iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** V **Month of publication:** May 2026

DOI: https://doi.org/10.22214/ijraset.2026.82335

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue V May 2026- Available at www.ijraset.com

Efficient Image Loading Application Using Jetpack Compose and Coil

Sathyanarayanachari D[1], Dr. Kavitha A. S[2], Pradeep A[3], Rajath Ganesh B V[4], Srinivas N D[5]
[1,3,4,5]*Students, Dept. of Artificial Intelligence and Machine Learning, East West Institute of Technology, Bangalore-560091, Karnataka, India*
[2]*Professor, Dept. of Artificial Intelligence and Machine Learning, East West Institute of Technology, Bangalore-560091, Karnataka, India*

Abstract: ***Modern mobile applications rely heavily on visual content to improve user interaction, enhance user experience, and create attractive interfaces. Applications such as social media platforms, e-commerce systems, online galleries, educational apps, and news applications continuously display large numbers of images from both local storage and online sources. However, inefficient image loading techniques can lead to increased memory usage, slow rendering, application lag, excessive network consumption, and poor scrolling performance. These issues negatively affect the responsiveness and usability of mobile applications, especially when handling high-resolution images or large image collections.***
This project presents an Image Loading App developed using Kotlin, Jetpack Compose, and the Coil image loading library for Android applications. The application demonstrates efficient asynchronous image loading, image caching, placeholder management, and error handling while maintaining smooth UI rendering and optimized performance. The system supports loading images from internet URLs as well as local application resources and displays them using modern Compose UI components such as lists and grid layouts. Lazy loading mechanisms and Coil's coroutine-based architecture help reduce memory consumption and improve rendering efficiency.
The application also includes features such as retry functionality, loading indicators, responsive screen adjustment, and smooth scrolling behavior for large image datasets. Extensive testing was performed to evaluate image rendering speed, application responsiveness, and performance optimization under different loading conditions. The implementation helped in gaining practical knowledge of modern Android UI development, asynchronous programming, image optimization strategies, and responsive mobile application design. The project successfully demonstrates how efficient image loading techniques can significantly improve application performance and user experience in Android-based systems.
Keywords: Jetpack Compose, Coil Library, Android Development, Image Loading, Kotlin, UI Optimization

I. INTRODUCTION

Images play a significant role in modern mobile applications because they improve user engagement and provide a visually appealing interface. Applications such as social media platforms, e-commerce apps, gallery systems, and news applications depend on efficient image loading mechanisms to maintain smooth performance and fast rendering.
Traditional image loading methods often consume excessive memory and may cause lag, slow scrolling, or application crashes when handling large image collections. To overcome these limitations, modern Android development frameworks provide optimized libraries for asynchronous image loading and caching. This project focuses on developing an Image Loading App using Jetpack Compose and the Coil library. The application demonstrates efficient techniques for displaying images from internet URLs and application resources while ensuring smooth UI performance. Features such as placeholders, retry mechanisms, grid layouts, and error handling improve the overall usability and responsiveness of the application. The project also provides practical exposure to Compose UI components, image rendering optimization, and state management in Android applications.

II. LITERATURE SURVEY

Before starting the actual development of the Image Loading App, we spent time exploring how other developers and researchers approached image rendering and optimization in Android applications. Understanding existing solutions helped us identify practical techniques, common limitations, and areas where modern tools like Jetpack Compose and Coil could improve application performance and user experience.

Early Android applications mostly depended on manual image loading techniques, where developers directly handled bitmap decoding and rendering inside activities or fragments. While this approach worked for smaller applications, it often caused memory leaks, application lag, and crashes when large images or multiple image requests were involved. These issues became more noticeable in applications containing image feeds, galleries, or continuously scrolling content.

The Glide library became one of the most widely used solutions for image loading in Android systems [1]. Glide introduced efficient caching, bitmap pooling, and automatic memory management, which significantly improved rendering speed and reduced memory usage. However, most Glide implementations were designed around XML-based UI systems. Integrating Glide with modern Compose-based interfaces required additional setup and compatibility handling, which slightly reduced development simplicity.

Picasso provided another lightweight approach for image loading and URL-based rendering [2]. It simplified downloading and displaying images from internet sources with minimal code. Although Picasso improved developer productivity, it lacked some advanced optimization and lifecycle-aware features required for modern Android applications handling large-scale image datasets and dynamic UI rendering.

Recent Android development trends shifted toward Jetpack Compose for declarative UI development [3]. Compose simplified UI creation and improved responsiveness compared to traditional XML layouts. However, efficient image handling remained important because Compose applications could still suffer from excessive recompositions, delayed rendering, and poor scrolling performance if image loading was not properly optimized.

Coil (Coroutine Image Loader) emerged as a modern image loading library specifically designed for Kotlin and Jetpack Compose applications [4]. Unlike older libraries, Coil uses Kotlin Coroutines for asynchronous processing, making image loading smoother and more memory efficient. Its lightweight architecture, caching system, placeholder support, and direct Compose integration made it particularly suitable for modern Android development.

Research and developer discussions also highlighted the importance of lazy loading and optimized scrolling mechanisms for applications displaying multiple images simultaneously [5]. Applications using grid layouts or long scrolling image lists often experienced frame drops and UI lag when images were loaded inefficiently. Techniques such as asynchronous loading, image caching, and lazy rendering were found to significantly improve performance and responsiveness.

From this survey, a clear pattern became visible. Many older approaches focused mainly on basic image display without giving enough attention to performance optimization, responsive rendering, or Compose integration. Modern solutions addressed these problems more effectively but often required proper implementation to fully utilize their advantages. Based on these observations, our project focused on combining Jetpack Compose and Coil to create an efficient, responsive, and optimized image loading application capable of handling online and local images smoothly while maintaining a better user experience.

III. PROBLEM STATEMENT

Loading high-quality images in Android applications can lead to increased memory consumption, slow rendering, application lag, and poor user experience if not managed properly. Applications that display multiple images from online sources often face issues such as delayed loading, broken images, excessive network usage, and inefficient scrolling performance. These problems become more noticeable in modern applications such as social media platforms, e-commerce applications, image galleries, and news applications where large numbers of images are loaded continuously. Poor image optimization can also cause frequent application crashes, high battery consumption, and increased device resource usage, negatively affecting the overall responsiveness of the system.

The objective of this project is to design and develop an Android application capable of efficiently loading and displaying images while maintaining smooth UI interaction and optimized performance. The system should support asynchronous image loading, image caching, placeholders, and proper error handling to improve reliability and responsiveness. The application should also ensure smooth scrolling behavior while displaying multiple high-resolution images in list and grid formats. In addition, the project aims to provide a better understanding of modern Android development techniques using Jetpack Compose and Coil for creating responsive and performance-optimized mobile applications.

IV. PROPOSED SYSTEM

The proposed Image Loading App uses Jetpack Compose for UI development and Coil for image loading and caching operations. The application retrieves images from online URLs and local resources and displays them using Compose UI components. Coil handles asynchronous loading in the background, reducing UI thread workload and improving application responsiveness.

1) Image Loading Module: This module loads images from URLs and application resources using Coil APIs. It ensures asynchronous loading and efficient memory usage.
2) UI Rendering Module: Jetpack Compose components are used to render images in lists and grid layouts with smooth scrolling support.
3) Placeholder and Error Handling Module: Placeholder images are displayed while loading is in progress. Error images and retry buttons are shown if image loading fails.
4) Performance Optimization Module: Caching and lazy loading techniques are used to improve performance and reduce repeated network requests.

V. METHODOLOGY

The development of the Image Loading App was carried out through a structured and systematic process to ensure efficient implementation, better performance, and smooth user experience.

The methodology focused on analyzing requirements, designing the user interface, integrating image loading functionality, and evaluating application performance under different conditions. Each phase contributed to improving the responsiveness and reliability of the application.

A. Requirement Analysis

The first stage involved analyzing the requirements necessary for developing an efficient image loading application. The project required support for loading images from both online URLs and local application resources while maintaining smooth rendering performance.

Important features identified during this phase included asynchronous image loading, caching mechanisms, placeholders during loading, retry options for failed requests, and responsive grid layouts for image display.

The analysis also focused on identifying common performance issues such as slow rendering, high memory consumption, delayed scrolling response, and image flickering. Understanding these challenges helped in selecting suitable technologies and optimization techniques for the project implementation.

B. Image Loading Integration

The Coil image loading library was integrated into the application to support efficient asynchronous image retrieval and caching. Coil was selected because of its lightweight architecture, Kotlin coroutine support, and direct compatibility with Jetpack Compose.

The integration process included configuring image requests, placeholders, error images, and memory caching. Asynchronous image loading ensured that network operations were executed in the background without blocking the main UI thread. Caching mechanisms reduced repeated network requests and improved loading speed for previously accessed images.

The application was also configured to handle large image files efficiently while minimizing memory usage and maintaining smooth rendering performance.

C. Performance Testing

After implementation, the application was tested using multiple image URLs, high-resolution images, and different grid layouts to evaluate its overall performance. Testing focused on image loading speed, memory utilization, responsiveness, scrolling smoothness, and error handling behavior.

Different network conditions were simulated to observe the stability of asynchronous loading and retry mechanisms. The performance testing phase confirmed that the application maintained stable rendering and smooth UI interaction even while displaying multiple large images simultaneously. The use of caching and lazy loading significantly improved application efficiency and reduced unnecessary resource consumption.

VI. SYSTEM IMPLEMENTATION

The Image Loading App was implemented using Android Studio with Kotlin programming language and Jetpack Compose for modern UI development. The system architecture was designed to ensure efficient image handling, responsive user interaction, and optimized rendering performance. The implementation process combined frontend UI development with backend image processing and error management techniques.

A. *Frontend Implementation*

The frontend interface was developed using Jetpack Compose components to create a clean, responsive, and modern application layout. Compose simplified UI development by allowing dynamic rendering and state management using less boilerplate code compared to traditional XML-based layouts.

Components such as LazyColumn and LazyVerticalGrid were used to display large collections of images efficiently while maintaining smooth scrolling performance. Cards and image containers were designed to automatically adjust according to screen size and image dimensions. Additional UI elements such as loading indicators, placeholders, retry buttons, and image captions improved overall usability and user experience.

The frontend implementation also focused on maintaining proper responsiveness across different Android devices and screen resolutions.

B. *Backend Processing*

The backend processing of the application was mainly handled by the Coil library, which managed asynchronous image downloading, caching, decoding, and rendering operations. Coil used Kotlin Coroutines to execute network and image processing tasks in the background without affecting the main application thread.

Memory caching and disk caching techniques were implemented to reduce repeated image downloads and improve rendering speed. The application also optimized image decoding to reduce memory usage while handling high-resolution images. These backend optimizations contributed significantly to improving application responsiveness and reducing performance lag.

C. *Error Management*

Proper error management mechanisms were implemented to improve application reliability and user experience. Placeholder images were displayed while images were loading to provide visual feedback to users during network operations.

In cases where image loading failed due to invalid URLs or poor internet connectivity, the application displayed error images along with retry buttons.

This ensured that users could reload failed images without restarting the application. Additional checks were included to prevent application crashes caused by image rendering failures or memory-related issues.The error handling implementation helped maintain stable application performance even under unstable network conditions.

VII. RESULTS AND DISCUSSION

The developed Image Loading App successfully loaded and displayed images from both internet URLs and local application resources while maintaining smooth UI interaction and optimized performance. The implementation of Jetpack Compose and Coil significantly improved the responsiveness and rendering efficiency of the application.

During testing, the application demonstrated fast image loading speed and stable scrolling behavior even when displaying multiple high-resolution images in list and grid layouts. The asynchronous image loading mechanism reduced UI blocking issues and prevented lag during continuous scrolling operations. Memory consumption remained optimized due to Coil's caching and efficient image decoding techniques.

The placeholder system improved the visual experience by displaying temporary loading indicators until images were fully rendered. Similarly, the retry mechanism helped users recover from loading failures caused by invalid URLs or unstable network conditions. These features improved the reliability and usability of the application.

Performance testing under different network conditions confirmed that the application maintained consistent responsiveness and smooth rendering. The integration of lazy loading components such as LazyColumn and LazyVerticalGrid further improved scrolling efficiency while handling large image collections.

A. *Results*

1) *Successful Retrieval of Mars Images*

The application successfully fetched and displayed Mars images from online sources using asynchronous image loading techniques. Coil efficiently handled image downloading and rendering without blocking the user interface. The images were loaded smoothly even under moderate network conditions, demonstrating the effectiveness of caching and background processing mechanisms.

Figure 1: Successful retrieval of Mars images

2) Importing Raw Image URL

The system supported loading images directly from raw internet URLs while handling network requests efficiently. Users were able to provide image URLs dynamically, and the application successfully rendered the images with minimal delay. Proper error handling mechanisms ensured stability even when invalid or broken URLs were provided.

Figure 2: Importing raw image URL

3) Screen Adjustment for Full Image

The application automatically adjusted image scaling and screen rendering according to different device sizes and image dimensions. Responsive Compose layouts ensured that large images were displayed correctly without distortion or overflow issues. This improved visual consistency across multiple Android devices.

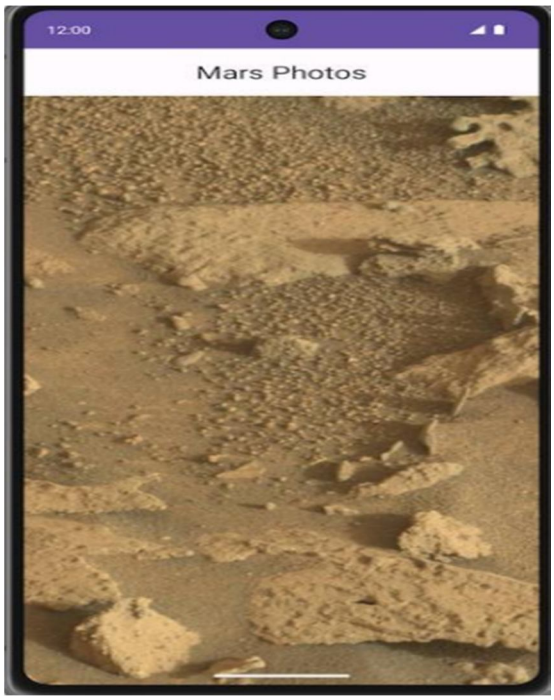


Figure 3: Screen adjustment for full image

4) Retry Button Addition

Retry functionality was implemented to reload failed image requests caused by connectivity issues or invalid image sources. When image loading failed, the application displayed an error state along with a retry option, allowing users to reload images without restarting the application. This feature improved application reliability and user experience.

Figure 4: Retry button addition

5) Final Grid Format Output

The final application successfully displayed images in optimized grid layouts with smooth scrolling and responsive UI behavior. The use of LazyVerticalGrid improved rendering performance while displaying multiple high-resolution images simultaneously. Efficient memory handling and lazy loading techniques minimized lag and ensured smooth user interaction.

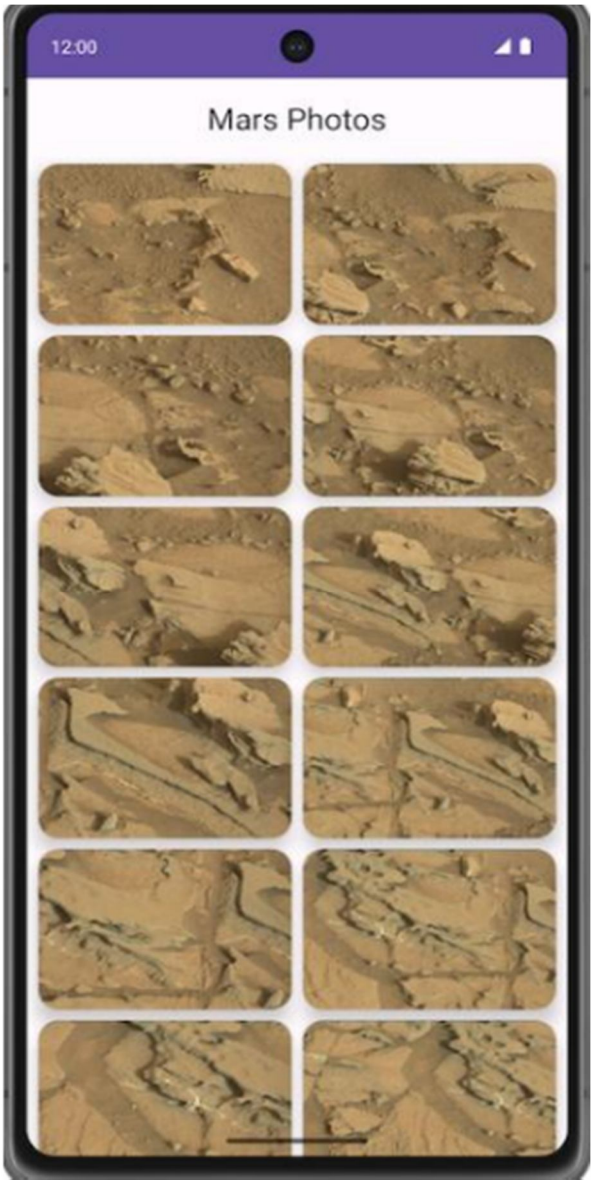


Figure 5: Final Grid Format output

B. Issues That Showed Up

Although the application performed efficiently in most scenarios, a few challenges were observed during development and testing. Loading extremely large high-resolution images occasionally increased memory usage and caused slight delays in rendering on lower-end devices. Slow or unstable internet connections sometimes resulted in delayed image loading and temporary placeholder visibility for longer durations. In certain cases, invalid image URLs caused loading failures, requiring the retry mechanism to reload the content. While scrolling through large image collections, minor frame drops were observed when multiple images were requested simultaneously under poor network conditions. Managing screen responsiveness for images with different aspect ratios also required additional UI adjustments to avoid stretching or cropping issues. Despite these challenges, the implemented caching, lazy loading, and error handling mechanisms helped maintain stable application performance and improved overall user experience.

VIII. CONCLUSIONS

The Image Loading App successfully demonstrated efficient image loading and rendering techniques using Jetpack Compose and the Coil library while providing practical understanding of asynchronous image handling, memory optimization, caching mechanisms, and responsive Android UI development. The implementation showed how modern Android development technologies can improve application performance by maintaining smooth scrolling behavior, optimized rendering, and efficient background image processing. The integration of Coil significantly reduced memory usage and improved image loading speed through effective caching and coroutine-based asynchronous operations. The project also emphasized the importance of proper error handling, placeholders, retry mechanisms, and responsive layouts in enhancing user experience and application reliability. Performance testing confirmed that the system could efficiently handle multiple high-resolution images in list and grid layouts without causing major lag or rendering issues. Overall, the project enhanced knowledge of modern Android application development practices and demonstrated how optimized image loading techniques can improve the usability, responsiveness, and performance of image-intensive mobile applications.

A. Future Improvements

The current implementation successfully demonstrates efficient image loading and rendering techniques; however, several additional features can further improve the functionality and user experience of the application.

Future improvements may include:

- Adding image zoom and gesture support for better image interaction
- Implementing offline image storage and local synchronization
- Integrating image filtering and editing capabilities
- Supporting video thumbnail loading and preview generation
- Adding AI-based image categorization and recommendation systems
- Enhancing animation and transition effects during image loading
- Implementing dark mode support and customizable UI themes
- Adding cloud synchronization and user profile management
- Optimizing image compression techniques for reduced bandwidth usage
- Supporting image sharing and download functionality

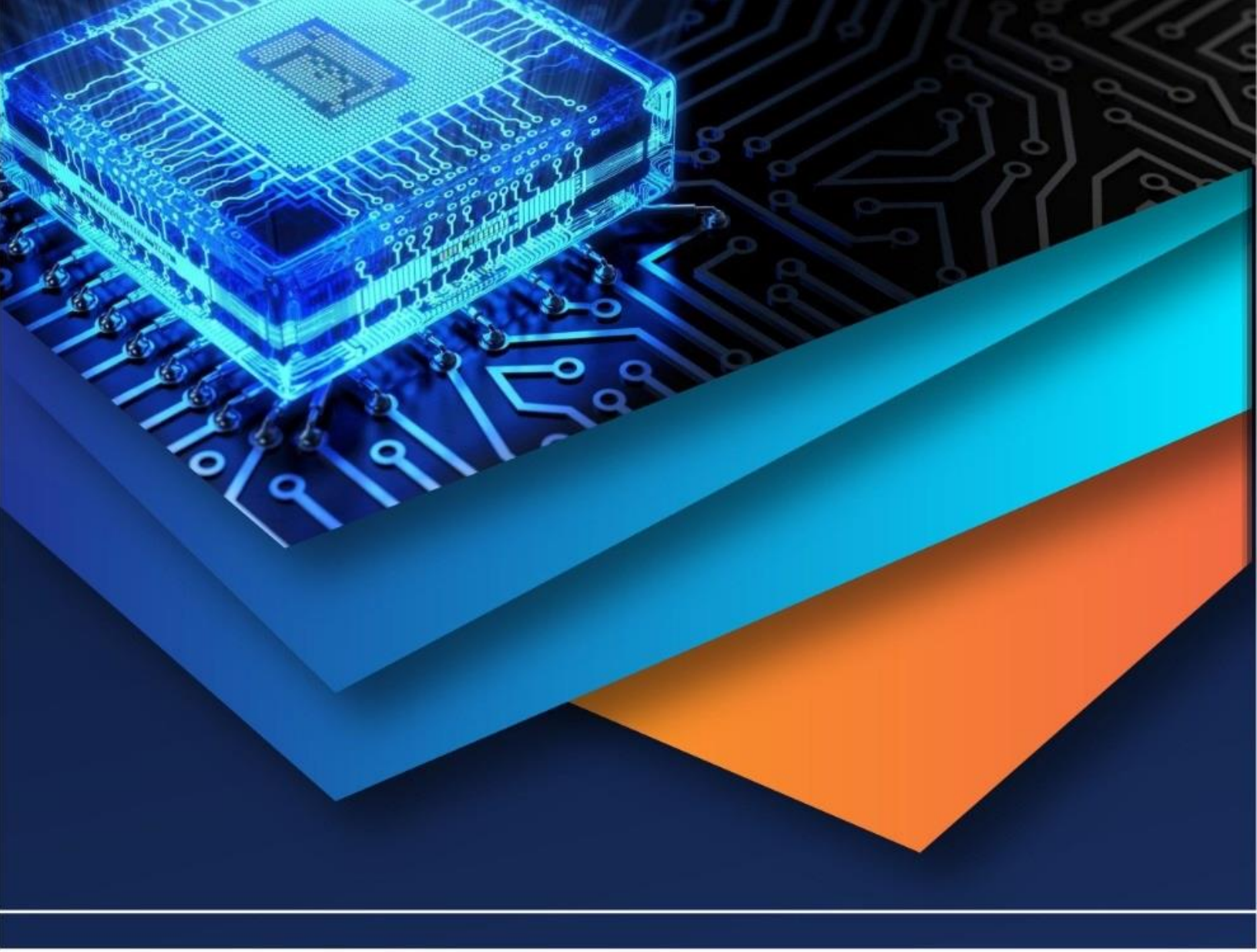
These enhancements can make the application more interactive, scalable, and suitable for real-world multimedia applications.

IX. ACKNOWLEDGEMENT

We express our sincere gratitude to our faculty members and project guides for their continuous support, encouragement, and valuable guidance throughout the development of this project. Their suggestions and technical insights greatly contributed to improving the quality and implementation of the application. We also thank our institution for providing the necessary laboratory facilities, development environment, and learning resources required for successful project completion. Special appreciation is extended to our classmates and peers who provided feedback and support during the testing and evaluation phases of the project. Finally, we would like to acknowledge the Android development community and the developers of Jetpack Compose and Coil for providing modern tools, libraries, and documentation that significantly assisted in the successful development and implementation of this application.

REFERENCES

[1] Bump Technologies, “Glide Image Loading Library for Android,” Available: https://github.com/bumptech/glide
[2] Square Inc., “Picasso: A Powerful Image Downloading and Caching Library for Android,” Available: https://square.github.io/picasso/
[3] Android Developers, “Jetpack Compose Documentation,” Available: https://developer.android.com/jetpack/compose
[4] Coil Contributors, “Coil - Coroutine Image Loader for Android,” Available: https://coil-kt.github.io/coil/
[5] Android Developers, “Lists and Grids in Jetpack Compose,” Available: https://developer.android.com/develop/ui/compose/lists



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY