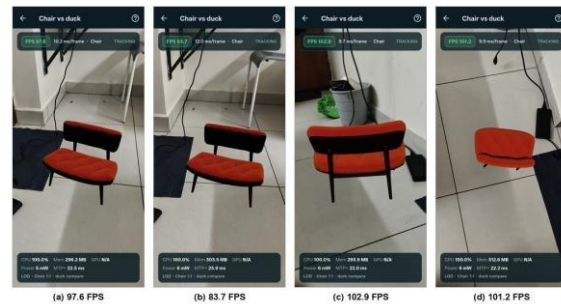


Fig. 12. Live AR HUD during chairCompare at four instants (FPS shown in each panel).

Fig. 13. In-app A/B charts: (a) FPS, (b) CPU and memory, (c) an earlier session.

VII. DISCUSSION

The experimental findings demonstrate that combining multiple optimisation strategies produces greater overall performance improvements than applying individual techniques independently. Rather than relying on a single dominant optimisation, the proposed framework benefits from the cumulative effect of several complementary approaches, each contributing incremental reductions in rendering overhead. This observation highlights the importance of considering optimisation as a coordinated process rather than a collection of isolated techniques.

The effectiveness of an optimisation strategy is also closely related to scene complexity. Techniques such as distance-based Level of Detail and far-object visibility control provide only limited benefits in relatively simple scenes because few objects satisfy the conditions required for these mechanisms to activate. Their impact is expected to become substantially more significant in environments containing larger numbers of virtual objects or greater rendering complexity. Consequently, optimisation strategies should always be evaluated within the context of the intended application workload rather than assumed to perform uniformly across all scenarios.

Another important observation concerns processor utilisation. Although CPU usage remained consistently close to maximum capacity throughout the experiments, rendering performance nevertheless improved after optimisation. This suggests that improvements were achieved primarily through more efficient rendering operations and resource management rather than by reducing overall processor demand. Therefore, metrics such as frame rate, frame time, and memory consumption provide more meaningful indicators of optimisation effectiveness than CPU utilisation alone for the evaluated workload.

Limitations

Several limitations should be considered when interpreting the reported results. First, all quantitative measurements were obtained using the ChairCompare benchmark scene. While this workload provides

consistent experimental conditions, it does not fully represent more complex AR environments containing larger numbers of simultaneously rendered objects.

Second, each optimisation configuration was evaluated using a single benchmark run. Although confidence intervals describe variability within each experiment, repeated trials would provide a more comprehensive assessment of run-to-run variation and the influence of device temperature on sustained performance.

Additionally, some experiments were conducted while the device battery level was relatively low, potentially affecting absolute performance because of powermanagement policies. Finally, the reported motion-to-photon latency represents an estimated value derived from frame timing rather than measurements obtained using specialised optical instrumentation. These considerations should be addressed in future investigations to further improve experimental accuracy.

VIII. CONCLUSION AND FUTURE WORK

This research introduced a Flutter-based benchmarking framework for evaluating the runtime performance of mobile Augmented Reality applications using Google ARCore. The proposed system combines real-time telemetry collection, configurable optimisation techniques, automated data logging, and comparative A/B analysis within a unified experimental platform.

Experimental evaluation performed on a physical Android device demonstrated that enabling all implemented optimisation strategies increased the average rendering performance from **92.94 FPS** to **103.27 FPS**, corresponding to an improvement of approximately **11.1%**. The optimised configuration also reduced average frame time by **9.4%** and lowered memory consumption by **6.6%**, while processor utilisation remained consistently close to maximum capacity. Among the evaluated techniques, adaptive quality control produced the largest individual improvement in rendering performance.

Future work will extend the proposed benchmarking framework by evaluating more computationally demanding benchmark scenes, performing multiple

experimental repetitions under controlled thermal conditions, incorporating detailed energy-consumption measurements, and expanding support to Apple's ARKit platform for crossplatform performance analysis. These enhancements will improve the generalisability of the benchmarking methodology and provide broader insight into the optimisation of Flutter-based mobile AR applications.

Acknowledgment

The author thanks Dr. Akshya Bhardwaj for his guidance during this M. Tech research work at the University Institute of Technology, Himachal Pradesh University, Shimla.

REFERENCES

1. L. Carius, C. Eichhorn, L. R. Rudolph, D. A. Plecher, and G. Klinker, "Cloud-based cross-platform collaborative augmented reality in Flutter," *Frontiers in Virtual Reality*, vol. 3, Art. no. 1021932, 2022. doi: 10.3389/frvir.2022.1021932.
2. M. A. Maneli and O. E. Isafiade, "A Multifactor Comparative Assessment of Augmented Reality Frameworks in Diverse Computing Settings," *IEEE Access*, vol. 11, pp. 12474–12486, 2023. doi: 10.1109/ACCESS.2023.3242238.
3. J. Huai, Y. Zhang, and A. Yilmaz, "The Mobile AR Sensor Logger for Android and iOS Devices," in *Proc. IEEE Sensors*, 2019. doi: 10.1109/SENSORS43011.2019.8956816.
4. P. Nowacki and M. Woda, "Capabilities of ARCore and ARKit Platforms for AR/VR Applications," in *Advances in Intelligent Systems and Computing*, Springer, 2020. doi: 10.1007/978-3-030-19501-4_36.
5. P.-E. Sarlin, M. Dusmanu, J. L. Schönberger, P. Speciale, L. Gruber, V. Larsson, O. Miksik, and M. Pollefeys, "LaMAR: Benchmarking Localization and Mapping for Augmented Reality," in *ECCV*, 2022. doi: 10.48550/arXiv.2210.10770.
6. E. Marino, F. Bruno, L. Barbieri, A. Lagudi, R. Tuma, J. Slater, and F. H. Jhanjhi, "Benchmarking Built-In Tracking Systems for Indoor AR Applications on Popular Mobile Devices," *Sensors*, vol. 22, no. 14, Art. no. 5382, 2022. doi: 10.3390/s22145382.
7. Google, "ARCore Performance Considerations," Google Developers. Available: Google ARCore Developer Documentation.
8. V. Popescu, G. Kesidis, G. de Veciana, and S. Fahmy, "Benchmarking Extended Reality Systems," *IEEE Internet Computing*, vol. 29, no. 6, pp. 7–14, 2025. doi: 10.1109/MIC.2025.3626597.
9. S. A. Syed et al., "In-depth Review of Augmented Reality: Tracking Technologies, Development Tools, AR Displays, Collaborative AR, and Security Concerns," *Sensors*, vol. 23, no. 1, Art. no. 146, 2023. doi: 10.3390/s23010146.
10. K. Apicharttrisor, X. Ran, J. Chen, S. V. Krishnamurthy, and A. K. Roy-Chowdhury, "Frugal Following: Power Thrifty Object Detection and Tracking for Mobile Augmented Reality," in *Proc. ACM SenSys*, 2019.
11. L. Merino, M. Schwarzl, M. Kraus, M. Sedlmair, D. Schmalstieg, and D. Weiskopf, "Evaluating Mixed and Augmented Reality: A Systematic Literature Review (2009–2019)," 2020.
12. J. Lacoche, A. Foulonneau, and S. L. D. Picard, "Comparative Analysis of AR, VR, and Desktop Tools for Prototyping Augmented Reality Services," *IEEE Transactions on Visualization and Computer Graphics*, vol. 31, no. 5, pp. 2579–2588, 2025. doi: 10.1109/TVCG.2025.3549885.
13. V. Popescu, G. Kesidis, G. de Veciana, and S. Fahmy, "Benchmarking Extended Reality Systems," *IEEE Internet Computing*, 2025. doi: 10.1109/MIC.2025.3626597.
14. "Cross-Platform Augmented Reality Mobile Application for Visualization of Genetic Facial Traits: System Design and Use Case," *Procedia Computer Science*, vol. 265, pp. 596–601, 2025. doi: 10.1016/j.procs.2025.07.225.
15. "An Adaptation of Fitts' Law for Performance Evaluation and Optimization of Augmented Reality Interfaces," *IEEE Access*, 2025.