INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA INFORMÁTICA

Sistema Multiplataforma de UI Adaptativa

TIAGO SILVA MACHADO

Setembro de 2025

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Multi-platform Adaptive UI System

Tiago Silva Machado

Dissertation for obtaining a Master's Degree in Computer Engineering, Specialization Area in Games, Graphics and Interactive Systems

Advisor: Helder Rodrigo Pinto [HSP]

Porto, 2025

Integrity Statement

I hereby declare that I have conducted this academic work with integrity.

I have not plagiarized or applied any form of misuse of information or falsification of results throughout the process that led to its preparation.

Therefore, the work presented in this document is original and of my own authorship and has not been used previously for any other purpose. The exceptions are explicitly recognized in the section “Ethical considerations” of the first chapter. This section also states how AI tools were used and for what purpose

I also declare that I am fully aware of P.PORTO's Code of Ethical Conduct.

ISEP, Porto, 14 de September de 2025

To my family and friends,
whose unwavering support and encouragement
have been my constant source of strength and inspiration.

All that I have achieved
is as much a testament to your support as it is to my effort.

This is ours, together.

Resumo

A presente dissertação centra-se no desenvolvimento de um sistema de interfaces adaptativas multiplataforma, com foco no módulo *Frontline Pick* da plataforma Frontline da *TeamViewer*. Este módulo é amplamente utilizado em contextos logísticos e industriais, suportando tecnologias como realidade aumentada (AR) e dispositivos portáteis, incluindo *Head Mounted Displays* (HMDs) e *smartphones*, com o objetivo de aumentar a eficiência e a precisão na execução de fluxos de trabalho.

O problema em análise resulta das limitações observadas no processo atual de criação e manutenção de interfaces de utilizador (UI) e *workflows*. Entre os principais constrangimentos identificados destacam-se a elevada complexidade de configuração, a fraca adaptação a diferentes dispositivos e a reduzida reutilização de layouts e componentes, fatores que comprometem a escalabilidade e elevam os custos de desenvolvimento.

Para mitigar estas limitações, esta dissertação propõe a utilização de *Kotlin Multiplatform* (KMP) em conjunto com o *Compose Multiplatform*, permitindo a partilha de lógica de negócio e de UI entre diferentes plataformas, sem comprometer a capacidade de adaptação às necessidades específicas de cada dispositivo. A arquitetura desenvolvida baseia-se na modularidade, na separação entre estrutura e conteúdo e na criação de layouts reutilizáveis e dinâmicos, ajustáveis a diferentes dimensões e densidades de ecrã.

Complementarmente, foram exploradas novas formas de interação, como comandos de voz e controlos externos, de forma a melhorar a usabilidade em ambientes industriais, onde a interação tátil se encontra muitas vezes limitada. O sistema proposto proporciona, assim, maior consistência visual e funcional, bem como uma experiência de utilizador uniforme e eficiente em diferentes plataformas.

Os resultados obtidos evidenciam uma redução significativa no tempo de criação de *workflows*, uma melhor compatibilidade entre dispositivos e uma diminuição das redundâncias na construção de UI. Estes resultados reforçam a importância do paradigma multiplataforma como estratégia para o desenvolvimento de soluções robustas, escaláveis e economicamente viáveis.

Por último, o trabalho contribui com um conjunto de melhores práticas de desenvolvimento multiplataforma, que podem servir de referência em projetos futuros. Estas incluem a adoção de componentes modulares, a priorização da eficiência no desempenho em dispositivos com recursos limitados e a garantia de uma experiência de utilizador consistente sem comprometer a flexibilidade necessária para diferentes cenários de aplicação.

Em suma, esta dissertação demonstra que a abordagem proposta responde eficazmente às limitações identificadas e posiciona o *Frontline Pick* como uma solução adaptável às exigências atuais e futuras do setor logístico e industrial.

Palavras-chave: Solução Multiplataforma, Interfaces Adaptativas, Kotlin Multiplatform, Workflows, Frontline Pick, Realidade Aumentada

Abstract

This dissertation focuses on the development of a multiplatform adaptive user interface system within the context of Frontline Pick, a module of TeamViewer's Frontline platform. Widely used in logistics and industrial operations, Frontline Pick supports technologies such as Augmented Reality (AR) and portable devices, including Head Mounted Displays (HMDs) and smartphones, with the goal of increasing efficiency and accuracy in workflow execution.

The challenges identified include the complexity of interface configuration, limited adaptability across different devices, and restricted reusability of layouts and components, all of which hinder scalability and increase development costs.

To address these issues, this research adopts Kotlin Multiplatform (KMP) together with Compose Multiplatform, enabling the sharing of business logic and UI components across platforms while allowing device-specific customization. The proposed architecture emphasizes modularity, separation of structure and content, and the use of reusable layouts adaptable to various screen sizes and densities.

The achieved results demonstrate a reduction in workflow creation time, improved device compatibility, and elimination of UI redundancies. Furthermore, this dissertation establishes a set of best practices for multiplatform development, contributing to the creation of scalable, efficient, and maintainable solutions.

Keywords: Multiplatform Solution, Adaptive Interfaces, Kotlin Multiplatform, Workflows, Frontline Pick, Augmented Reality

Index

1 Introduction 1

1.1 Context... 1
1.1.1 Frontline Overview 2
1.1.2 Frontline Pick 2

1.2 Problem Description 3
1.2.1 Overwhelming UI Definition Process... 3
1.2.2 Inconsistent Styling and Manual Adjustments...................................... 3
1.2.3 Lack of Device Adaptation 4

1.3 Goals 4
1.3.1 Develop a Multiplatform Solution 4
1.3.2 Simplify Workflow UI definition process 4
1.3.3 Enable Adaptive and Modular Layouts 5
1.3.4 Integrate Advanced Interaction Methods 5
1.3.5 Establish Multiplatform Development Best Practices 5
1.3.6 Limitations Associated.. 5

1.4 Stakeholders Definition and Considerations... 6

1.5 Resource Constraints... 7

1.6 Document Structure... 8
1.6.1 Chapter 1: Introduction 8
1.6.2 Chapter 2: Project Management 8
1.6.3 Chapter 3: State of the Art 8
1.6.4 Chapter 4: Design and Implementation 8
1.6.5 Chapter 5: Testing and Validations... 9
1.6.6 Chapter 6: Conclusions 9
1.6.7 References.. 9
1.6.8 Appendices 9

2 Project Management 11

2.1 Methodology.. 11
2.1.1 Research Methodology 11
2.1.2 Development Methodology.. 12

2.2 Ethical Considerations 13
2.2.1 Responsible Use of Generative AI.. 13

2.3 Skill management 13

2.4 Project Risks 14

2.5 Planning.. 14

3 State of the Art 17

3.1 Frontline Pick 17
3.1.1 Workflow Overview 18

3.2 Cross-platform Development ... 22
3.2.1 Benefits and Challenges ... 23
3.2.2 Balancing Standards, Performance, and Maintainability ... 23
3.2.3 Frameworks and Tools ... 24
3.3 Compose Multiplatform ... 31
3.3.1 Native UI vs Shared UI ... 31
3.3.2 Compose Multiplatform Overview ... 34
3.3.3 Comparison with other frameworks ... 35
3.3.4 Project Ponderations ... 37
3.4 Handling Platform-Specific Variations ... 38
3.4.1 Platform Abstractions: *expect* and *actual* ... 38
3.5 Testing for Multi-Platform Consistency ... 40
3.5.1 Automation Frameworks ... 40
3.5.2 Physical Devices vs Device Emulators ... 41
3.5.3 Performance Benchmarks ... 42
3.5.4 Feedback Validation ... 43
3.5.5 Project Ponderations ... 43
4 Design and Implementation ... 45
4.1 Architecture Definition ... 45
4.1.1 Architectural Layers ... 46
4.1.2 Integration with FrontlineBase ... 47
4.1.3 User Interface and Theming ... 48
4.1.4 Navigation Architecture ... 49
4.1.5 Translations and Asset Management ... 49
4.2 UI Development ... 49
4.2.1 Theming and Color Adaptation ... 50
4.2.2 UI Components ... 50
4.2.3 Dialog and Notification Management ... 57
4.3 Application flow ... 59
4.3.1 Authentication Flow ... 59
4.3.2 Workflow Flow ... 61
4.3.3 Settings Flow ... 61
4.4 Workflow Integration ... 62
4.4.1 Overriding FrontlineBase's UI module ... 62
4.4.2 New Workflow UI Definition ... 66
4.5 Interaction Methods ... 70
4.5.1 Voice Control Integration ... 70
4.5.2 Physical Button Navigation ... 71
5 Testing and Validations ... 75
5.1 Performance Benchmarks ... 75
5.1.1 Startup Performance Comparison ... 75
5.1.2 Application Size Comparison ... 76
5.1.3 CPU Usage Comparison ... 78
5.1.4 Memory Usage Comparison ... 79

5.2 Comparative Evaluation 79
5.2.1 Application Comparison 80
5.2.2 Workflow Comparison 84
5.3 Feedback Validation 87
5.3.1 Demographic 88
5.3.2 Methodology 89
5.3.3 Application related feedback 89
5.3.4 Workflow related feedback 91
5.3.5 Open feedback 93
6 Conclusions 95
6.1 Contributions 95
6.2 Achieved Goals 96
6.3 Limitations 97
6.4 Future Work and Potential Enhancements 98
6.5 Final Remarks 98
References 99
Appendix A. Project Risks 106
Appendix B. Project Planning 110

Figure Index

Figure 1. Samsung SDS boosts picking speed by 30% with Frontline Pick [4]. 18
Figure 2. Creator with a simple workflow. 19
Figure 3. UI Definition on Creator, with the UI structure on the left panel. 20
Figure 4. Combining *LayoutPage* and *LayoutModel* [10]. 22
Figure 5. Flutter's architecture overview [22] 25
Figure 6. JavaScript's architecture overview [22] 25
Figure 7. Kotlin Multiplatform flexibility in project architecture [24] 26
Figure 8. APK/IPA size in megabytes, lower = better [35] 36
Figure 9. Startup time on a Pixel 4a and iPhone 12 Mini in milliseconds, lower = better [35]. 36
Figure 10. Picking Client Architecture 47
Figure 11. Outlined, primary and text buttons, respectively, for smartglasses (top) and smartphones (bottom) 51
Figure 12. Home Screen on smartglasses (on the left) and smartphones (on the right)........... 52
Figure 13. Dialog design for smartglasses (on the left) and smartphones (on the right) 53
Figure 14. Side drawer running on an Android Device 54
Figure 15. Notification design for the smartglasses (on top) and smartphones (on the bottom) 55
Figure 16. Picking Cards States for Idle, Highlighted and Disable, respectively, for smartglasses (left) and smartphones (right) 56
Figure 17. Scanner page running on an Android device 57
Figure 18. Application flow overview 59
Figure 19. Login Page running on iOS (on the left) and on a Vuzix M300 (on the right) 60
Figure 20. Settings Page running on an iPhone 16e (on the left) and on a Vuzix M300 (on the right) 62
Figure 21. Sequence Diagram of the Workflow UI Renderer 65
Figure 22. Example of a LayoutPage definition and its rendered user interface....................... 67
Figure 23. Adaptiveness of the LayoutPage: layout definition and rendered user interface of the Exceptions page 68
Figure 24. Example of a NativePage definition and its rendered interface. 69
Figure 25. Startup performance comparison between iPhone 16e, Vuzix M300 and Samsung A52 76
Figure 26. App size performance comparison (in MB) between Android and iOS 77
Figure 27. Peak CPU utilization across devices during workflow execution............................. 78
Figure 28. Peak memory usage across devices during workflow execution (in MB) 79
Figure 29. Landing Page on the Frontline Workplace app (bottom-right) and on the Picking Client app for smartphone (left) and smartglasses (top-right)... 80
Figure 30. Login Scanner on the Frontline Workplace app (top) and on the Picking Client app for smartphone (bottom-left) and smartglasses (bottom-right) 81
Figure 31. Settings Page on the Frontline Workplace app (top-right) and on the Picking Client app for smartphone (left) and smartglasses (bottom-right) 82

Figure 32. Info Menu on the Frontline Workplace app (top-right) and on the Picking Client app for smartphone (left) and smartglasses (bottom-right) 82
Figure 33. Device Settings Page on the Frontline Workplace app (top) and on the Picking Client and smartglasses (bottom) 83
Figure 34. Reset Configuration Dialog on the Frontline Workplace app (bottom-right) and on the Picking Client app for smartphone (left) and smartglasses (top-right) 84
Figure 35. Files required to define the UI shown in the workflow on the right........................ 84
Figure 36. UI definition of the workflow shown in Figure 37 85
Figure 37. Workflow example running on iOS (left), smartglasses (top right), and Android phones (bottom right) 86
Figure 38. Example of paged content navigation 86
Figure 39. Demographic from the feedback form 88
Figure 40. Level of experience with Frontline on the feedback form...................................... 89
Figure 41. Devices used for the feedback form 89
Figure 42. Visual Consistency chart from the feedback form 90

Table Index

Table 1. Overview of key stakeholders and their interests in the Frontline Pick 6
Table 2. Comparison of Cross-Platform Frameworks in Terms of Performance. 26
Table 3. Use Cases and Development Architecture of Frameworks. 28
Table 4. UI Consistency and Platform-Specific Adaptation in Frameworks.............................. 29
Table 5. Comparison overview of Shared and Native UI in Kotlin Multiplatform 33
Table 6. Comparison: Native UI, Compose Multiplatform, and Flutter.................................... 37

Code Blocks Index

Code Block 1. Example of a PartTemplate [10]. 21
Code Block 2. Example of a *LayoutPage* [10]. 21
Code Block 3. Example of a *LayoutModel*. 22
Code Block 4. Example of a *Style* [10].. 22
Code Block 5. Text component using the *InfoContentStyle* defined earlier. 22
Code Block 6. Common *expect* class that defines the methods to be implemented on each platform.. 38
Code Block 7. Android-specific implementation of the *FileReader* class using Android file system APIs.. 39
Code Block 8. iOS-specific implementation of the *FileReader* class using iOS file system APIs. 39
Code Block 9. Common code usage that accesses platform-specific implementations through the *expect* API.. 40
Code Block 10. ScannerCamera and Camera Permission State.. 56
Code Block 11. Scanner Composable Usage.. 57
Code Block 12. example of button mapping configuration in *app_config.yaml* 72

Notation and Glossary

AI	Artificial Intelligence
API	Application Programming Interface
AR	Augmented Reality
CPU	Central Processing Unit
DI	Dependency Injection
FCC	Frontline Command Center
HMD	Head Mounted Display
JSON	JavaScript Object Notation
KMP	Kotlin Multiplatform
MDM	Mobile Device Management
PE	Project Expert
UI	User Interface
UML	Unified Modeling Language
UX	User Experience
XML	Extensible Markup Language

1 Introduction

The development of user interfaces (UI) that adapt seamlessly across diverse devices and platforms has become an essential requirement in modern software engineering. As industries increasingly rely on digital solutions to enhance operational efficiency, the need for scalable and adaptive UI systems has grown significantly. This thesis focuses on addressing these challenges within the context of Frontline Pick, a module of the Frontline platform developed by TeamViewer, which integrates AR and wearable devices to optimize workflows in logistics and industrial environments.

The introduction chapter outlines the scope and context of this thesis, beginning with a comprehensive overview of the Frontline platform and its applications in various operational workflows. The chapter further delves into the specific challenges faced in the existing UI definition and workflow creation processes, including complexity, limited device adaptation, and redundancy in component reusability. These issues highlight the need for an innovative, multiplatform solution to improve both the user experience and the development process.

Finally, the goals of the thesis are articulated, emphasizing the development of an adaptive, modular, and multiplatform UI solution. By leveraging Kotlin Multiplatform and Compose Multiplatform, this research aims to propose best practices, address inefficiencies, and lay the foundation for scalable and maintainable UI systems. This chapter sets the stage for the subsequent sections, providing a structured approach to the problem and its proposed solution.

1.1 Context

TeamViewer, widely recognized for its remote support solution [1], is a company that has expanded its offerings to include Frontline, a platform designed to improve operational efficiency in industries where manual labor plays a crucial role [2]. While TeamViewer is best known for providing remote desktop access and technical assistance, Frontline represents the company’s foray into solutions tailored to logistics, warehousing, manufacturing, and field services.

1.1.1 Frontline Overview

Frontline integrates augmented reality, wearable devices, and real-time data analytics to provide frontline workers with the information and tools they need to optimize their tasks. These solutions are delivered via various devices, including wearable scanners, smart glasses, tablets, and smartphones, allowing workers to interact with both the physical and digital aspects of their environment, leading to increased accuracy and efficiency.

The Frontline platform is divided into separate solutions to cater to the specific needs of different operational tasks. Each solution focuses on distinct aspects of industrial workflows, ensuring that the technology is tailored to the unique requirements of each task [2], [3]:

1. **xPick**: A hands-free picking solution utilizing AR technology to guide warehouse workers through the order picking process, improving both speed and accuracy.

2. **xMake**: A tool that provides step-by-step guidance for assembly and quality assurance processes, ensuring consistency and adherence to operational standards.

3. **xInspect**: A solution designed to streamline maintenance and inspection workflows by offering real-time guidance and capturing data for reporting and compliance purposes.

4. **xAssist**: Enables remote experts to assist on-site workers by providing real-time guidance for complex tasks, bridging the gap between the field and remote support.

5. **Frontline Command Center (FCC)**: A centralized platform that allows managers to monitor and track workflows, analyze key performance metrics, and make informed decisions to enhance overall operations.

These separate solutions are designed to address specific workflows within an organization. By breaking down the platform into specialized components, Frontline offers flexibility and scalability, allowing businesses to select and implement the tools that best fit their operational needs. Each solution is optimized for a particular function, whether it’s optimizing the picking process, improving assembly accuracy, or providing expert remote assistance, which ensures that the platform delivers maximum value to users in diverse industrial environments.

1.1.2 Frontline Pick

This thesis focuses on xPick, also known as Frontline Pick, which will be referred to as such throughout the document. The picking process, central to supply chain management, involves retrieving items from storage to fulfill orders. Traditional methods are often inefficient, error-prone, and labor-intensive, highlighting the need for innovative solutions like Frontline Pick.

The solution allows for the creation and deployment of customized workflows tailored to the specific requirements of an organization. These workflows are presented to workers on devices such as smartglasses, wearable scanners, or even smartphones. By providing step-by-step

guidance through augmented reality overlays, the technology reduces reliance on manual processes, thereby minimizing errors and improving overall productivity.

Frontline Pick is designed to support a range of critical tasks in supply chain operations. It enhances accuracy and efficiency in retrieving items during order fulfillment (Order Picking), simplifies stock counts and audits in inventory management (Inventory Management), and ensures proper packing and labeling during shipment preparation (Shipment Preparation)[2].

While Frontline Pick offers flexibility and customization, making it valuable across various industries, its current design presents challenges. These challenges particularly affect the user experience and operational efficiency, especially when it comes to defining workflows and their corresponding user interfaces.

A more comprehensive overview and analysis of the current system, including its challenges and limitations, will be presented in the State of the Art chapter, where a deeper investigation into the current workflow definition will be conducted (Chapter 3).

1.2 Problem Description

Frontline Pick is a highly customizable platform, allowing businesses to configure workflows that match their specific needs. However, this flexibility comes at a cost, as the current implementation lacks streamlined solutions for creating and managing workflows. The current implementation is relatively outdated, and with advancements in technology, it is now feasible to design a more efficient and user-centric solution. While these challenges have persisted over time, they have become increasingly pronounced due to evolving business requirements and the growing demand for intuitive and adaptable tools. The main challenges include:

1.2.1 Overwhelming UI Definition Process

The platform requires users to manually define a variety of elements for each workflow, including UI components, layout pages, playout models, and styling. This process becomes convoluted as all these elements are interwoven, making it difficult to manage and leading to confusion. The need to repeatedly define and customize these aspects for each workflow adds unnecessary complexity.

1.2.2 Inconsistent Styling and Manual Adjustments

Styling is manually applied throughout the platform, with paddings, weights, and other design elements defined individually in many places. This approach results in a scattered design system that’s difficult to maintain. In the long term, any changes to the styling require updates in multiple locations, creating a high maintenance burden. Additionally, the platform's styling is not fine-tuned for the best user experience, as it's not developed with input from professional

designers. This lack of design expertise leads to a suboptimal user interface that doesn't provide the seamless experience businesses need.

1.2.3 Lack of Device Adaptation

The platform's UI does not adapt dynamically to different screen sizes or device types. This means that workflows optimized for one device, such as a Head Mounted Displays (HMDs), may not translate well to mobile devices, or specialized equipment like wearable scanners. As a result, businesses must manually adjust their workflows for each device, significantly increasing development and maintenance time. For clients using iOS devices, a completely separate solution must be developed, further complicating the process and increasing overhead.

1.3 Goals

The primary goal of this thesis is to explore and implement a comprehensive multiplatform solution that resolves inefficiencies and challenges in workflow creation and user interface optimization for the Frontline Pick client. By addressing these issues, the proposed solution seeks to improve user experience, streamline development processes, and ensure scalability across a range of devices and platforms.

1.3.1 Develop a Multiplatform Solution

A key focus of this research is to evaluate and implement a robust framework that supports both Android and iOS environments while maintaining optimal performance across various devices. These devices include smartphones, commonly used in logistics, but also specialized hardware like HMDs, which are increasingly employed in industrial settings for hands-free operation and AR capabilities.

1.3.2 Simplify Workflow UI definition process

Another critical objective is to simplify and accelerate the process of defining workflow user interfaces. The current approach requires multiple interdependent files and manual configuration of styles, layouts, and components, which increases complexity and maintenance effort. The proposed solution aims to introduce a unified schema that consolidates these elements into a single structure, reducing redundancy and improving clarity. This will enable faster workflow creation, minimize errors, and ensure consistency across different workflows and devices.

1.3.3 Enable Adaptive and Modular Layouts

The solution aims to facilitate the creation of adaptive UI layouts capable of adjusting seamlessly to diverse types of devices. This includes accommodating a wide range of screen sizes, from small handheld devices to larger ones, and supporting unconventional aspect ratios often found in specialized hardware. The adaptability must also consider unconventional aspect ratios commonly found in specialized hardware and maintain visual clarity across different screen densities. Furthermore, the solution will account for devices with distinct interaction capabilities, such as touch-based smartphones versus non-touch devices like HMDs. The behavior of the UI should adjust accordingly to provide intuitive and efficient interactions for each device type. Emphasizing modularity and reusability, the design approach will allow these layouts to be easily adapted or reused across workflows and device types.

1.3.4 Integrate Advanced Interaction Methods

To enhance usability and efficiency, the refactored solution will explore innovative interaction methods. Features such as voice commands should be integrated to enable hands-free operation, especially in environments where touch interaction is impractical. Support for external hardware controls, including hardware buttons on wearable devices and HMDs, will also be prioritized to streamline input processes.

1.3.5 Establish Multiplatform Development Best Practices

The project should define and document best practices for multiplatform development, emphasizing key aspects such as UI adaptation, modularity, and performance optimization. Standardized methods for creating responsive and adaptive UIs will ensure consistent user experiences across devices. Modular UI components will reduce redundancy, facilitating efficient development of future workflows. Performance optimization will target the constraints of industrial devices, such as limited processing power and battery life, ensuring reliable and efficient operation.

By addressing these goals, the proposed solution will significantly reduce workflow creation time, eliminate redundant UIs, and improve device compatibility. This will result in a more scalable, user-friendly, and efficient platform for Frontline Pick, positioning it to meet the demands of modern logistics environments.

1.3.6 Limitations Associated

The objectives of this thesis are constrained by several factors. The solution must support a wide variety of devices, including smartphones, and specialized hardware like HMDs, each with different screen sizes, aspect ratios, and input methods. This requires an adaptive UI design, which, while necessary, limits customization and flexibility.

Performance optimization is another key challenge, as many industrial devices have limited processing power and battery life. This constraint restricts certain features and necessitates careful management of resources.

Furthermore, the integration of advanced interaction methods, such as voice commands and external hardware controls, introduces additional limitations. These features must be compatible across multiple platforms and user environments, which adds complexity to their implementation.

1.4 Stakeholders Definition and Considerations

In the context of project development, identifying and managing stakeholders is a critical aspect of ensuring the successful implementation and long-term sustainability of the project. Stakeholders, defined as individuals or groups with a vested interest in the project's outcome, vary significantly in terms of their roles, influence, and involvement. This section presents a comprehensive overview of the key stakeholders involved in the development and deployment of the Frontline Pick system, classified according to their level of influence and contribution to the project's success.

The stakeholders involved in this project encompass both high-level decision-makers and those engaged in the day-to-day use of the system. Each stakeholder group brings distinct perspectives, priorities, and requirements, all of which must be addressed to achieve a cohesive and effective solution. The interests of these stakeholders must be carefully considered throughout the project's lifecycle to ensure alignment with both operational and strategic goals.

Table 1**.** Overview of key stakeholders and their interests in the Frontline Pick

Stakeholder	**Role**	**Key Interests**	**Considerations**
Company Executives	Decision-makers	ROI, strategic alignment, competitive advantage	Operational cost savings, innovative features, regular progress updates.
Shareholders	Financial backers of the company	ROI, project feasibility, long-term value generation	Regular financial updates, cost-saving strategies, alignment with market trends to ensure strong returns.
Partners	Technology providers (e.g., smart glasses, ProGlove)	Integration of their hardware into workflows, showcasing	Seamless integration with devices, optimized use of partner hardware, mutual promotion opportunities.

Stakeholder	Role	Key Interests	Considerations
		the effectiveness of their solutions	
Project Experts (PEs)	Workflow designers and implementers	Understanding workflow concepts, tailoring workflows to meet specific requirements	Collaborative design process, adaptability to client needs, effective implementation strategies
Warehouse Managers	Supervisors managing operations	Workflow visibility, performance analytics, integration with inventory systems	Reporting tools, configurable workflows, integration with third-party systems.
Warehouse Operators	End users of the system	Efficient workflows, error reduction, device compatibility	Adaptive UI, simplified workflow creation, tailored UIs for tasks and devices.
Internal Teams	Development, IT, and support teams	Clear goals, robust tools, scalability, and security	Agile methodologies, Kotlin Multiplatform for updates, modular and secure system design.

Project Experts and Warehouse Operators are the most affected stakeholders when using the current Frontline Pick. Project Experts are significantly affected by the current complexity of defining and customizing workflows. The lack of flexibility and streamlined processes limits their ability to design solutions quickly and efficiently, making it harder to meet diverse client needs and adding unnecessary overhead.

Warehouse Operators, as the end users, are most directly impacted. The system's poor adaptability to different devices and its lack of a simplified UI lead to inefficiencies and frequent errors in daily operations. This reduces productivity and increases the time spent managing workflows, underlining the need for a more user-friendly and adaptable solution that ensures smoother operations across various devices.

1.5 Resource Constraints

Addressing the challenges in Frontline Pick requires navigating financial and human resource constraints. Costs include the allocation of designers to tailor components and layouts for optimal performance across multiple devices and developers to implement these improvements. The platform's reliance on legacy systems poses technological hurdles,

including integration with modern frameworks and achieving cross-platform compatibility. Limited team expertise in areas like UI/UX design and Kotlin Multiplatform further complicates development. Additionally, time constraints, stakeholder alignment, and resistance to change may impact the scope and pace of the solution.

1.6 Document Structure

This document is organized into six primary chapters, along with references and appendices, to provide a comprehensive overview of the project's objectives, methodology, and outcomes. Below is a summary of each chapter:

1.6.1 Chapter 1: Introduction

The introduction establishes the context for the project, and outlining the goals of the project. It also identifies the key stakeholders and provides insights into their needs and expectations.

1.6.2 Chapter 2: Project Management

This chapter outlines the project's management framework, including the methodologies adopted, ethical considerations, risk analysis, and planning details. It ensures the project is well-structured and aligned with best practices for efficient execution.

1.6.3 Chapter 3: State of the Art

An in-depth exploration of cross-platform mobile development forms the core of this chapter. It first gives a brief overview on what Frontline Pick is and its workflow definition, which is crucial to better understand the needs of the project. Following this, it covers the benefits and challenges, technology frameworks, tools, and Compose Multiplatform. It also examines how platform-specific variations are handled and the testing methodologies employed to ensure consistency across platforms.

1.6.4 Chapter 4: Design and Implementation

The implementation chapter focuses on the technical aspects of the project. It defines the architecture, details the UI development process, discusses workflow integration, and describes advanced interaction methods.

1.6.5 Chapter 5: Testing and Validations

This chapter highlights the validation process, including performance benchmarks, testing validations, and user experience evaluations. A comparison with current applications is also provided to demonstrate the improvements achieved.

1.6.6 Chapter 6: Conclusions

The final chapter reflects on the project's outcomes, summarizing the contributions made, goals achieved, and limitations encountered. It also identifies future work and potential enhancements to guide further development

1.6.7 References

This section includes all the sources, materials, and tools referenced throughout the document. It provides readers with the necessary resources to explore the concepts, methodologies, and technologies discussed in greater depth.

1.6.8 Appendices

The appendices contain supplementary materials that support the main content of the document. These include additional details, tables, or visual aids that provide further clarity and context to the topics discussed in the core chapters.

2 Project Management

This chapter outlines the framework and strategies employed to ensure the successful execution of the project. It provides a comprehensive overview of the methodologies adopted, ethical considerations observed, and the approach to managing skills and resources effectively. Additionally, it addresses potential project risks and details the planning process that guided the development lifecycle.

2.1 Methodology

This thesis employs a dual approach combining research and development to address the challenges of creating adaptive, cross-platform solutions for Frontline Pick workflows using Kotlin Multiplatform and Compose Multiplatform. The process is focused on both understanding the technical and theoretical underpinnings of the technologies involved and applying this knowledge to develop scalable and efficient solutions tailored to diverse devices and use cases.

The research aims to provide a foundational understanding of existing methodologies, tools, and challenges in cross-platform development. The development focuses on implementing and testing solutions, emphasizing iterative improvement based on experimental findings.

2.1.1 Research Methodology

The research conducted in this thesis follows the Design Science Research (DSR) methodology. DSR focuses on solving practical problems through the creation and evaluation of artifacts that address identified needs. The chosen methodology is particularly suitable for this thesis as it not only seeks to understand the challenges of implementing scalable cross-platform solutions but also aims to produce artifacts that can be applied to similar contexts in the future.

The research methodology employed in this thesis was structured to ensure the acquisition of credible and relevant results, supported by both scientific and practical sources.

The central topics of the research included Kotlin Multiplatform, Compose Multiplatform, cross-platform UI scalability, workflow optimization, and device-specific adaptation strategies. These topics provided a clear direction for the investigation and helped focus the research on critical aspects, avoiding the dispersion of efforts into less relevant areas.

Due to the recent emergence of Kotlin Multiplatform and Compose Multiplatform, which have only recently exited beta, there is a limited body of academic literature on these technologies, particularly concerning the use of shared user interfaces approach when developing in Kotlin Multiplatform and Compose Multiplatform. To address this gap, the research incorporated not only peer-reviewed articles, books, and conference proceedings but also non-bibliographic sources such as official documentation and developer blog posts. These non-traditional sources are vital for understanding the state of the art and best practices for these emerging technologies.

The research was conducted using well-known academic repositories like Google Scholar and ScienceDirect, alongside reputable online platforms such as the Kotlin Blog, JetBrains official documentation, and Compose Multiplatform community resources. Specific keywords were used for each topic, such as "Kotlin Multiplatform," "Compose Multiplatform," "Cross-Platform UI," "Platform agnostic UI" and "Device-Specific Adaptation," ensuring comprehensive coverage of the topics and identification of relevant sources.

To ensure the quality of the information, priority was given to sources with significant impact in their respective fields, including highly cited papers, well-regarded books, and official technical documentation. For non-bibliographic sources, credibility was assessed based on the reputation of the publisher, the technical accuracy of the content, and its alignment with other verified sources.

2.1.2 Development Methodology

The development process for this thesis adopts the Agile Scrum methodology, which is well-suited for iterative and flexible development. The project will be carried out in collaboration with the company, beginning with an assessment of the requirements for completion. This initial phase involves working closely with stakeholders to define the project's scope, objectives, and key deliverables, ensuring alignment with both the thesis goals and the company's needs.

Once the requirements are established, a product backlog is created with prioritized tasks. These tasks are then divided into smaller, manageable pieces and organized into sprints, each focusing on specific deliverables. At the start of each sprint, tasks are planned, assigned, and broken into subtasks, with progress regularly reviewed and adjustments made to address any challenges encountered. Each sprint will have the duration of two weeks.

2.2 Ethical Considerations

This project adheres to the General Data Protection Regulation (GDPR), ensuring that all personal data is handled responsibly. It is important to note that all data concerning future users will be managed by TeamViewer in accordance with GDPR standards. The focus of this project is not on collecting user data but rather on improving the Frontline Pick value, by tackling the workflow creation process and enhancing the user experience of the current picking mobile application with better end results for those interacting with the application.

Collaboration with TeamViewer ensures that the project complies with internal company policies and confidentiality standards. All deliverables and findings are reviewed by TeamViewer before being shared externally, ensuring that any shared materials are approved, and that sensitive information will be safeguarded throughout the development process.

2.2.1 Responsible Use of Generative AI

During the development and writing of this dissertation, Generative Artificial Intelligence (AI) was employed as a supportive tool. In the development phase, AI was used to suggest strategies and possible approaches to technical challenges, offering alternative perspectives that informed the decision-making process. In the writing phase, it provided assistance in structuring sections, improving clarity, and refining the formulation of complex ideas.

All AI generated content and suggestions were carefully reviewed, validated, and adapted by the author to ensure correctness, scientific integrity, and alignment with the objectives of the research. The use of generative AI was therefore conducted in an ethical and responsible manner, functioning strictly as an aid to enhance efficiency and clarity, without replacing the author's original contributions, critical analysis, or reasoning.

2.3 Skill management

Skill management is an essential part of ensuring this project's success, focusing on identifying, developing, and applying the necessary expertise at every stage. The technical skills required include proficiency in Kotlin Multiplatform, particularly Compose for Multiplatform, XML parsing, and workflow automation design. Additional technical competencies include working with Agile Scrum methodology, version control using Git, and resource optimization for multiplatform solutions. Non-technical skills include research documentation, stakeholder communication, and project management.

To address potential skill gaps, training and self-learning opportunities will be leveraged. For example, resources on advanced Kotlin features and Compose integration for cross-platform adaptability will be made available. Tasks are planned to correspond with these skills, ensuring that the project progresses systematically while fostering individual growth. Collaboration with

TeamViewer ensures that domain-specific insights and feedback are incorporated into the workflow design process. Skill management is revisited during sprint reviews to reassess and align the team's capabilities with project demands.

2.4 Project Risks

Effective project management involves identifying, analyzing, and addressing potential risks that could impact the success of the project. Risks can arise from a variety of sources, including technical challenges, resource limitations, stakeholder expectations, or process inefficiencies.

This document highlights key risks associated with the project, categorized by their causes, effects, and mitigation strategies. A systematic approach has been employed to assess the probability and impact of each risk, to prioritize responses and allocate resources effectively.

For detailed information, including risk descriptions, their associated causes and effects, probability and impact scores, and planned response strategies, refer to Appendix A. Project Risks.

2.5 Planning

A comprehensive project plan is integral to tracking progress, identifying dependencies, and ensuring the timely completion of deliverables. The Gantt chart, presented in Appendix B. Project Planning, provides a detailed schedule that outlines the sequential tasks and phases required for the project.

The project is divided into two main phases: Introduction and State of the Art and Development, each encompassing a series of tasks with specific objectives. The first phase focuses on foundational activities, including research planning, problem identification, solution analysis, and documentation. These tasks ensure a robust understanding of the project's context and set the groundwork for subsequent development efforts. The second phase transitions into technical implementation, addressing tasks such as architecture design, UI development, workflow integration, performance optimization, and validation through rigorous testing.

The Gantt chart also reflects the necessary skills to be applied throughout the project. For instance, the early research-focused tasks demand strong analytical and comparative abilities, while the later development-oriented tasks require technical proficiency in design, coding, and optimization. By aligning the schedule with these skills, the project plan ensures that the required expertise is deployed effectively across all stages.

Additionally, monitoring and control procedures are integrated into the plan to safeguard project quality and alignment with objectives. Specific tasks, such as documentation reviews and testing cycles, serve as checkpoints to evaluate progress and address potential issues.

It is important to note that the associated expenses are limited to the essential personnel involved. Specifically, the project entails the development efforts of the assigned developer and the contribution of a designer responsible for creating the necessary designs.

For a full breakdown of the project schedule and task breakdown, refer to Appendix B. Project Planning, which includes the full project plan in Gantt chart format.

3 State of the Art

This chapter delves into the foundational concepts of cross-platform development, with a particular focus on Frontline Pick and its workflow definition. As a core component of this project, Frontline Pick's workflow plays a pivotal role in the success of the proposed solution. A thorough understanding of its structure and operation is crucial, as the project's goal is to enhance and adapt this workflow to address existing limitations and improve efficiency.

Building upon this foundation, the chapter explores the broader landscape of cross-platform development, highlighting the motivations behind adopting these frameworks and the benefits they bring in terms of scalability and code reuse. It also addresses critical challenges, such as balancing performance, maintainability, and adherence to standards. The discussion transitions to Compose Multiplatform, examining its capabilities and handling platform-specific variations. The chapter concludes by evaluating technologies specific to the project's needs and emphasizing the importance of testing to ensure consistency across platforms.

3.1 Frontline Pick

Frontline Pick is an advanced solution designed to optimize supply chain operations, particularly in order picking, inventory management, and shipment preparation. It leverages devices like smart glasses, wearable scanners and smartphones, sometimes enhanced with augmented reality (AR) overlays to provide real-time, step-by-step guidance. This minimizes errors, reduces manual effort, and increases productivity [2].

Central to the product is the workflow definition, which serves as the foundation of its functionality. These workflows are customized to specific organizational needs, guiding workers through tasks via intuitive interfaces. Once defined and deployed, workflows integrate with devices, providing users with clear, actionable instructions. The final implementation can resemble the interface shown in Figure 1, where workflows are presented interactively to facilitate efficient task execution.

Before diving into specific aspects of the project, it’s important to understand what Frontline Pick is and how its workflow definition currently functions.

Figure 1. Samsung SDS boosts picking speed by 30% with Frontline Pick [4].

3.1.1 Workflow Overview

In the context of this thesis, it is essential to understand how a workflow for Frontline Pick is created. This is done through the Creator, a central tool within the FCC. The Creator is used to design and configure workflows, providing a flexible and scalable approach that allows for efficient updates and deployment across different sites and use cases [5]. This process can be divided into two key stages: defining the workflow logic and configuring the workflow UI. Due to its complexity, this process is usually done by PEs provided by TeamViewer. These experts collaborate with companies to understand their desired workflow concepts and then design and implement workflows tailored to meet those specific requirements.

In the Creator, users define the workflow logic by setting up actions, transitions, and conditions that drive the workflow's flow. This ensures that tasks, such as scanning items and validating data, are completed in a structured manner. Additionally, the UI configuration within the Creator allows for designing the visual layout, defining the elements users will interact with, and configuring dynamic updates that guide users through the picking process. Together, these stages ensure the workflow is both functional and user-friendly, with each step tailored to the specific needs of the task at hand. In the following sections, a detailed exploration of these stages will highlight their importance in creating an effective picking workflow [2], [5].

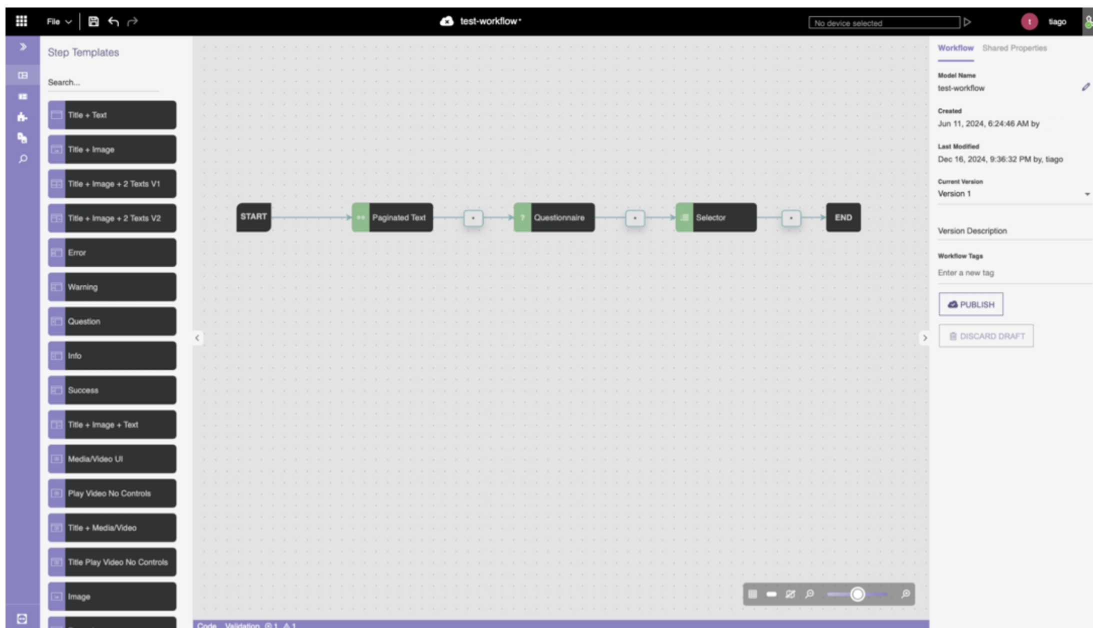


Figure 2. Creator with a simple workflow.

3.1.1.1 Workflow Logic

Workflow logic is essentially the blueprint of how the workflow will proceed step-by-step. This involves defining actions, transitions, and handling conditions that determine the flow from one stage to another. The logic is built using a combination of actions and transitions that are condition-based and guide users through specific tasks, such as scanning and validating items in a warehouse.

At the core of the workflow logic are ***actions***, this represent individual tasks or events that need to be executed during the workflow [6]. For example, a *scan action* might trigger the system to prompt the user to scan an item, or a *data validation action* could check if the scanned barcode matches a stored record. Actions are modular, meaning that they can be reused and combined to form more complex workflows. Each action could have different types, such as UI updates, data validation, or triggering external system calls like fetching data from a database or communicating with hardware devices (e.g., barcode scanners).

Transitions are the next key element in workflow logic. They determine how the workflow moves between actions based on the outcome of previous steps [7]. These transitions can be conditional, meaning they only occur if a certain condition is met (e.g., if the barcode scan is successful). For instance, after an item is scanned, if the barcode is correct, the workflow could transition to the next task, such as confirming the item's pick. However, if the barcode is invalid, the workflow might loop back to the scanning step. Thus, transitions allow the workflow to adapt dynamically to different situations, providing a guided and responsive user experience.

Finally, ***mapping*** is another very important element, which connects specific UI elements to the actions in the workflow logic. The mapping process in the Creator involves associating the UI elements with workflow steps defined earlier in the logic stage [8], [9]. For example, a scan

button could be mapped to trigger the scan item action, while a confirmation button might be mapped to proceed to the next step of the workflow, like confirming a pick. This process also includes defining the behavior of elements when certain conditions are met. For instance, if an error occurs during scanning (e.g., a wrong item is scanned), the system could map the error state to display a specific message on the screen, guiding the user to retry the action.

3.1.1.2 Workflow UI

Defining the UI for workflows in Frontline is a structured process that separates styling, layout, and component logic, enabling developers to create adaptable, reusable, and visually consistent interfaces. The UI definition revolves these four key elements: styles, part templates, layout pages, and layout models, each playing a distinct role in constructing the user interface. These definitions are written in XML format, allowing for clear and structured organization. Moreover, they can be stored within a dedicated folder called ***Resources/SharedLayout*** inside the workflow definition. This folder centralizes and shares these elements across the entire workflow, ensuring reusability and consistency.

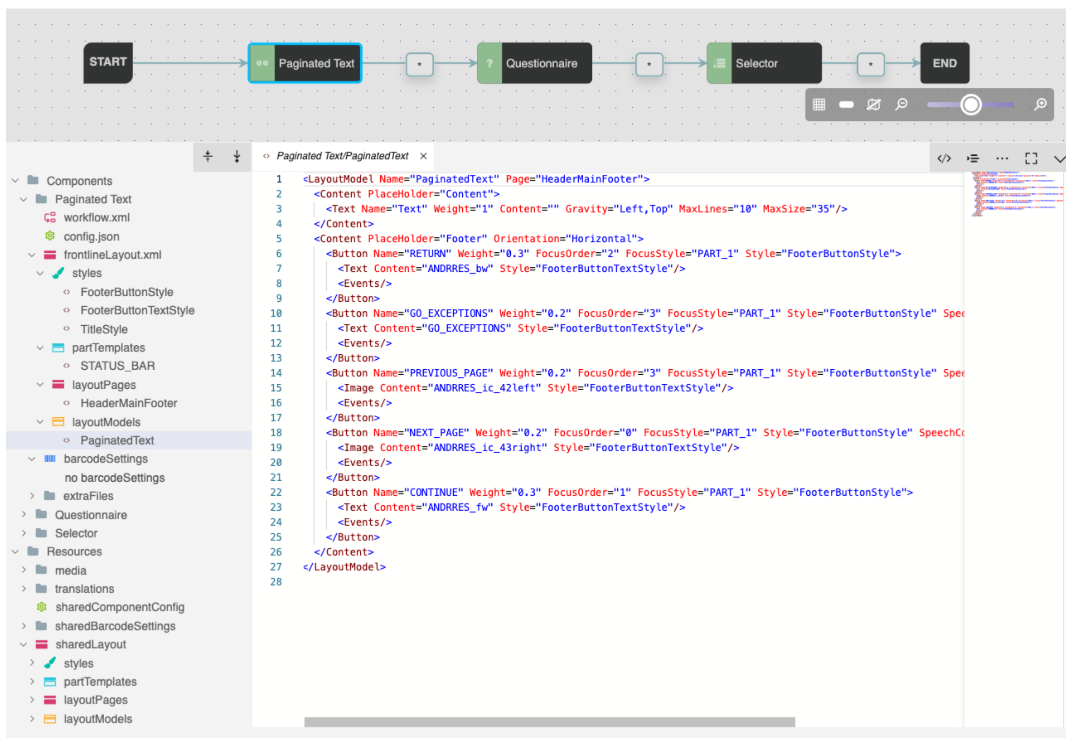


Figure 3. UI Definition on Creator, with the UI structure on the left panel.

Part templates are reusable within the step they are being defined on, predefined modules that allow developers to define UI components once and use them consistently across multiple views [10]. These templates encapsulate common elements such as headers, status bars, or footers, ensuring that changes to a template automatically reflect wherever it is used. For example, a *status bar* can be defined as a reusable part template that contains essential components such as an icon, battery status, and Wi-Fi indicator:

```
<PartTemplate                Name="STATUS_BAR"                Margin="1,1,1,1"
BackgroundColor="#00000000">
    <StackLayout>
        <StackItem Name="L2_BACKGROUND_HEADER" Orientation="Horizontal">
            <Panel Name="UBIMAX_SPACE" Weight="0.9"/>
            <Image              Name="UBIMAX_ICON"              Weight="0.1"
Content="ANDRRES_ubimax_logo"/>
        </StackItem>
        <StackItem     Name="L3_TEXT_OVERLAY"     Orientation="Horizontal"
Padding="3,0,3,5">
            <BatteryStatus Weight="0.05" Padding="1,1,1,1"/>
            <WiFiStatus Weight="0.05" Padding="1,1,1,1"/>
            <Text   Name="StatusBarInfo"   Weight="0.5"   Gravity="Center"
TextColor="gray" Font="Roboto-Light"/>
        </StackItem>
    </StackLayout>
</PartTemplate>
```

Code Block 1. Example of a PartTemplate [10].

Layout pages define the spatial arrangement of UI components. They act as abstract templates, providing placeholders that are filled later with content by the layout models [10]. This separation of structure and content ensures flexibility and reusability. This structure ensures a clear separation between the visual layout and the content that fills it:

```
<LayoutPage Name="DefaultMaster" Padding="0,20,0,20">
    <Part Template="STATUS_BAR" Weight="0.1"/>
    <Panel Weight="0.8">
        <ContentPlaceHolder Name="Content"/>
    </Panel>
    <Panel Weight="0.1">
        <ContentPlaceHolder Name="Footer"/>
    </Panel>
</LayoutPage>
```

Code Block 2. Example of a *LayoutPage* [10].

Layout models are implementations of ***layout pages***. They act as the concrete realization of the abstract templates defined in layout pages [10]. In layout models, placeholders are replaced with specific content, such as buttons, text fields, or images. An example of a layout model filling placeholders could look like following:

```
<LayoutModel Name="PaginatedText" Page=" DefaultMaster">
     <Content PlaceHolder="Content">
         <Text Name="MainContent" Text="Welcome" Style="HeaderStyle"/>
     </Content>
     <Content PlaceHolder="Footer">
         <Button Name="NextButton" Text="Next" Style="PrimaryButtonStyle"/>
     </Content>
</LayoutModel>
```

Code Block 3. Example of a *LayoutModel*.

This distinction between layout pages and layout models ensures that structural designs (layout pages) can be reused for multiple screens, with each screen having its own unique content.

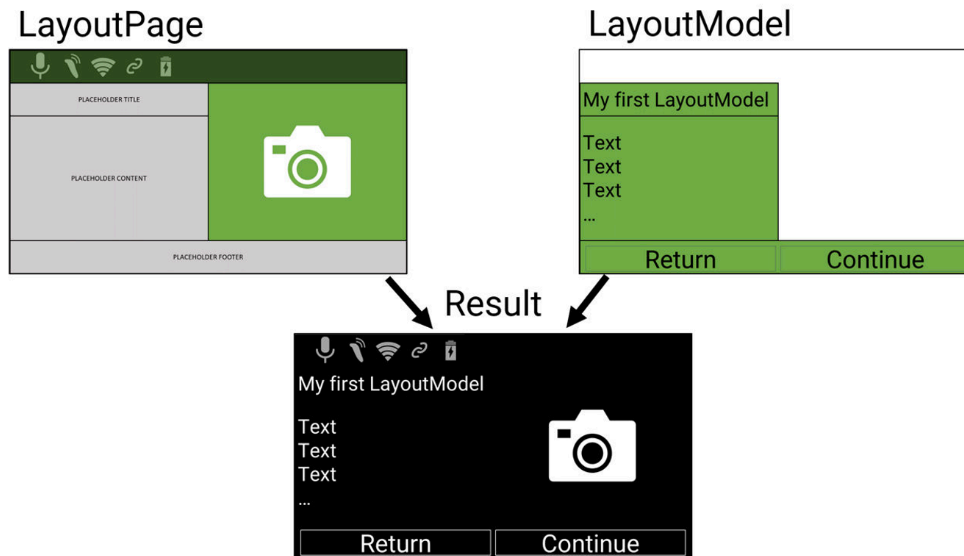


Figure 4. Combining *LayoutPage* and *LayoutModel* [10].

Styles allow developers to predefine visual properties such as text color, padding, borders, and alignment [10]. Styles are applied to components using the *style* attribute, enabling a uniform appearance across the workflow. For instance, a predefined style could include attributes for text alignment and padding:

```
<Style Name="InfoContentStyle" MaxLines="1" Gravity="Center"
Padding="0,5,0,5" Border="2,gray"/>
```

Code Block 4. Example of a *Style* [10].

When applied to a text element, this style ensures it adheres to the predefined visual guidelines:

```
<Text Name="Information" Text="Details" Style="InfoContentStyle"/>
```

Code Block 5. Text component using the *InfoContentStyle* defined earlier.

3.2 Cross-platform Development

The rise of diverse devices and operating systems has reshaped software development, prompting a shift from native to cross-platform approaches [11]. Native development, while offering optimal performance and platform-specific experiences, required separate codebases for each platform, leading to high development costs and increased complexity. This inefficiency, coupled with the growing need to deliver consistent experiences across devices, highlighted the limitations of maintaining siloed development workflows.

Cross-platform frameworks such as Xamarin and PhoneGap emerged to address these challenges by enabling developers to create applications for multiple platforms using a single codebase [12]. While these early solutions were limited in performance and user experience, they laid the foundation for modern frameworks like Flutter, React Native and the most recent Kotlin Multiplatform. These newer tools provide near-native performance, robust libraries, and greater flexibility, streamlining development, reducing costs, and accelerating time-to-market. This transition from native to cross-platform development reflects the industry's need for scalable, efficient solutions that balance performance with accessibility [11].

3.2.1 Benefits and Challenges

Cross-platform development has become essential due to the proliferation of devices and operating systems that users interact with daily. However, users expect a seamless, consistent experience whether they access an application on a mobile phone, tablet, or desktop, which is not always easy to achieve. This demand has pushed engineers to adopt frameworks and techniques that allow a single codebase to serve multiple platforms, reducing development time and cost.

Cross-platform mobile development frameworks have become popular in software development for their ability to streamline processes and cut costs by enabling code reuse across platforms like Android and iOS. This approach contrasts with native development, where separate codebases are typically maintained for each platform, leading to higher development costs and more complex update cycles. Cross-platform frameworks simplify maintenance and provide compatibility across multiple operating systems [13].

Moreover, market research emphasizes the importance to consider platform-specific conventions to ensure a positive user experience, as differences in operating systems affect user expectations [14]. Although cross-platform frameworks can introduce performance overhead due to bridging mechanisms, research by Biørn-Hansen et al. indicates that performance differences between native and cross-platform applications are often minimal, making cross-platform solutions viable for many use cases [15].

Despite these advantages, achieving UI/UX consistency across various platforms remains a critical concern for developers. Variations in the look, feel, and interaction behaviors across different operating systems can lead to user experience discrepancies, complicating the development [16], [17].

3.2.2 Balancing Standards, Performance, and Maintainability

Cross-platform development presents a multitude of challenges that software engineers must navigate to ensure high-quality user interfaces and user experiences (UX) [11]. These challenges are often exacerbated by the inherent differences between platforms, which can influence the design and functionality of applications:

- **Device Fragmentation and Platform-Specific Standards**: One of the primary challenges in cross-platform development is device fragmentation, which necessitates that applications operate seamlessly across a diverse array of screen sizes, hardware capabilities, and operating systems. Wu et al. discuss the complexities of developing applications for multiple platforms and emphasize the need for adaptable UI layouts to address these challenges [18]. Furthermore, adhering to established platform-specific standards, such as Material Design for Android and Apple's Human Interface Guidelines for iOS, is crucial for fostering user familiarity and comfort across different platforms [19].
- **Performance Optimization**: Performance optimization is another critical concern in cross-platform applications, which often fall behind native applications in terms of speed and responsiveness. Biørn-Hansen et al. identify that performance issues frequently stem from the additional abstraction layers required to support multiple platforms, leading to increased latency [15]. To counteract these performance drawbacks, engineers implement various optimization techniques, including lazy loading, efficient data handling, and minimizing unnecessary re-renders, which are essential for achieving performance levels that approach those of native applications.
- **Code Maintainability and Scalability**: Using a single codebase for multiple platforms can simplify initial development but often leads to complex, hard-to-maintain code. Biørn-Hansen et al. note that while cross-platform solutions streamline processes, they increase maintenance challenges as applications grow [15]. To address this, engineers should use modularization and separate platform-specific code, which makes future updates easier. This approach is supported by findings from Albesher, who notes that the cross-platform development landscape is continually evolving, necessitating ongoing adaptation and refinement of codebases [20].

3.2.3 Frameworks and Tools

Cross-platform frameworks have evolved to enable engineers to deliver consistent UI/UX across multiple operating systems from a single codebase. Selecting an appropriate framework is essential for achieving high performance, maintainability, and scalability. In addition to Flutter, React Native and Kotlin Multiplatform (KMP) has emerged as a promising choice, especially for developers aiming to maintain native performance with flexible platform-specific code.

1. **Flutter**: A prominent open-source framework by Google and known for its Dart-based widget architecture, Flutter compiles directly to native code, resulting in near-native performance. Similarly to Kotlin Multiplatform, one of the key advantages of this solution is its ability to utilize a single codebase for both iOS and Android applications [21]. However, it requires a dedicated Flutter UI for each platform, making it less ideal for projects needing deep integration with custom or native UIs.

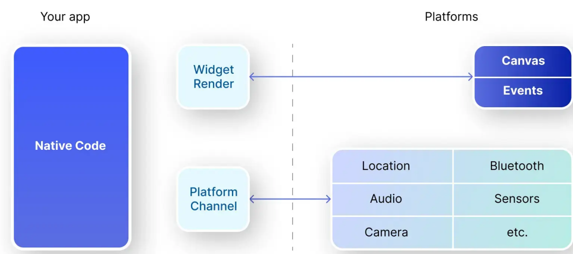


Figure 5. Flutter's architecture overview [22]

2. **React Native**: Developed by Meta Platforms (formerly Facebook Inc.), React Native leverages JavaScript to allow developers to write once and adapt minimally for native rendering on each platform [12]. While JavaScript's wide usage makes React Native highly accessible, certain features still require native bridging, which can sometimes lead to performance bottlenecks.

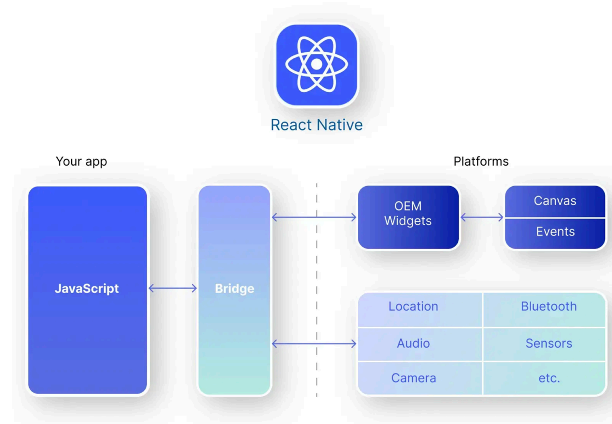


Figure 6. JavaScript's architecture overview [22]

3. **Kotlin Multiplatfor**: Developed by JetBrains, allows developers to share business logic across multiple platforms while still enabling the use of platform-specific user interface components. This hybrid approach distinguishes KMP from fully cross-platform solutions by maintaining native user interfaces for each platform, thereby minimizing code duplication for business logic. Additionally, KMP also supports shared UI components across platforms, offering a balance between consistency and customization. Recent research highlights KMP's unique advantage in its interoperability with native code, facilitating seamless integration with existing iOS and Android projects without compromising the performance or flexibility typically associated with native applications [23].

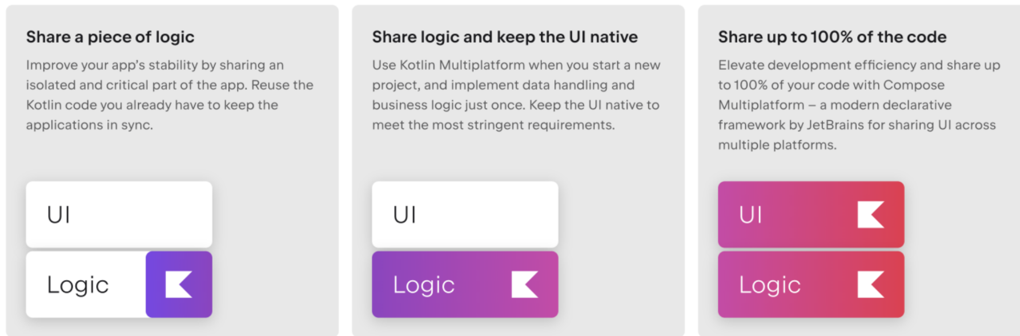


Figure 7. Kotlin Multiplatform flexibility in project architecture [24]

3.2.3.1 Framework Comparison

When selecting a cross-platform framework, software engineers must consider several critical criteria, including performance, flexibility in code sharing, UI consistency, and integration with existing codebases. Recent research highlights these metrics and provides guidance for engineers in choosing the most suitable framework for their specific project requirements.

3.2.3.1.1 Performance

Performance is a crucial factor in the selection of a cross-platform framework. React Native employs a bridge between JavaScript and native code, which can introduce latency, particularly in applications requiring high-performance graphics or complex animations. Studies indicate that React Native applications may experience longer load times and higher memory usage compared to their Flutter counterparts, which utilizes a compiled approach that directly interacts with native components, resulting in faster performance metrics [25], [26]. For instance, a comparative analysis found that Flutter applications exhibited a load time of 1.69 seconds, significantly lower than React Native's 4.26 seconds [25].

Kotlin Multiplatform, on the other hand, allows developers to write platform-specific code while sharing business logic across platforms. This hybrid approach can lead to performance that is closer to native applications, especially for computationally intensive tasks, as it leverages the strengths of the Kotlin language and its interoperability with existing Java codebases [27], [28]. However, the performance can vary depending on how much native code is utilized versus shared code, which necessitates careful architectural planning [15].

Table 2. Comparison of Cross-Platform Frameworks in Terms of Performance.

	KPM	Flutter	React Native
Performance	Near-native performance with Kotlin code compiled to native for each platform.	Compiles to native ARM code, reducing latency and avoiding intermediary layers.	JavaScript bridge may introduce performance overhead but supports native modules for optimization.

	KPM	Flutter	React Native
Resource Usage	Efficiently uses system resources, with shared logic reducing redundancy.	Optimized for graphical applications, ensuring minimal overhead during UI rendering and animation.	May consume more memory due to the additional layer of the JavaScript bridge.
Responsiveness	Highly responsive, particularly in computationally heavy tasks, depending on native-to-shared code ratio.	Maintains responsiveness in animations and transitions with consistent 60 FPS performance.	Responsiveness depends on native modules, with additional tuning needed for demanding applications.

3.2.3.1.2 Code Sharing and Development Flexibility

Code sharing is a significant advantage of cross-platform frameworks, allowing developers to maintain a single codebase for multiple platforms. React Native facilitates this by enabling developers to write components in JavaScript, which can be shared across iOS and Android. However, the reliance on native modules for certain functionalities can limit flexibility, as developers may need to write platform-specific code when performance or native features are critical [29].

Flutter stands out in terms of development flexibility, as it provides a rich set of pre-built widgets and a comprehensive toolkit that allows for rapid development and customization. Its single codebase approach simplifies the development process, enabling developers to create visually consistent applications across platforms without the need for extensive platform-specific adjustments [26], [30].

Kotlin Multiplatform offers a unique blend of flexibility and code sharing, allowing developers to share business logic while still enabling both shared UI components across platforms and native UI customization for each platform. This approach can lead to more maintainable codebases, as developers can leverage existing native UI components while ensuring that core functionalities remain consistent across platforms [27], [28]. Additionally, Kotlin Multiplatform supports shared UI components across platforms, allowing for consistency in user interfaces while also accommodating platform-specific customization when necessary. However, this may also require a deeper understanding of both Kotlin and the native development environments, which can be a barrier for some teams.

Table 3. Use Cases and Development Architecture of Frameworks.

	KPM	Flutter	React Native
Use Cases	Suitable for high-performance tasks like data processing, encryption, and image manipulation.	Best for interactive media, complex animations, and applications with rich visual elements.	Effective for standard applications, rapid prototypes, and projects needing quick third-party integrations.
Architecture	Shares business logic across platforms and supports both shared UI and native UI.	Offers a unified architecture with its own rendering engine, reducing reliance on platform APIs.	Employs a JavaScript bridge for cross-platform compatibility, enabling incremental or full-scale adoption.
Development Complexity	Balances code reuse and platform-specific design but requires knowledge of native frameworks.	Simplifies cross-platform UI development but demands understanding of Flutter's custom widget system.	Easier for web developers but may require native coding for advanced functionality and optimizations.

3.2.3.1.3 UI Consistency and Platform-Specific Adaptation

UI consistency is critical for user experience, and each framework approaches this differently. React Native allows for a degree of customization but often requires additional effort to ensure that applications look and feel native on both iOS and Android. This can lead to inconsistencies if not managed carefully, as developers must account for the different design philosophies of each platform [26], [29].

Flutter excels in UI consistency due to its widget-based architecture, which allows developers to create a uniform look across platforms. The framework's rich set of customizable widgets ensures that applications can maintain a consistent aesthetic while also adapting to platform-specific design guidelines when necessary [25], [30]. However, this can sometimes lead to challenges in achieving true native look and feel, particularly for complex UI elements.

Kotlin Multiplatform allows for the use of native UI components, which can enhance the user experience by providing a more authentic look and feel on each platform. This approach can

lead to better user satisfaction but may require additional development effort to manage the differences in UI design between platforms [27], [28].

Table 4. UI Consistency and Platform-Specific Adaptation in Frameworks

	KMP	Flutter	React Native
Features	Uses platform-specific UI components to create experiences tailored to each operating system.	Provides a robust library of widgets and uses high-performance rendering for consistent interfaces.	Combines JavaScript components with native modules for a balanced user experience.
Pros	Enables designs align with native patterns, leading to a more intuitive user experience.	Simplifies development of intricate animations and visually rich designs while maintaining cross-platform parity.	Supports rapid UI updates and adaptability, leveraging its integration with native components.
Cons	May require managing separate UI code for each platform, adding complexity to development workflows.	Demands familiarity with Flutter's widget-based system, which may require a learning period for newcomers.	Customizing advanced animations or transitions can require extra development effort and configuration.

3.2.3.2 Emerging Trends and Future Directions

Trends in cross-platform frameworks are increasingly focused on leveraging Generative AI to enhance user interfaces across devices, creating more adaptive and seamless experiences. As users expect consistent interactions regardless of platform, Generative AI enables UIs to personalize and dynamically adjust in real time, catering to individual needs and preferences. This adaptability is key to developing UIs that are accessible on any device, from desktops to mobile phones, ensuring that interactions remain intuitive and contextually relevant. Lightweight frameworks are crucial in this space, as they allow these advanced AI functionalities to perform efficiently on hardware-limited devices by balancing cloud-based and on-device processing.

According to J. Bieniek, M. Rahouti et al. the evolution of user interfaces is moving from single modal to more adaptive, multimodal systems that accommodate a diverse range of inputs, creating seamless experiences across platforms [31]. This shift addresses what the authors call the "interface dilemma," where designers must balance various interaction methods to create

effective multimodal experiences. These innovations highlight the need for scalable, lightweight frameworks, especially on mobile platforms, to support these dynamic interactions while addressing challenges around privacy, context retention, and resource management. Generative AI's role in enhancing multimodal capabilities underscores its potential to redefine cross-platform user interfaces, making them more intuitive, responsive, and user-centric across digital devices [31].

Looking forward, cross-platform frameworks will likely emphasize emotionally adaptive and predictive AI-driven UIs that respond to user behaviors and real-time cues. This evolution, however, brings ethical considerations such as privacy, data security, and computational efficiency, which are essential for building and maintaining user trust in AI-powered systems. As Generative AI continues to advance, it is positioned to redefine UIs, setting new standards for adaptive, intelligent, and responsible technology across digital platforms [31].

3.2.3.3 Technology Evaluation and Project Considerations

When selecting a cross-platform framework for a software project, engineers must evaluate various criteria, including performance, maintainability, scalability, and alignment with the organization's expertise and ongoing initiatives. This section explores the comparative strengths of KMP Flutter, and React Native, and provides a rationale for the final decision to proceed with KMP for this project.

3.2.3.3.1 Evaluation of Frameworks

Each framework offers distinct advantages, which makes the choice largely dependent on the specific requirements and constraints of the project.

1. **Flutter**

   Flutter is ideal for projects requiring highly customizable, visually rich user interfaces with consistent cross-platform performance. Its powerful widget system and unified rendering engine simplify the development process, enabling developers to deliver visually engaging experiences. However, Flutter's custom rendering engine can introduce challenges for deep integration with existing native components or applications, which is a significant consideration for projects requiring compatibility with legacy systems.

2. **React Native**

   React Native excels in projects with rapid development needs and a focus on leveraging existing web development expertise. Its flexibility, facilitated by JavaScript and native modules, allows teams to iterate quickly. However, the performance overhead caused by the JavaScript bridge and its dependency on native modules can be limiting for applications with demanding performance requirements, such as those involving intensive graphics or real-time processing.

3. **Kotlin Multiplatform**

KMP strikes a balance between shared code and native customization, making it a versatile choice for projects requiring native performance and seamless integration with existing systems. Its ability to share business logic across platforms while enabling platform-specific UI components provides developers with the flexibility to create optimized experiences tailored to each platform. This hybrid approach ensures maintainability and scalability without compromising user experience. However, it requires a higher level of expertise in Kotlin and native development, which could pose a challenge for teams without prior experience in these technologies.

3.2.3.3.2 Project Ponderations

Based on the evaluation, Flutter is an excellent choice for projects prioritizing uniform UI/UX and rapid prototyping, while React Native is well-suited for quick adaptations and leveraging web development skills. However, Kotlin Multiplatform emerges as the most appropriate framework for this project due to its flexibility, performance, and compatibility with existing systems.

This aligns closely with the company's strategic objectives, as existing teams and projects are already leveraging KMP, fostering a cohesive ecosystem of shared expertise and resources. This established infrastructure not only streamlines onboarding processes but also expedites development timelines. Furthermore, KMP's seamless integration with existing codebases ensures operational continuity and mitigates potential disruptions during implementation.

3.3 Compose Multiplatform

Cross-platform development requires balancing shared and platform-specific code to deliver a consistent and cohesive user experience across platforms. For organizations like TeamViewer, which prioritize uniformity in user interfaces on Android and iOS, KMP provides powerful tools to streamline the development process. Among these tools, Compose Multiplatform emerges as a leading framework for implementing a Shared UI approach, making it a natural fit for projects emphasizing consistency.

3.3.1 Native UI vs Shared UI

As previously mentioned, some frameworks allow the developers to approach their project in two distinct approaches in cross-platform development:

- **Native UI**: Focuses on developing UI components specifically tailored to each platform. While this enables better adherence to platform design guidelines and optimized performance, it requires managing separate codebases, increasing complexity and development time [32].

- **Shared UI**: Involves creating a single codebase for user interface elements that work across multiple platforms. This approach ensures consistency and simplifies development and maintenance but may sacrifice some platform-specific optimizations and interactions [33]

Native UI in KMP empowers developers to design user interfaces tailored to the conventions and expectations of each platform. By closely adhering to platform-specific guidelines, such as Material Design principles for Android and Apple's Human Interface Guidelines for iOS, Native UI ensures applications deliver a familiar and optimized user experience. This approach enhances usability and fosters a sense of integration with the operating system.

A significant advantage of Native UI is its ability to optimize performance for applications with resource-intensive features. Whether it involves real-time interactivity, intricate animations, or computationally demanding tasks, Native UI leverages the underlying platform architecture to provide smoother transitions, faster response times, and a polished user experience. Additionally, Native UI offers unparalleled flexibility, enabling developers to incorporate unique platform capabilities such as gesture controls, native animations, and hardware integrations like biometric authentication and haptic feedback. This customization allows developers to create features that resonate deeply with users of each platform.

However, Native UI development presents certain challenges. One major drawback is the increased complexity of managing separate codebases for each platform's UI components. This results in longer development cycles and greater maintenance efforts, as updates, bug fixes, and testing must be carried out independently for each platform. Moreover, ensuring consistent design and functionality across platforms can be challenging, as minor discrepancies in visual elements or behavior may arise, potentially undermining brand coherence. Another limitation is the slower rollout of new features, as platform-specific development and testing extend the time required to synchronize updates across platforms.

On the other hand, shared UI in KMP enables developers to create a single codebase for user interfaces that can be compiled to run seamlessly across multiple platforms, including Android, iOS, and desktop. This approach is made possible by frameworks like Compose Multiplatform. Shared UI is particularly advantageous for projects where consistent design, efficient development, and simplified maintenance are key priorities.

One of the primary benefits of shared UI is extensive code reusability. By centralizing UI development in a unified codebase, developers can reduce redundancy, ensuring consistent functionality across platforms while significantly accelerating development cycles. This approach simplifies maintenance and allows for faster rollouts of updates and bug fixes. Furthermore, shared UI promotes consistency across platforms, which is crucial for maintaining brand identity and delivering a predictable user experience. This cohesion is especially beneficial for users who switch between devices in business or productivity contexts.

Another notable advantage is the ability to update features and resolve bugs simultaneously across all platforms. Changes made to the shared codebase are deployed uniformly, ensuring

rapid implementation and reducing the risk of inconsistencies. This centralized workflow enables teams to respond efficiently to user feedback, streamlining the overall development process.

Despite these benefits, Shared UI has its limitations. One significant challenge is the reduced ability to customize experiences for individual platforms. The abstraction inherent in this approach can make it difficult to replicate native interactions or adhere to platform-specific conventions, potentially diminishing the user experience. Performance is another consideration, as the generalized nature of shared UI may not meet the demands of resource-intensive applications, leading to slower performance and less responsiveness compared to native UI implementations. Shared UI is also heavily reliant on Jetpack Compose Multiplatform, making it dependent on the framework's functionality and stability. This dependency can propagate framework-related issues across all platforms and complicate the integration of advanced platform-specific features or libraries.

Table 5. Comparison overview of Shared and Native UI in Kotlin Multiplatform

	Native UI	**Shared UI**
Overview	Develops UI components specific to each platform, aligning with platform conventions and expectations.	Uses a single UI codebase for multiple platforms.
Advantages	**Adherence to Platform Guidelines**: Aligns with unique design conventions (e.g., Material Design for Android, Human Interface Guidelines for iOS).	**Code Reusability**: Simplifies development by reducing platform-specific code, enabling faster updates and maintenance.
	Enhanced Performance: Optimized for platform architecture, ideal for resource-intensive or animation-heavy applications.	**Consistency Across Platforms**: Ensures cohesive user experiences and branding across devices.
	Greater Flexibility: Allows use of platform-specific gestures, animations, and hardware features (e.g., haptics, biometrics).	**Simultaneous Updates**: Features and bug fixes can be rolled out across platforms from a single codebase.
Challenges	**Increased Complexity**: Requires managing separate UI codebases for each platform, increasing development and maintenance workload.	**Limited Customization**: Lacks platform-specific optimizations, potentially reducing the intuitiveness of UI interactions.

	Native UI	Shared UI
	Inconsistent Experiences: Minor discrepancies in UI and functionality can arise between platforms, impacting branding and user trust.	**Performance Constraints**: May struggle with complex animations or platform-specific optimizations for data-intensive applications.
	Slower Updates: Platform-specific development delays rollout of features compared to the shared UI approach.	**Dependency on Compose**: Relies on Jetpack Compose Multiplatform, making it susceptible to its limitations or bugs.
Best Use Cases	Apps requiring platform-optimized interactions, high performance, and advanced user engagement features.	Apps prioritizing design consistency and fast iteration cycles.
Target Audience	Projects aiming to deliver superior platform-specific user experiences with tailored interactions and optimized performance.	Projects focusing on efficiency, uniformity, and reduced maintenance for multi-platform projects.

3.3.2 Compose Multiplatform Overview

Compose Multiplatform is a declarative UI framework developed by JetBrains that allows developers to create a unified UI codebase that works seamlessly across platforms, including Android, iOS, and desktop. Building on Jetpack Compose's architecture, Compose Multiplatform leverages Kotlin Multiplatform's shared logic capabilities to streamline cross-platform development [34].

Compose Multiplatform operates on a set of core principles and features that make it a versatile framework for cross-platform development. Leveraging the declarative UI paradigm and Kotlin's robust language features, it provides developers with a unified way to build user interfaces for multiple platforms while maintaining native performance and appearance. The following key elements illustrate how Compose Multiplatform achieves this functionality [34]:

3.3.2.1 Declarative Syntax

Compose Multiplatform employs a declarative programming model where developers define the desired UI state, and the framework dynamically updates the UI when state changes occur. This approach simplifies UI logic and ensures consistency in rendering.

3.3.2.2 Kotlin Multiplatform Integration

Compose Multiplatform is deeply integrated with Kotlin Multiplatform, enabling developers to share business logic and UI components across platforms. This reduces duplication of effort and fosters code reusability, allowing developers to focus on platform-specific customizations only where necessary.

3.3.2.3 Native Rendering

Rather than relying on a generic rendering engine, Compose Multiplatform uses native UI components on each platform. This ensures that applications not only achieve high performance but also retain the native look and feel expected by users on each platform. For example:

- On Android, Compose leverages the Jetpack Compose toolkit.
- On iOS, it translates components to UIKit counterparts.
- On desktop platforms, it uses Skia for rendering.

3.3.2.4 Composable functions

The framework's foundation lies in composable functions, which are modular, reusable building blocks of UI. These functions allow developers to create dynamic and interactive UIs by combining smaller, self-contained components into larger structures, all while adhering to the principle of immutability.

3.3.2.5 Reactive State Management

Compose Multiplatform is inherently reactive, meaning that it responds dynamically to changes in state. When the underlying data changes, the UI updates automatically without requiring explicit intervention from the developer, reducing boilerplate code and improving reliability.

3.3.2.6 Tooling Support

Compose Multiplatform benefits from comprehensive tooling provided by JetBrains and IntelliJ IDEA. Features such as live previews, debugging, and integration with Kotlin Multiplatform projects make it easier for developers to build, test, and optimize their applications.

3.3.2.7 Platform-Specific Adaptations

While Compose Multiplatform emphasizes code sharing, it allows for platform-specific UI customizations. This flexibility ensures that developers can meet unique requirements or leverage advanced capabilities of a given platform without compromising the overall project structure.

3.3.2.8 Active Ecosystem and Community

Compose Multiplatform is backed by an active ecosystem, including JetBrains' regular updates and a growing community of contributors. This ensures continuous improvement, additional features, and up-to-date support for new platforms and technologies.

3.3.3 Comparison with other frameworks

To evaluate the suitability of Compose Multiplatform, it is essential to compare it against Native UI and Flutter. Jacob Ras recently compared the app size & startup performance of the same app in native (Compose/Swift UI), Flutter and Kotlin/Compose Multiplatform (KMP) on Android and iOS [35]:

3.3.3.1 App size

On Android, the app size for Native and KMP/Compose was identical at approximately 1.46 MB, highlighting that adding KMP does not inflate the Android app size. On iOS, the KMP/Compose app size was 24.8 MB due to the bundling of Skia, a rendering library required for iOS but not Android. Flutter's app size was slightly smaller on iOS at 17.9 MB. These findings reflect the additional overhead required for Skia in KMP/Compose on iOS.

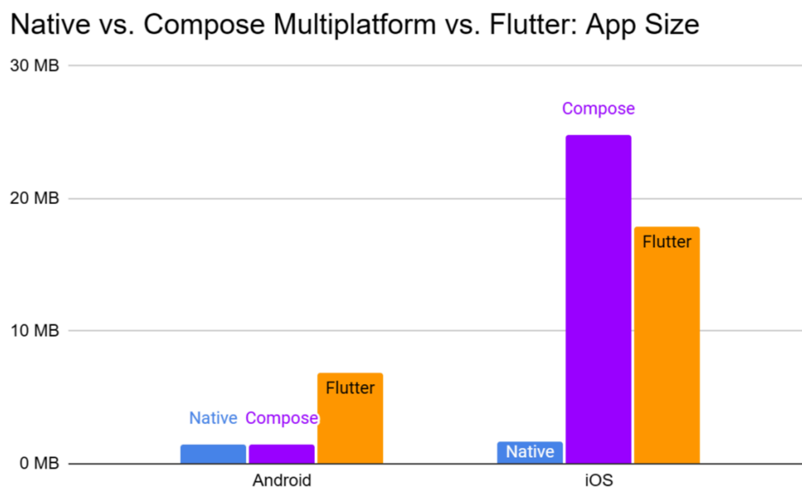


Figure 8. APK/IPA size in megabytes, lower = better [35]

3.3.3.2 Startup time

In startup time tests, Compose Multiplatform's performance was nearly identical to Native UI, with only minor differences observed on Android and iOS. In contrast, Flutter exhibited slower startup times, particularly on Android, due to the overhead of initializing the Flutter engine. This additional startup time is consistent with Flutter's architecture.

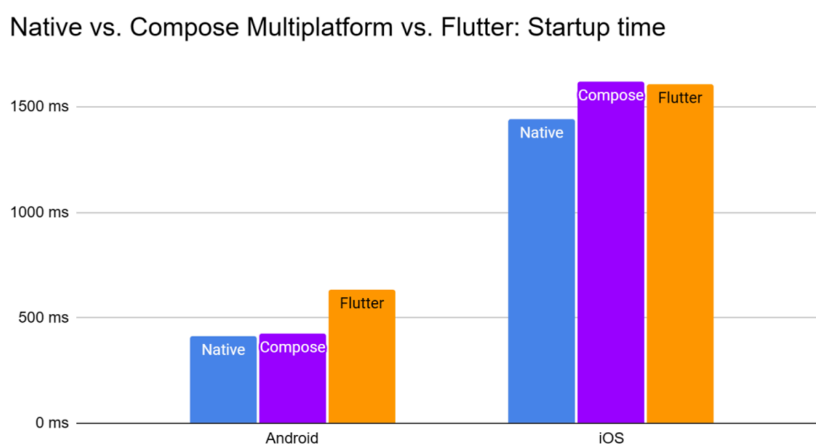


Figure 9. Startup time on a Pixel 4a and iPhone 12 Mini in milliseconds, lower = better [35]

3.3.3.3 Conclusion

Compose Multiplatform represents a middle ground between the flexibility of Native UI and the simplicity of Flutter. It is particularly well-suited for teams familiar with Kotlin or those already

invested in the JetBrains ecosystem. While some challenges, such as app size on iOS, remain, its continuous evolution suggests significant potential for cross-platform development [35].

Table 6. Comparison: Native UI, Compose Multiplatform, and Flutter

Aspect	**Native UI**	**Compose Multiplatform**	**Flutter**
Codebase	Separate for each platform	Shared for UI and logic	Shared for UI and logic
Performance	Optimal	Near-native, platform-dependent	High, but less native fidelity
Development Complexity	High	Moderate	Low
Platform-Specific Features	Excellent	Good	Limited
Tooling	Platform-specific	Kotlin and JetBrains ecosystem	Dart and Flutter ecosystem
Consistency	Challenging to maintain	Excellent	Excellent

3.3.4 Project Ponderations

Given the specific needs and constraints of this project, the decision to adopt a Shared UI approach using Compose Multiplatform appears to be the best fit. The project's primary goals are consistency across platforms, efficient development, and meeting tight deadlines, all of which are supported by the Shared UI approach. Compose Multiplatform, as part of the Kotlin Multiplatform ecosystem, offers the flexibility to create a unified UI codebase while allowing for platform-specific customizations where necessary.

Time constraints are a key factor influencing this decision. A Shared UI approach significantly accelerates the development cycle by centralizing UI code into a single codebase. This reduces redundancy and ensures that updates, bug fixes, and new features can be rolled out simultaneously across Android and iOS platforms. The ability to address these updates across both platforms at once will be invaluable in meeting the project's deadlines.

On the other hand, while Native UI could offer superior platform-specific performance and customization, it requires managing separate codebases for each platform. This would increase both development time and maintenance complexity, making it less suitable for this particular project given the time constraints. Additionally, the extra effort required to ensure consistency between the two platforms when using Native UI could delay the project's progress.

The Shared UI approach with Compose Multiplatform is the optimal choice. It meets the need for cross-platform consistency, accelerates development timelines, and leverages the existing knowledge of Kotlin Compose. While Native UI may be better suited for projects requiring heavy

platform-specific customizations, the time constraints of this project make Shared UI the most practical and effective solution.

3.4 Handling Platform-Specific Variations

With KMP established as the chosen technology for this project, its ability to facilitate both seamless code sharing and platform-specific adaptations makes it imperative to explore how such variations are effectively managed.

3.4.1 Platform Abstractions: *expect* and *actual*

Kotlin Multiplatform enables developers to share code across multiple platforms while also allowing platform-specific implementations where necessary. One of the most significant features in KMP is the *expect* and *actual* declarations. This declaration helps handle cases where certain functionalities are platform-specific, such as device APIs or hardware interactions. The abstraction allows developers to define platform-agnostic behavior in common code while providing platform-specific logic in separate platform modules [36]:

- ***expect***: This is defined in the common code and declares an interface or class that is expected to exist across platforms. It contains no actual implementation details but defines the properties or methods that are required.
- ***actual***: This is implemented in platform-specific code and provides the actual implementation for the expect class or interface. Each platform's actual implementation can differ depending on platform-specific capabilities, while the shared code remains the same.

Consider an app that needs to read a file from the device's storage. While the general logic for reading a file is the same across platforms (access the file, retrieve its content, and return it), the underlying APIs are different for Android and iOS. For this Kotlin's *expect-actual* declarations can be used to abstract this functionality.

First, we define an *expect* class in the common code. This class will serve as the interface for reading files, and both Android and iOS will provide their own *actual* implementations.

```
expect class FileReader {
    fun readFile(fileName: String): String
}
```

Code Block 6. Common *expect* class that defines the methods to be implemented on each platform

By doing this, the *expect* class defines the general structure that both platforms (Android and iOS) will follow. It declares a method for reading a file but leaves the actual implementation to the platform-specific code.

To enable KMP to substitute the expected implementation with the actual implementations on different platforms, those should be provided.

```
actual class FileReader(private val context: Context) {

    actual fun readFile(fileName: String): String {
        return try {
            // Access file from Android's internal storage
            val file = File(context.filesDir, fileName)
            val fileInputStream = FileInputStream(file)
            val data = fileInputStream.readBytes()
            fileInputStream.close()
            String(data)
        } catch (e: IOException) {
            e.printStackTrace()
            "Error reading file"
        }
    }
}
```

Code Block 7. Android-specific implementation of the *FileReader* class using Android file system APIs.

On Android, the *actual* class can use the Android file system APIs, such as *FileInputStream*, to read the file's content. On iOS, on the other hand, as *FileInputStream doesn't exist,* the *actual* class can use the *NSFileManager* and *NSURL* to access the file system.

```
actual class FileReader {

    actual fun readFile(fileName: String): String {
        val fileManager = NSFileManager.defaultManager
        val fileUrl = NSURL.fileURLWithPath("/path/to/directory/$fileName")

        return try {

            val fileData = fileManager.contentsAtPath(fileUrl.path)
            fileData?.let {
                it.toString(UTF8String)
            } ?: "File not found"
        } catch (e: Exception) {
            e.printStackTrace()
            "Error reading file”
        }
    }
}
```

Code Block 8. iOS-specific implementation of the *FileReader* class using iOS file system APIs.

Finally, the common code can call these methods without needing to know the platform-specific details. The correct *actual* implementation will be used based on whether the app is running on Android or iOS.

```
fun readFileContent(fileName: String): String {
    val fileReader = FileReader()
    return fileReader.readFile(fileName)
}
```

Code Block 9. Common code usage that accesses platform-specific implementations through the *expect* API.

3.5 Testing for Multi-Platform Consistency

Ensuring consistency across platforms is essential for maintaining both functionality and user experience in cross-platform applications. Automated testing plays a key role in verifying that an app performs seamlessly across Android and iOS. This chapter explores various testing frameworks and approaches, emphasizing the importance of a well-rounded testing strategy to guarantee consistency in KMP projects.

3.5.1 Automation Frameworks

Several tools and frameworks exist to facilitate automated testing for cross-platform mobile applications. Notable ones include Appium, Detox, and Espresso. These tools enable the testing of UI components on multiple platforms from a single codebase, allowing developers to run tests across both Android and iOS environments simultaneously.

3.5.1.1 UIAutomator

UIAutomator is a mobile testing framework developed by Google and included as part of the Android SDK [37]. It is particularly designed for Android-specific UI testing and supports functional UI tests on both system and installed applications. The framework includes a graphical interface called the UI Automator Viewer, which simplifies identifying UI components [38]. However, it supports only Java and Kotlin for test case creation, lacks support for web views, and is cumbersome when handling complex UI components such as lists. Additionally, its Android-specific nature limits its utility in cross-platform scenarios.

3.5.1.2 Espresso

Espresso, also developed by Google, is a white-box testing framework for Android applications [39]. This tool allows developers to create highly customizable UI tests and supports features like the Hamcrest library for matchers, as well as an Espresso Recorder for test generation without writing manual code [38]. Its exclusivity to Android further reduces its suitability for KMP projects that require cross-platform compatibility.

3.5.1.3 Appium

Appium is a widely adopted, open-source, cross-platform automation tool designed to test native, hybrid, and web applications on Android and iOS [40]. This tool supports multiple programming languages, including Kotlin, Java, and Python, making it highly flexible. Additionally, Appium enables tests on real devices, emulators, and simulators, offering

comprehensive functionality [38]. Despite these advantages, Appium can be challenging to set up initially, and its performance may be slower compared to platform-specific tools. Nonetheless, its ability to unify test processes across multiple platforms makes it an ideal choice for KMP projects.

3.5.2 Physical Devices vs Device Emulators

Testing on emulators or simulators and physical devices presents distinct advantages and challenges. This section discusses the advantages and challenges of both approaches, particularly within the context of KMP development, where cross-platform compatibility is crucial.

3.5.2.1 The Importance of Real-Device Testing

Real-device testing is crucial for ensuring the robustness and reliability of mobile applications. Testing on physical devices provides the ability to assess how apps interact with hardware in real-world scenarios, such as GPS accuracy or the touch sensitivity of the screen [41]. Real devices also give developers insights into how apps perform under real network conditions, varying environmental factors, and hardware limitations. These aspects are often overlooked in an emulator, which can only simulate but not reproduce actual device behavior accurately [42].

Additionally, real-device testing helps uncover issues that may not be apparent in emulators. These include device-specific bugs, performance issues like overheating, and subtle differences in hardware optimization [42]. While real-device testing can be more resource-intensive, it is essential for providing the most accurate feedback on the user experience, ensuring that the app meets the expectations and requirements of real users.

In KMP development, testing on real devices becomes even more critical, as the app needs to run smoothly on both Android and iOS platforms. Although KMP allows sharing code between these platforms, there are platform-specific features and behaviors that cannot be accurately simulated on an emulator. For example, interactions with native APIs, device-specific hardware optimizations, or the management of touch gestures across different devices need to be tested on real devices to ensure seamless user experiences across platforms [42].

3.5.2.2 Pros and Cons of Emulators

Despite the advantages of testing in physical devices, emulators are widely used in mobile application testing due to their cost-effectiveness. They allow for testing without requiring physical devices, which is especially helpful for smaller teams or those with limited budgets [42]. Additionally, emulators provide the ability to simulate multiple devices and OS versions, which can speed up initial testing and debugging [41]. This flexibility helps developers test apps in diverse virtual environments without the need for a variety of physical devices.

However, emulators do not replicate all aspects of real-world performance. Hardware-specific functionalities, such as battery behavior, camera quality, or GPS accuracy, can be simulated, but

not in the exact way real devices would perform [42]. The virtual environment of emulators often leads to slower performance compared to physical devices, which can affect the testing of apps, especially those with high graphical requirements [43]. Emulators also miss out on real-world conditions, like network fluctuations, which could affect how an app behaves in everyday use [41].

For cross-platform development, emulators are especially useful when testing across multiple platforms, such as Android and iOS, from a single codebase. Emulators allow developers to test app behavior on different device configurations, OS versions, and screen sizes, without needing to physically own each device [41]. This makes emulators an efficient tool for ensuring compatibility across a broad range of devices during the early stages of KMP app development.

3.5.2.3 Hybrid Approach

Balancing emulator and real-device testing can be a challenging aspect of cross-platform development. Emulators offer speed and efficiency in the early stages of development, where quick testing and bug-fixing are necessary [43]. However, relying solely on emulators risks missing device-specific issues and performance concerns that only real-device testing can reveal [42]. Given the variety of devices on the market, testing on every possible configuration would be impractical using only physical devices, making emulators a useful tool for broader coverage [43].

For optimal testing, a hybrid approach combining both emulators and physical devices is often employed, where emulators provide early-stage testing and real-device testing is used for final validation before release [42].

3.5.3 Performance Benchmarks

Performance benchmarks are a widely used approach to assess the efficiency and reliability of mobile applications across different platforms. They provide objective, measurable data that helps determine whether an application meets the performance standards expected in real-world scenarios [44][45]. By focusing on key indicators such as startup time, application size, CPU usage, and memory consumption, benchmarks allow developers to identify potential bottlenecks and optimize resource utilization:

- **Startup Time**: Evaluating the time from process creation to the first rendered frame helps determine the perceived responsiveness of the application, which is a major factor in user satisfaction [46].
- **Application Size**: Measuring the size of the application package validates its suitability for deployment in environments with strict storage and update constraints, such as Mobile Device Management (MDM) systems [44].
- **CPU Usage**: Monitoring CPU consumption during typical workflows ensures that the application operates efficiently without overloading the device, which is essential for maintaining performance and battery life [44].
- **Memory Usage**: Assessing memory consumption helps identify risks of instability or crashes, particularly on resource-constrained devices like smartglasses [47].

By analyzing these metrics across different platforms and devices, performance benchmarks provide a clear, objective basis for comparing implementations and identifying optimization opportunities [45][46].

3.5.4 Feedback Validation

Feedback validation is a critical approach to ensure that the application delivers a consistent and intuitive user experience across platforms. Unlike purely technical tests, this method incorporates real-world perspectives, allowing developers to identify usability issues that automated, or performance-based tests might overlook [48], [49].

- **Visual Consistency**: Confirming that the interface maintains a coherent look and feel across platforms.
- **Navigation Flows and Interaction Methods**: Evaluating the usability of touch, voice commands, and hardware buttons in different contexts.
- **Responsiveness and Perceived Performance**: Gathering subjective impressions of speed and smoothness, complementing objective performance data.

Responses are often collected using Likert-scale questionnaires [49], combined with open-ended feedback to capture qualitative insights. This approach ensures that validation covers both quantitative usability metrics and qualitative user experience factors, making it a powerful complement to technical performance testing [47], [48].

3.5.5 Project Ponderations

After evaluating the available testing strategies, this project adopts Performance Benchmarks (3.5.3) and Feedback Validation (3.5.4) as the primary methods for testing and validating the implemented solution. This decision is based on the dual need to:

- **Quantify technical improvements in performance**, ensuring that the solution meets the efficiency requirements of industrial environments.
- **Validate user experience through real-world feedback**, confirming that the adaptive UI system delivers consistency and usability across heterogeneous devices.

This combined approach ensures a comprehensive evaluation, addressing both objective performance metrics and subjective user satisfaction, which are equally critical for the success of cross-platform applications [44], [46].

Automation frameworks were considered but not selected due to their high setup complexity and limited added value for the project's research objectives, which focus on adaptability and performance rather than functional regression testing. Additionally, implementing automated tests for multiple environments would have been very time-consuming, reducing the time available to focus on development quality. By prioritizing benchmarks and real-user feedback, the project ensures practical, real-use testing that provides both quantitative and qualitative insights without the overhead of maintaining complex automation suites.

4 Design and Implementation

This chapter presents the design and implementation details of the application, which from now on will be addressed as Picking Client, as it is specifically designed for picking operations within the Frontline product. It outlines the architectural decisions, user interface design, and integration strategies that guided the development process. The goal is to provide a comprehensive view of how the application was structured to ensure scalability, maintainability, and a seamless user experience.

The chapter begins by defining the overall architecture, addressing its layers, integration with FrontlineBase, and the navigation model. It also addresses key aspects such as theming, translations, and asset management, which are essential for delivering a consistent and localized user interface.

Following the architectural overview, the chapter delves into the UI development process, detailing the theming approach, component design, and mechanisms for managing dialogs and notifications. The subsequent sections describe the application flow, covering authentication, workflow execution, and settings management.

Finally, the chapter explores advanced features such as workflow integration, where the application overrides and extends FrontlineBase's UI module, and interaction methods, including voice control and physical button navigation, to enhance usability in diverse operational environments.

4.1 Architecture Definition

This section describes the system architecture of the project, which is implemented using KMP with Jetpack Compose. The architecture is designed to maximize code reusability while ensuring adaptability to different platforms and device types.

The project follows a shared codebase architecture, where most of the logic, user interface components, and assets are implemented in the common module. Platform-specific code is used only for functionalities that require native access, such as permissions, file management, and camera handling**.**

The project also integrates with a third-party multiplatform package from Frontline named FrontlineBase, which provides core business logic, including user management, asset management, and workflow execution. The application primarily focuses on rendering the user interface and adapting it dynamically based on device characteristics.

4.1.1 Architectural Layers

The architecture of the Picking Client application is organized into three distinct layers, each with clearly defined responsibilities, and built using Kotlin Multiplatform to maximize code reuse across platforms. The design separates the user interface, business logic, and platform-specific functionalities, ensuring maintainability, scalability, and adaptability to multiple devices. A good visual overview of the client's architecture can be seen on Figure 10.

4.1.1.1 Presentation Layer

The presentation layer is responsible for rendering the user interface and managing user interactions. It is implemented in the picking-ui module using Jetpack Compose Multiplatform within the common module. The UI logic is shared across platforms but specialized through device-specific submodules. The Mobile module provides an interface tailored for smartphones with touch-based interaction, while the Smartglasses module supports alternative interaction methods specific to smartglasses.

The PickingUI component acts as a unified entry point, delegating rendering to the appropriate device-specific UI. Workflow steps are dynamically rendered by the WorkflowUIRenderer, which interprets workflow definitions and communicates with PickingUI. Importantly, this layer does not contain any business logic; instead, it relies on the underlying backend services to handle data and workflow execution, maintaining a clean separation of concerns.

4.1.1.2 Business Logic Layer

All business logic is provided by the FrontlineBase KMP library, which serves as the core processing engine for the application. This layer manages workflow execution, asset management, user authentication, and role-based access control.

Key components include the WorkflowEngine, which drives workflow progression, and the WorkflowManager, which coordinates operations between workflows and data providers. Data services such as UserInfoProvider, AssetManager, and WorkflowDataRepository handle persistent storage and asset retrieval. By delegating these responsibilities to FrontlineBase, the Picking Client itself remains a thin client that focuses on presenting workflows and capturing user interactions.

4.1.1.3 Platform-Specific Layer

The platform-specific layer provides access to functionalities that cannot be abstracted in shared code, such as camera operations, file handling, and permissions management. These features are implemented using Kotlin Multiplatform's *expect/actual* mechanism, which allows each platform to define its specific behavior while presenting a unified interface in the shared module. This approach ensures that device-dependent operations remain encapsulated, enabling the application to run seamlessly on multiple platforms.

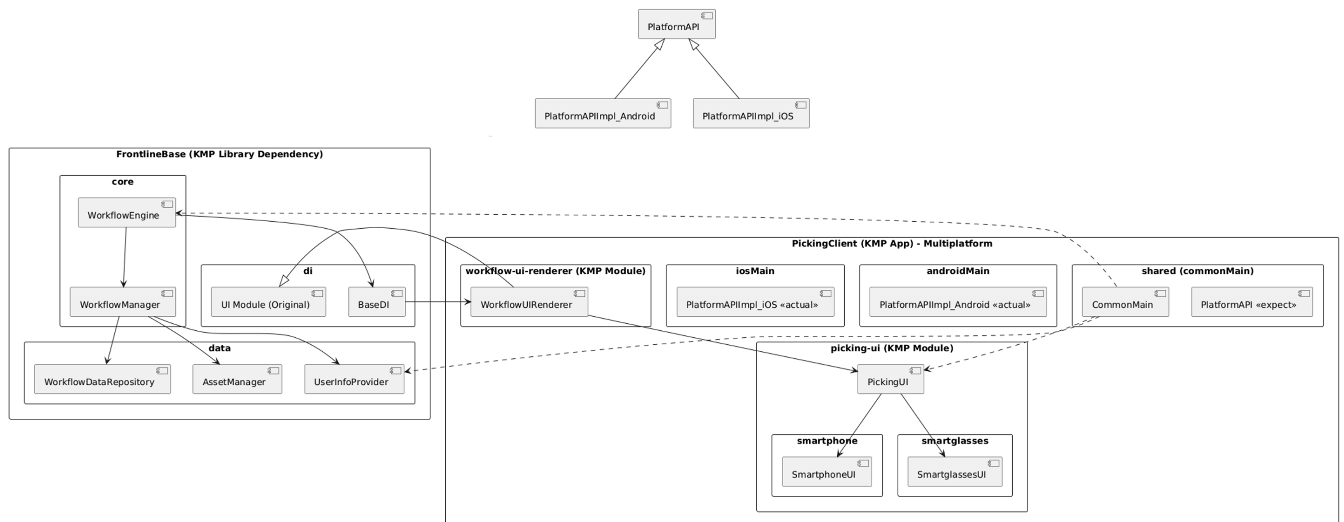


Figure 10. Picking Client Architecture

4.1.2 Integration with FrontlineBase

The project integrates with FrontlineBase, a multiplatform dependency that provides the essential backend functionalities, including:

- **User Management**: Handles authentication and role-based access control.
- **Asset Management**: Manages workflow-related assets and resources.
- **Workflow Catalog**: Stores workflow defined in Creator.
- **Workflow Engine**: Executes workflows dynamically based on user interactions.

The application does not implement custom business logic but instead interacts with the Frontline package, which acts as the core processing unit and enables:

- Separation of concerns, ensuring the app remains focused on UI rendering.
- High modularity, as new workflow features can be introduced without modifying the UI logic.
- Consistency across platforms, since workflow execution is handled uniformly.

The interaction between the application and the Frontline package occurs through shared APIs, making it independent of platform-specific implementations.

4.1.3 User Interface and Theming

The user interface is implemented entirely within the common module, ensuring a consistent and unified experience across all supported platforms. This centralized approach simplifies maintenance and guarantees that visual and interactive elements behave uniformly, regardless of the device.

A custom theming system has been developed to support adaptability across various screen sizes and interaction models. The theme is composed of a centralized color palette, defined in a shared theme file, which promotes visual consistency; and custom dimensions that adjust dynamically based on the device type, ensuring responsive layouts.

To accommodate different device categories, namely smartphones and smartglasses, the application uses device-specific UI adaptation. This is achieved through the DeviceUtils utility, which leverages the Kotlin Multiplatform *expect/actual* pattern to determine device characteristics at runtime. Devices with touch input are identified as smartphones, whereas devices without touch input are treated as smart glasses.

Beyond device type detection, DeviceUtils provides comprehensive device information, including manufacturer and model details, which are useful for debugging and optimization. It also exposes screen characteristics such as aspect ratio and screen size, which are particularly important for supporting non-standard formats.

These parameters are used to adjust both layout and behavior dynamically. The utility supports conditional logic for device-specific and manufacturer-specific behavior, enabling the application to tailor the user interface and functionality to the characteristics of each device type. For instance, aspect ratio data ensures that the UI remains visually coherent on unconventional screens, while manufacturer information can guide optimizations for specific devices.

Integration with Jetpack Compose is facilitated through the *rememberDeviceUtils()* composable function [50], which provides a snapshot of the device state, allowing real-time, reactive updates to the UI. By combining device type detection, screen metrics, and manufacturer-specific logic, DeviceUtils allows the application to maintain a consistent, accessible, and responsive user experience across a wide variety of device form factors.

4.1.4 Navigation Architecture

Navigation is managed centrally within the common module, reinforcing the application's cross-platform consistency and reducing platform-specific dependencies. The navigation system is anchored by the *App()* function, which serves as the top-level UI component.

Within this function, a *NavHost* [51] is responsible for managing transitions between screens. This structure enables:

- Centralized control of navigation logic, simplifying the addition of new screens.
- Dynamic screen transitions based on the current state of the workflow engine.

By handling navigation in a shared module, the application ensures a seamless user experience across platforms. This approach also enhances modularity and reduces code duplication, as navigation behavior is defined once and reused consistently.

4.1.5 Translations and Asset Management

To support future internationalization and maintain a cohesive visual identity, the application adopts a centralized strategy for both localization and asset management. All relevant resources are stored in the common module, ensuring uniform behavior across platforms.

The translations system is designed for scalability:

- All translatable strings are stored in a centralized repository.
- New languages can be added without modifying platform-specific code.

Similarly, asset management is handled in a unified manner:

- Images, icons, and other visual assets are stored in the shared module.
- Platform-specific code is only responsible for loading and rendering these assets, not for maintaining separate versions.

This centralized approach improves maintainability and ensures that updates to translations or assets are applied consistently across all supported devices.

4.2 UI Development

The UI of this project is entirely developed in Compose Multiplatform [52], ensuring a shared codebase across different platforms while allowing for device-specific adaptations. By keeping UI implementation within the common module, the project maintains consistency across

platforms while handling permissions, file management, and hardware interactions separately in platform-specific code.

The core UI structure follows a scaffold-based layout [53], with three primary sections:

- **Header:** Displays essential navigation elements and contextual information.
- **Main Content:** The primary interactive area, adapting to different screen sizes and input methods.
- **Footer:** Provides navigation actions and secondary controls, ensuring easy access across devices.

This structure ensures a uniform user experience, while allowing dynamic adaptation based on the device type, such as smartphones, smartglasses, and other wearable displays.

4.2.1 Theming and Color Adaptation

An initial application design draft adopted a blue-centric theme aligned with TeamViewer's branding. However, an analysis of OLED technology revealed several challenges. Bright, saturated colors, especially blue, consume more power and accelerate pixel degradation, leading to image retention and burn-in [54]. Blue subpixels age faster than others, making them the main contributor to these issues. OLED can display true black by turning pixels off, reducing power consumption significantly. However, high-contrast combinations like bright blue or white on black can cause eye strain in near-eye devices. To improve durability and comfort, muted tones and warmer hues are recommended, as they reduce stress on blue pixels and distribute energy demand more evenly.

To address these findings, a device-specific theming strategy was implemented. On **s**martphones, the original blue-based theme was retained for brand consistency, as energy and burn-in concerns are less critical. On smartglasses, a yellow-dominant theme replaced blue, combined with absolute black backgrounds to maximize energy savings and contrast. Yellow provides strong visibility without the degradation issues of blue, while black backgrounds leverage OLED's ability to turn off pixels entirely. Additionally, muted tones and warm accents were introduced to reduce eye strain and improve long-term usability. This approach ensures the interface is optimized for power efficiency, durability, and user comfort, while maintaining a visual link to the brand identity.

4.2.2 UI Components

A key focus of this project is the development of reusable UI components that uphold a consistent design system while supporting a wide range of interaction paradigms and screen form factors. Each component has been tailored for usability across both touchscreen smartphones and smartglasses, with specific adaptations made to accommodate each platform's unique capabilities and constraints.

4.2.2.1 Buttons

The system implements three types of buttons: primary, outlined, and text buttons. All are designed to be scalable and adaptable across devices while maintaining visual consistency and accessibility.

Across both smartphones and smartglasses, buttons provide clear visual feedback for focus and press states (Figure 11). These states are managed using an *InteractionSource* [55], which captures user interactions and applies dynamic styling such as container and outline colors based on theme-aware values.

On smartglasses, an additional focus indicator is implemented through a dual-outline design. This approach enhances visibility and interaction clarity without altering the color palette A custom modifier [56] was created for this purpose. In Jetpack Compose, a modifier is a declarative construct used to alter the behavior or appearance of a composable UI element. The custom modifier dynamically renders a second inner outline within the button when it is in focus, improving accessibility and interaction clarity without altering the color palette.

Figure 11. Outlined, primary and text buttons, respectively, for smartglasses (top) and smartphones (bottom)

4.2.2.2 Menu Cards

Menu cards (Figure 12) provide visually rich selection elements and are primarily used on smartphones. Each card supports focus and press feedback, using custom styling via *InteractionSource* [55] and dynamic theme-aware colors.

A custom animation system enables horizontal pagination of cards, with dynamic scaling to visually emphasize the currently selected item. This is achieved by calculating the focused state relative to the central card and applying a scale transformation accordingly. On smartglasses, menu cards are rendered at a fixed size without pagination or animation, ensuring stability and clarity in an AR context.

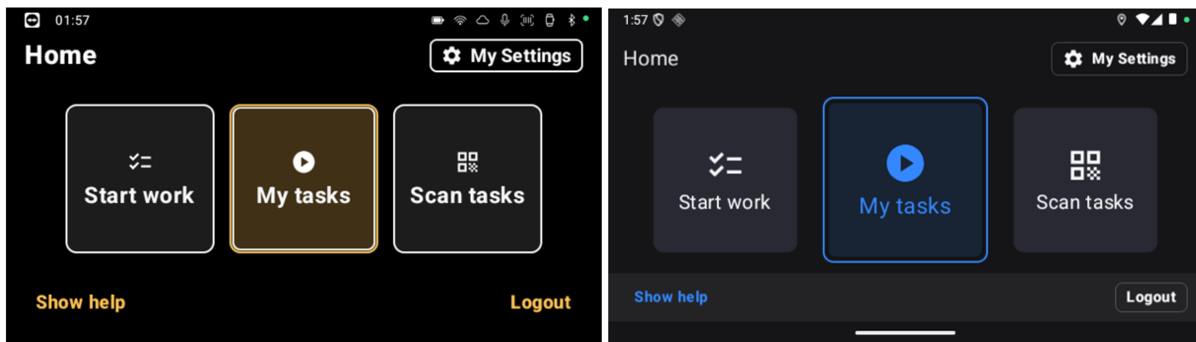


Figure 12. Home Screen on smartglasses (on the left) and smartphones (on the right)

4.2.2.3 List and Pagination System

The Pagination component was designed to manage the presentation of large data sets in a way that supports efficient navigation and content discovery. Its implementation varies across platforms to align with their interaction models and constraints.

The List and Pagination components were designed to present structured data efficiently and enable smooth navigation through large datasets. Their implementation varies across platforms to accommodate different interaction models and device constraints.

On smartphones, list items are optimized for touch interaction, arranged vertically for easy scrolling. Infinite scrolling is employed to allow users to browse content seamlessly without explicit navigation actions.

On smartglasses, list items are adapted for focus-based interaction or voice commands, featuring larger spacing and clear feedback cues to enhance usability in augmented reality environments. Instead of infinite scrolling, a paging system is used, displaying a fixed number of items, typically three per page. Navigation is performed through voice or hardware inputs, supported by visual indicators that denote available pages.

An example of this implementation can be seen in Figure 20.

4.2.2.4 Header and Footer

The header and footer (Figure 12) serve as essential layout anchors, providing contextual information and housing key interactions. Their design adapts to the constraints and interaction models of each platform to ensure clarity and usability.

On smartphones, the header typically includes titles and navigation icons, while the footer aligns actions horizontally, adjusting padding based on screen ratio for ergonomic interaction.

On smartglasses, headers adopt a minimalist approach with space-efficient typographic elements, and footers emphasize interaction clarity through larger tap targets or focus outlines to support hands-free or limited-input scenarios.

4.2.2.5 Dialogs

The Dialog component (Figure 13) was designed to support short, focused interactions that require user acknowledgment or decision-making, such as confirmations, alerts, or transient notifications. Its primary role is to ensure critical information is communicated clearly without introducing unnecessary cognitive load. Although the functional purpose of dialogs remains consistent across platforms, their visual and behavioral characteristics are adapted to the interaction paradigms and constraints of each device.

On smartphones, dialogs appear as modal overlays centered on the screen, following established mobile design conventions. They include clear action buttons and concise messaging to facilitate quick comprehension and response.

On smartglasses, given the limited display area and the need to preserve situational awareness, dialogs adopt a compact and unobtrusive layout. Font sizes and positioning are optimized for legibility while minimizing obstruction of core content.

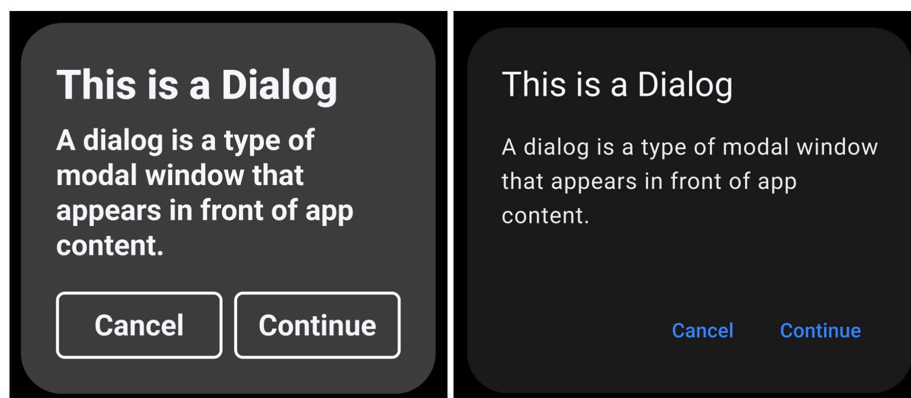


Figure 13. Dialog design for smartglasses (on the left) and smartphones (on the right)

4.2.2.6 Side Drawer

A side drawer (Figure 14) is exclusively used on smartglasses to provide an accessible menu for context switching. While this component is referred to as a side drawer, its behavior and interaction model more closely resemble that of a side sheet. However, the official side sheet component is not yet available in Compose [57], requiring a custom implementation using the *Navigation Drawer* [58] that replicates its functional characteristics while maintaining compatibility with the Compose UI framework.

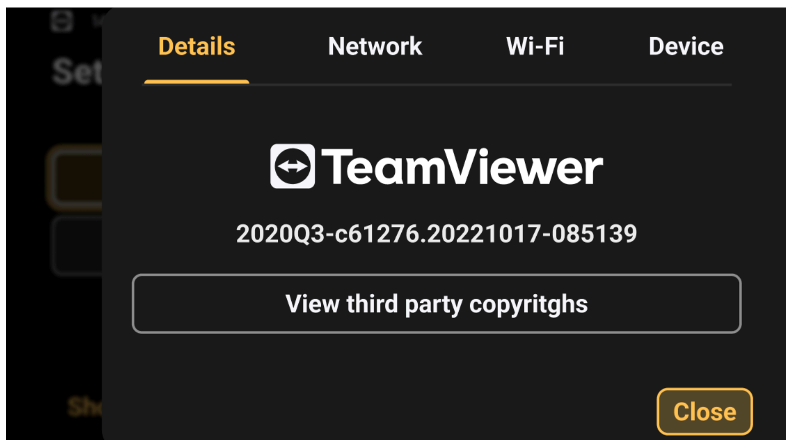


Figure 14. Side drawer running on an Android Device

4.2.2.7 Notifications

Notifications (Figure 15) were designed to behave consistently across platforms, appearing at the top of the screen on both smartphones and smartglasses. This placement was intentionally chosen to ensure visibility regardless of device orientation, screen size, or available interaction methods. Although their placement remains uniform, the visual styling of the notification differs slightly to reflect the aesthetic requirements of each platform.

On smartphones, notifications follow the Material Design's *Snackbar* [59] conventions, with standard padding, shape, and font size, tailored for touch-based interfaces and larger visual elements.

On smartglasses, where space is more constrained and minimal visual distraction is essential, the notifications are visually more compact and subtly styled to integrate with the heads-up display nature of the device.

To achieve this top-placement behavior, the implementation leverages the *SnackbarHost* [59] mechanism provided by the *Scaffold* system. A custom *SnackbarHost* was created to override the default positioning (which by convention appears at the bottom of the screen, similar to a toast) and enforce the desired placement at the top. In addition, a specialized ExtendedSnackbarHostState class was introduced to extend the default state management capabilities.

This extended state is consumed by the Notification Manager (described in detail in chapter 4.2.3), which centralizes the snackbar lifecycle management, determining when to display a notification, how long it remains visible, and under which conditions it should be dismissed. This separation of responsibilities ensures that the notification system remains both visually consistent and operationally robust across different devices and interaction contexts.

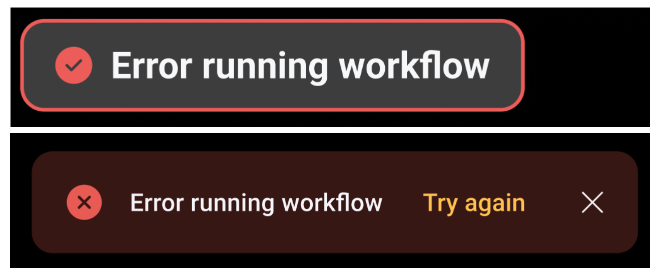


Figure 15. Notification design for the smartglasses (on top) and smartphones (on the bottom)

4.2.2.8 Picking Card

The Picking Card component (Figure 16) was designed to present essential workflow information in a clear and actionable manner, supporting the user during the picking process in a warehouse environment. Its primary role is to display key data such as the article number, which identifies the item to be picked; the pick location, which indicates where in the warehouse the item is stored; etc. This ensures that workers can quickly access the information necessary to complete each step of the workflow efficiently. Although the functional purpose of the Picking Card remains consistent across platforms, its visual presentation is adapted to the constraints and interaction models of different devices.

On smartphones, the component employs larger fonts, standard iconography, and a color scheme optimized for touch-based interaction and larger screen real estate. Conversely, on smartglasses, where screen space is limited and minimal distraction is critical, the component adopts a more compact layout, with adjusted font sizes, simplified icon display, and subtle color variations to maintain readability without overwhelming the user's field of view.

The component reflects the current workflow state through three visual states:

- **Idle**: Represents upcoming steps that are pending execution, such as picking the correct quantity or delivering the item.
- **Highlighted**: Indicates the active step in the workflow. For example, when the user needs to locate a specific article, its card appears highlighted.
- **Disabled**: Denotes completed steps, signaling that no further action is required for that stage.

This state-based approach provides clear visual guidance, helping users understand their progress and next actions within the workflow.

Figure 16. Picking Cards States for Idle, Highlighted and Disable, respectively, for smartglasses (left) and smartphones (right)

4.2.2.9 Scanner

The Scanner component enables barcode and QR code reading within the application and is implemented as a reusable multiplatform element. It supports various functional contexts, such as scanning items during workflows or authenticating users during login.

The component leverages Kotlin Multiplatform's *expect/actual* mechanism to handle platform-specific camera and permission logic while exposing a unified API. The shared module declares the expect interfaces for *ScannerCamera* and *rememberCameraPermissionState*, while each platform provides its actual implementation.

```
expect class ScannerCamera {
    fun startPreview()
    fun stopPreview()
}

expect fun rememberCameraPermissionState(): CameraPermissionState
```

Code Block 10. ScannerCamera and Camera Permission State

The Scanner accepts configuration parameters that define its behavior and presentation, including the list of supported barcode formats (via the CodeType enumeration), text for permission-denied states, labels for navigation to settings, and additional user instructions. Internally, it manages permission requests and dynamically displays either the scanning interface or a permission prompt. Once active, the Scanner continuously processes frames from the camera feed, detecting supported barcode formats and returning the scanned value to the *onScanned* callback. If the callback returns true, scanning stops to prevent duplicate reads.

```
Scanner(
    supportedFormats = listOf(CodeType.QR_CODE, CodeType.EAN_13),
    onScanned = { value ->
        handleScannedValue(value)
        true // Stops scanning after successful read
    }
)
```

Code Block 11. Scanner Composable Usage

On Android, the implementation uses *CameraX* [60] for camera control and preview rendering, combined with *ML Kit* [61] for barcode recognition. A *PreviewView* [62] displays the live feed, while an ImageAnalysis pipeline delegates frame processing to a BarcodeAnalyzer. Permissions are managed using *Accompanist Permission* [63] for seamless integration with Jetpack Compose.

On iOS, the implementation relies on *AVFoundation* [64]. An *AVCaptureSession* [65] configured with the rear camera captures the feed, while *AVCaptureMetadataOutput* [66] detects barcodes mapped from CodeType to *AVMetadataObject.ObjectType* [67]. The live preview is integrated into the Compose hierarchy via a native *UIView* [68]. Camera access is requested through *AVCaptureDevice.requestAccess(for: .video)* [69], with redirection to system settings if denied.

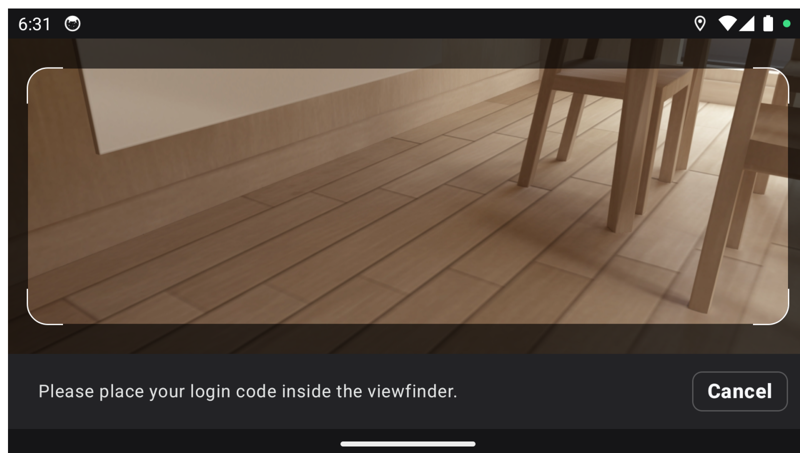


Figure 17. Scanner page running on an Android device

4.2.3 Dialog and Notification Management

In order to standardize user feedback mechanisms across heterogeneous device environments a dedicated dialog and notification management layer was introduced. This layer ensures consistent behavior, predictable lifecycle management, and simplified integration within the broader UI framework.

4.2.3.1 Dialog Handling

To manage modal interfaces effectively, a centralized DialogManager abstraction was implemented. This approach abstracts the rendering and visibility control of dialog components through a stack-based architecture. Each dialog is represented by an IDialog instance, which encapsulates its content and visibility state. From a design perspective, this methodology ensures that:

- Only one dialog is visible at any given time, mimicking a modal navigation stack.
- Dialogs can be nested or triggered sequentially without coupling their rendering to the local Composable hierarchy.
- Dialog visibility is handled externally, enabling device-agnostic dialog lifecycles and simplifying state restoration on configuration changes or UI recompositions.

This strategy facilitates reuse and consistency across platforms, particularly in constrained environments like smartglasses, where the system dialog behavior might differ significantly from conventional Android devices.

4.2.3.2 Notification Handling

In parallel, a NotificationManager was established to handle transient notifications through snackbars. Leveraging a *SharedFlow* [70] for event propagation and a coroutine-based consumer loop, this architecture supports:

- Asynchronous, thread-safe delivery of visual feedback messages,
- Decoupling of notification logic from the Composable UI tree,
- Cross-platform support, especially important for devices that do not use the typical Scaffold structure.

By initializing the snackbar host state at the application level and managing it centrally, the system ensures that snackbars are presented in a predictable and non-overlapping manner, regardless of the source triggering the event.

4.2.3.3 Reusability and Integration in External Projects

A key design consideration was ensuring that the UI module is reusable and easily integrable into external applications. By isolating dialogs and notifications into centralized managers with well-defined APIs and lifecycles, the system avoids tight coupling to any specific navigation framework or project structure. As a result:

- The UI module can be consumed by other projects with minimal setup or dependency injection configuration,
- Host applications retain full control over when and how dialogs and snackbars are rendered,

- The solution remains scalable and flexible, accommodating custom dialog implementations or notification visuals as required by the integrating project.

This modular design promotes clean separation of concerns, aligns with the principles of composability, and enables the UI library to serve as a plug-and-play toolkit for other teams or products within TeamViewer.

4.3 Application flow

The overall navigation and interaction structure of the application is illustrated in the UML state diagram shown in Figure 18**.** This diagram captures the major pages of the application, Landing Page, Login Page, Home Page, My Tasks, Workflow Execution, Workflow Completed, and Settings, as well as their navigation relationships, including authentication and logout transitions. Based on this flow, the application can be divided into three main flows: Authentication Flow, Workflow Flow, and Settings Flow.

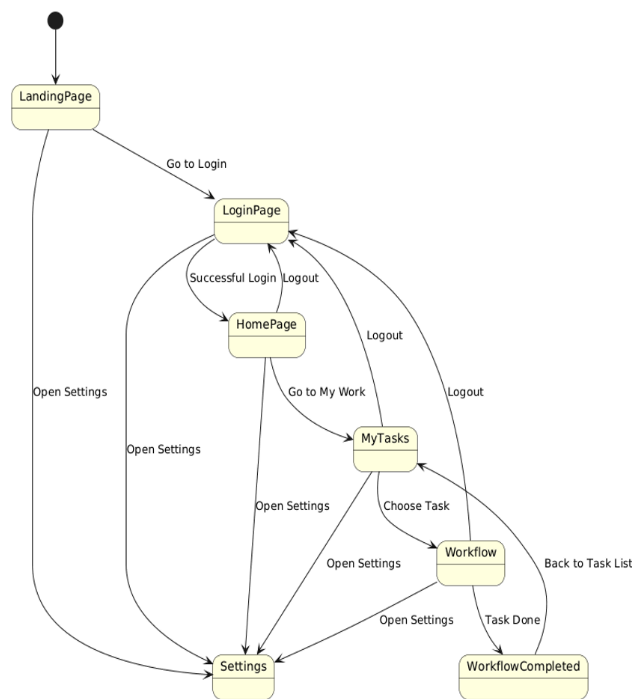


Figure 18. Application flow overview

4.3.1 Authentication Flow

The flow begins at the Landing Page, where the user can either access the Login Page or open the Settings menu.

On the Login Page (Figure 19), the default process requires the user to provide their username. After submission, the application launches a WebView that connects to the FCC instance defined in the *app_config.yaml* file (file placed within the application's files allowing local configuration like witch instance of the FCC to use and others, that will be addressed later on), ensuring secure and centralized authentication.

For smartglasses, the process is simplified: instead of entering credentials, the FCC provides a login QR code that can be scanned directly, removing the friction of text input on wearable devices. Smartphones may also take advantage of this QR login method.

The FrontlineBase package is used to execute the login and logout processes, ensuring consistent session handling across devices. Logout transitions are available from the Home, MyTasks, and Workflow pages, redirecting the user back to the Login Page.

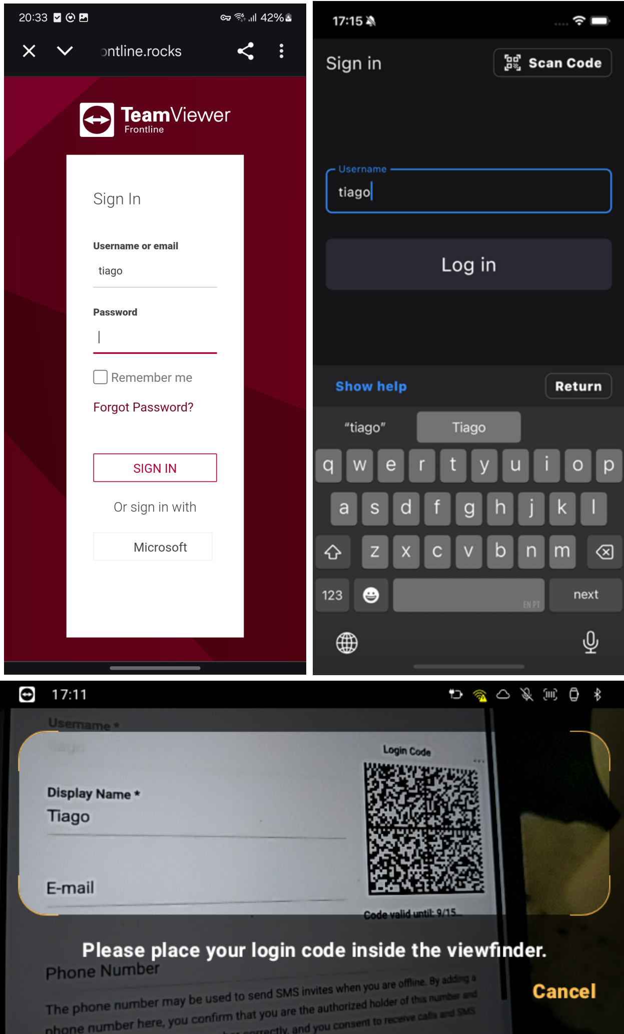


Figure 19. Login Page running on iOS (on the left) and on a Vuzix M300 (on the right)

4.3.2 Workflow Flow

After successful authentication, the user reaches the Home Page (Figure 12), which leads into the operational part of the application. From here, the user navigates to the My Tasks page, where available workflows are listed.

Selecting a workflow moves the user into the Workflow Page, where workflows are executed. The execution lifecycle, starting, monitoring, and completing workflows, is managed by the FrontlineBase package, providing robustness and consistency. Meanwhile, the application itself is responsible for rendering the user interface of the workflow execution. This separation allows workflows to be executed in the same standardized way across devices, while still providing UIs that are adapted to the device type.

Upon successful completion, the system transitions to MyTasks, allowing the user to continue working on further tasks.

4.3.3 Settings Flow

The Settings menu (Figure 20) is accessible from every screen of the application, making it a global entry point for device and application configuration.

On smartphones, settings are rendered with standard mobile UI components, while on smartglasses, settings are optimized for wearable layouts that are minimal and hands-free. This is enabled by a centralized Settings Source, which defines what settings are available; their grouping and categorization; which devices they apply to, their navigation destinations.

For example, some settings like Wi-Fi connectivity apply to all devices, while others (e.g. display brightness, volume) are specific to smartglasses. By centralizing this logic in the SettingsSource class, the UI simply renders what is defined. Any changes or additions to settings require only modifying the source definition, which then propagates automatically to both smartphone and smartglasses UIs. This architecture minimizes duplication, ensures consistency, and makes the system easier to evolve as new features and devices are introduced.

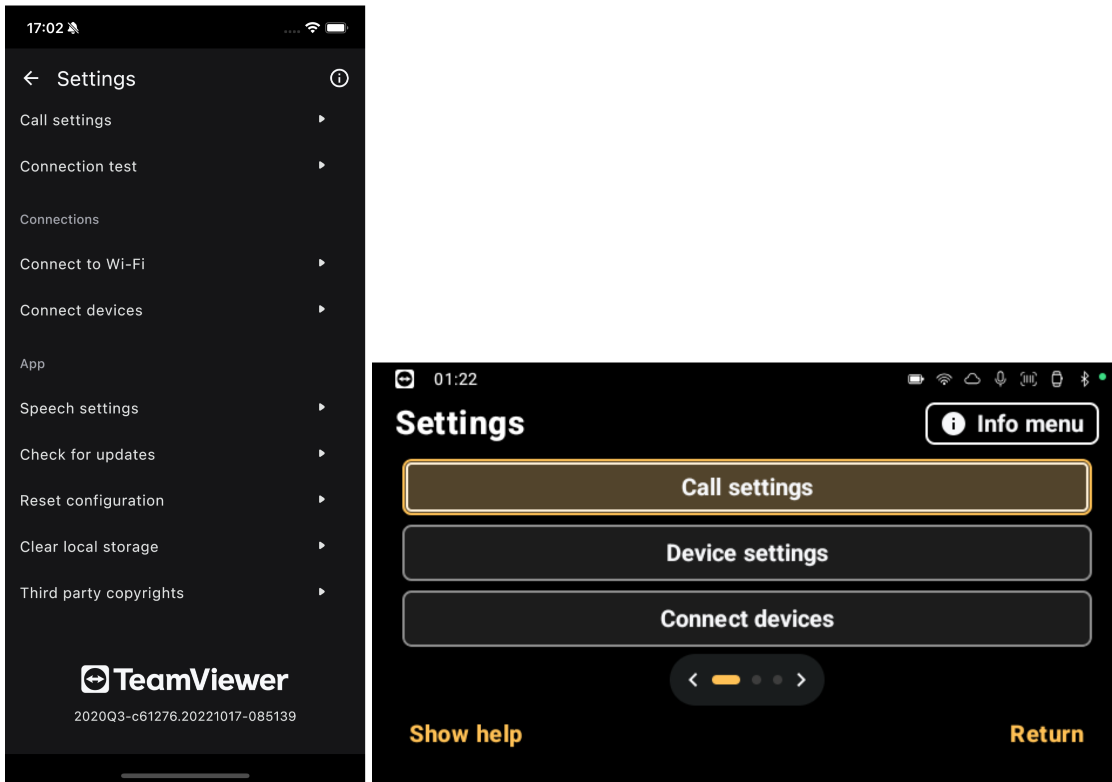


Figure 20. Settings Page running on an iPhone 16e (on the left) and on a Vuzix M300 (on the right)

4.4 Workflow Integration

The workflow integration introduces a unified approach that allows for simplified, flexible, and consistent UI creation across various platforms and device types. Through this approach, PEs can more easily design workflows that are adaptable to different devices, from smartphones to smartglasses, and maintain them efficiently.

By using Dependency Injection (DI), we are able to override FrontlineBase's UI module, offering full control over how workflows are rendered. This allows for seamless integration with the existing navigation and layout systems while ensuring flexibility and adaptability to meet the specific needs of the application.

4.4.1 Overriding FrontlineBase's UI module

In order to enable a more flexible and maintainable user interface architecture within the Frontline platform, we introduced a custom implementation that overrides the default UI module provided by FrontlineBase. This customization was essential to support our unified workflow rendering strategy and to accommodate the use of a modern UI library and

composable components. The following subsections describe the key components that were overridden or newly introduced to achieve this integration.

4.4.1.1 CatalogManager

The CatalogManager is responsible for storing UI catalogs developed with Creator on FCC and retrieving layout pages by their unique identifiers. In our implementation, this class was overridden to support a custom layout structure tailored to our simplified UI declaration model. This change allows for more intuitive and centralized management of layout definitions, aligning with the unified XML-based schema introduced in Chapter 4.4.2.

4.4.1.2 DataModelRenderer

The DataModelRenderer serves as the intermediary between the workflow engine and the layout system. Given a page identifier, it utilizes the CatalogManager to fetch the corresponding layout page and returns it for rendering. This component ensures that the correct layout is dynamically retrieved and rendered based on the workflow context.

4.4.1.3 LayoutFactory

Unlike the original FrontlineBase UI module, which lacks a direct equivalent, the LayoutFactory was developed to support our composable UI rendering approach. This class is responsible for rendering layout pages and orchestrating the creation of mappers for the header, footer, and main content sections. It acts as the central rendering engine, translating the XML-based layout definitions into fully functional UI components.

4.4.1.4 Mappers

To modularize the rendering process, we introduced dedicated mappers for each section of a layout page: header, content, and footer. These mappers interpret the XML structure and return a structured representation of the UI components, which is then parsed and rendered by the LayoutFactory. This separation of concerns enhances maintainability and reusability across different workflows.

4.4.1.5 StateManager

Workflows often require dynamic UI updates in response to user interactions or server-side events (e.g., stock unavailability during a picking operation). The StateManager handles these dynamic changes by managing and updating the state of UI components such as buttons, images, and text fields. It ensures that the UI remains responsive and contextually accurate throughout the workflow execution.

4.4.1.6 TemplateTagsManager

Due to the introduction of a new layout structure, the TemplateTagsManager was overridden to support the parsing of component properties (e.g., name, text, type). This class processes all tags defined in the XML layout and assigns the appropriate properties to each component. It also enables dynamic updates to these properties via the StateManager, ensuring consistency between the layout definition and runtime behavior.

4.4.1.7 UIManager

At the top of the UI architecture hierarchy is the UIManager, which integrates all the components and interfaces directly with the FrontlineBase workflow engine. It manages key interactions such as displaying dialogs (*dialogHandler*), rendering new interfaces (*showNewUserInterface*), updating existing interfaces (*updateUserInterface*), and providing user feedback through warnings and toast messages (*showWarning*, *showShortToast*). The UIManager serves as the central coordination point for all UI-related operations within the workflow system.

When a workflow is published in Creator (from FCC), its LayoutPages are stored in a catalog that becomes accessible through the CatalogManager. This allows the system to retrieve the correct layout page for the current workflow step at runtime. The sequence diagram (Figure 21) illustrates this rendering flow, showing how the UIManager coordinates the process: it requests layout definitions via the DataModelRenderer and CatalogManager, delegates rendering to the LayoutFactory and its mappers, and applies dynamic state updates through the StateManager.

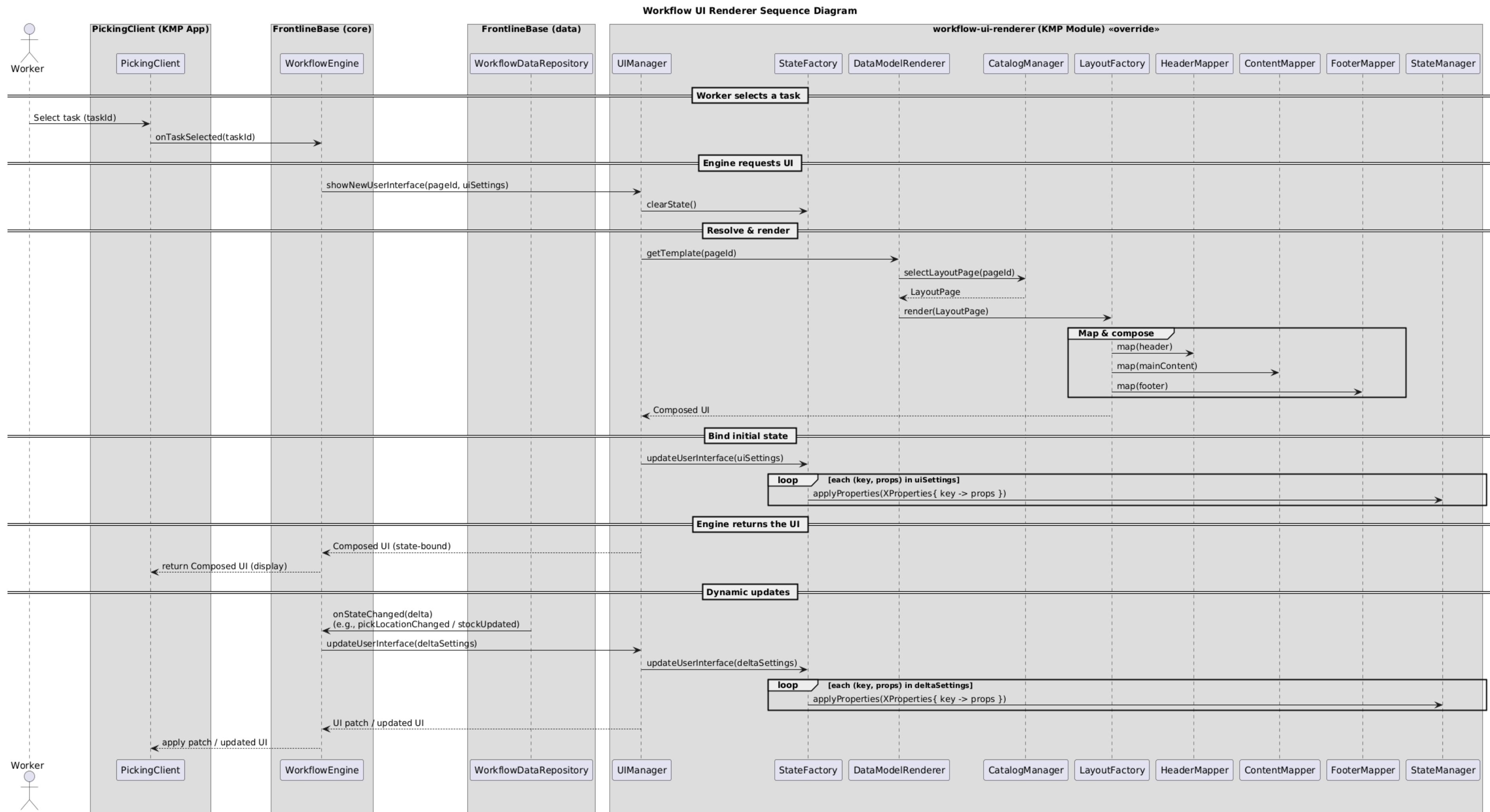


Figure 21. Sequence Diagram of the Workflow UI Renderer

4.4.2 New Workflow UI Definition

Defining the UI for workflows initially required maintaining multiple files and components, one for managing styles, another for layout templates, and yet another for content models. This fragmentation made workflow management complex and hindered the ability to maintain a consistent user interface across different workflows.

At first, the idea was to create a very simple and basic UI definition. Based on the most used layouts, the definition would only require mapping the values to be displayed into a template of these layouts. This was designed to be implemented in a lightweight JSON structure, keeping things minimal and without introducing significant complexity. In this model, the content to display would be defined by the PEs or customers, while the way it was displayed would be fully managed by the application itself. This ensured flexibility and adaptability across devices, while reducing the effort needed to create new workflows.

However, after presenting the first drafts to the PEs, their feedback highlighted the need for greater flexibility, not only in *what* was displayed, but also in *how* it was presented within the layout. The JSON based approach, while simple, proved too restrictive for these requirements.

To address this, we opted to continue with XML. Unlike JSON, XML allows for more expressive declarations in coding blocks, offering better readability and a higher degree of flexibility. With XML, both the structure and presentation of UI elements could be defined with much greater control, meeting the expectations of the PEs while still ensuring a unified approach.

After several iterations, the work evolved further into the concept of Native Pages, which serve as templates that balance flexibility with standardization. This approach will be discussed in detail in the following subsection.

4.4.2.1 Schema Structure

At the core of the schema is the <Catalog> element, which acts as a container for workflow pages. Each catalog is uniquely identified by a name and version, facilitating version control and configuration management. Within a catalog, two types of pages can be defined:

- **LayoutPage**: Offers full control over layout and content, ideal for complex workflows.
- **NativePage**: Provides a simplified structure for common, purpose-specific screens.

The unified schema supports both declarative and dynamic UI behavior. LayoutPages allow developers to define custom content and structure, making them suitable for workflows with complex logic or variable data. NativePages, by contrast, are optimized for rapid development of standard screens, such as confirmations, data entry, or status updates.

This dual-structure model ensures that developers can choose the most appropriate approach for each workflow scenario, balancing flexibility and efficiency. Dynamic content binding is

supported throughout, enabling real-time updates based on user interaction or backend responses.

The **LayoutPage** structure offers full control over layout and content, making it ideal for complex workflows. It is composed of three main sections:

- **Header** (optional): Can include a PickingProgressBar or a DefaultHeader with configurable buttons.
- **MainContent** (required): Contains one or more Panel elements, each of which can include reusable components such as:
    - InfoCard: Displays labeled information with optional icons.
    - Image: Embeds visual content with accessibility support.
    - ActionList: Lists actionable items.
    - Grid: Organizes components in horizontal or vertical layouts.
- **Footer** (optional): Supports either a DefaultFooter with action/help buttons or an InformationFooter for static messages.

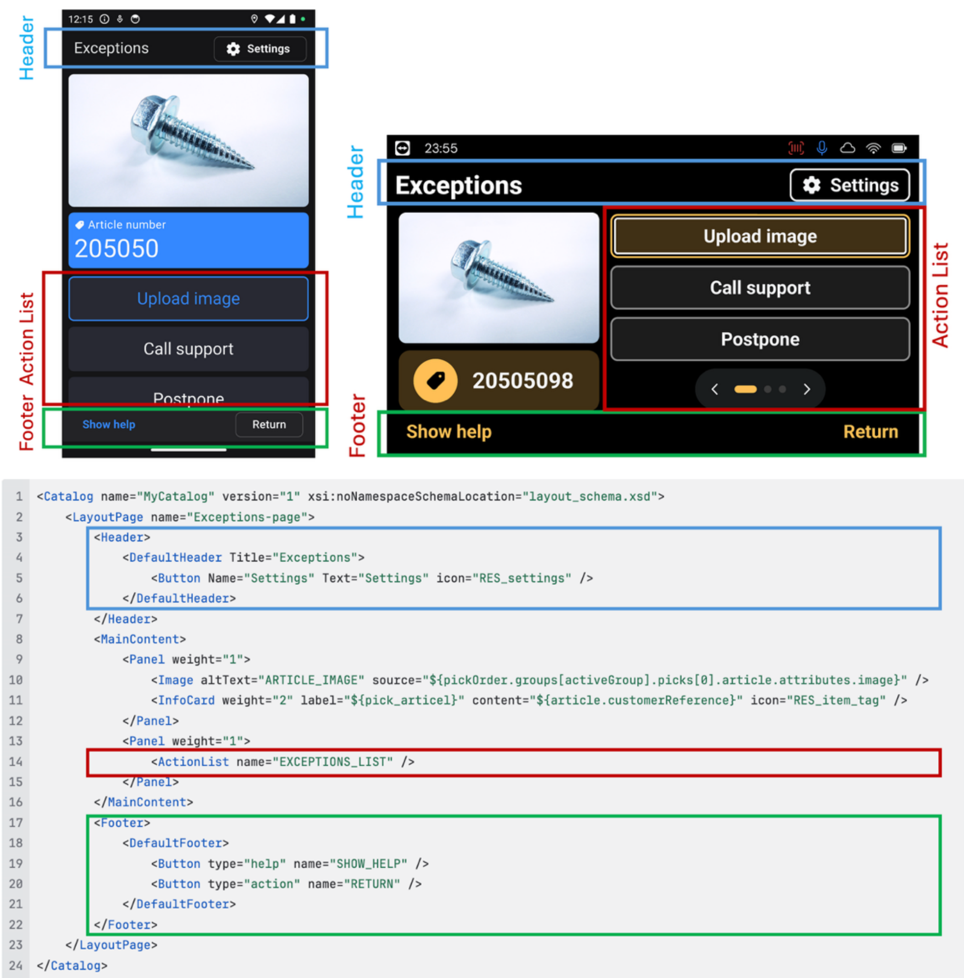
Each component supports layout attributes like weight, verticalWeight, and horizontalWeight, allowing for responsive design across devices.

To illustrate this structure, Figure 22 presents a real-world example of a LayoutPage used in a picking workflow. The screen includes a progress bar in the header, an image and multiple info cards in the main content area, and a default footer. All of this is defined in a single XML file, demonstrating the clarity and modularity of the new schema.

Figure 22. Example of a LayoutPage definition and its rendered user interface

Another example of the new layout schema can be seen in Figure 23. Adaptiveness of the LayoutPage: layout definition and rendered user interface of the Exceptions page, which defines and renders the *Exceptions page*. This page is a common element of a picking workflow, used when an issue occurs with the item being processed, such as postponing the task, canceling the pick, or reporting a damaged product.

The first panel displays the article image and related metadata, while the second panel contains the ActionList component. This component dynamically renders the available exception actions, which are defined in the workflow and can vary depending on the operational context and business rules. When the list of exceptions exceeds the available space in its panel, the interface adapts to the device type: on smartglasses, the list uses a pagination mechanism, showing a limited number of items per page and providing navigation controls; on smartphones, the list becomes scrollable, leveraging the precision and familiarity of touch-based navigation. This adaptive design ensures that the same layout definition delivers an optimized user experience across devices with very different interaction paradigms.




```
1  <Catalog name="MyCatalog" version="1" xsi:noNamespaceSchemaLocation="layout_schema.xsd">
2      <LayoutPage name="Exceptions-page">
3          <Header>
4              <DefaultHeader Title="Exceptions">
5                  <Button Name="Settings" Text="Settings" icon="RES_settings" />
6              </DefaultHeader>
7          </Header>
8          <MainContent>
9              <Panel weight="1">
10                 <Image altText="ARTICLE_IMAGE" source="${pickOrder.groups[activeGroup].picks[0].article.attributes.image}" />
11                 <InfoCard weight="2" label="${pick_articel}" content="${article.customerReference}" icon="RES_item_tag" />
12             </Panel>
13             <Panel weight="1">
14                 <ActionList name="EXCEPTIONS_LIST" />
15             </Panel>
16         </MainContent>
17         <Footer>
18             <DefaultFooter>
19                 <Button type="help" name="SHOW_HELP" />
20                 <Button type="action" name="RETURN" />
21             </DefaultFooter>
22         </Footer>
23     </LayoutPage>
24 </Catalog>
```

Figure 23. Adaptiveness of the LayoutPage: layout definition and rendered user interface of the Exceptions page

The **NativePage** structure, by contrast, is designed for workflows that follow a more rigid and standardized layout. It is particularly suited for task-specific screens that require minimal customization. A NativePage includes:

- **Header** (optional): Same options as LayoutPage.
- **Main Element** (required): One of the following:
    - WorkflowInfo: Displays a message and status (e.g., done, info, or loading), with optional ProgressBar.
    - ShortPick: Used for when the amount the worker needs to pick is greater than the existing stock. One panel is used for the input, and the other can be personalized by the PEs.
    - UploadImage: Displays a camera where the worker can take a photo of the article to upload it. Gives the same freedom of personalization as the ShortPick.
- **Footer** (optional): Supports DefaultFooter or InformationFooter.

NativePages encapsulate common UI patterns into predefined templates, reducing development time while still allowing for customization through attributes and bindings.

Figure 24 shows an example of a NativePage used for a short picking task. The screen includes a title in the header, a simplified content section displaying key data points (that can be customized by the PEs), and a footer. This example highlights how NativePages streamline the creation of focused, reusable screens with minimal configuration. On smartphones, an input field triggers the device's native keyboard. On smartglasses, however, typing on native keyboard is impractical, so a virtual keyboard is displayed instead. The worker can then use the interaction methods supported by the smartglasses to enter the amount to be picked.

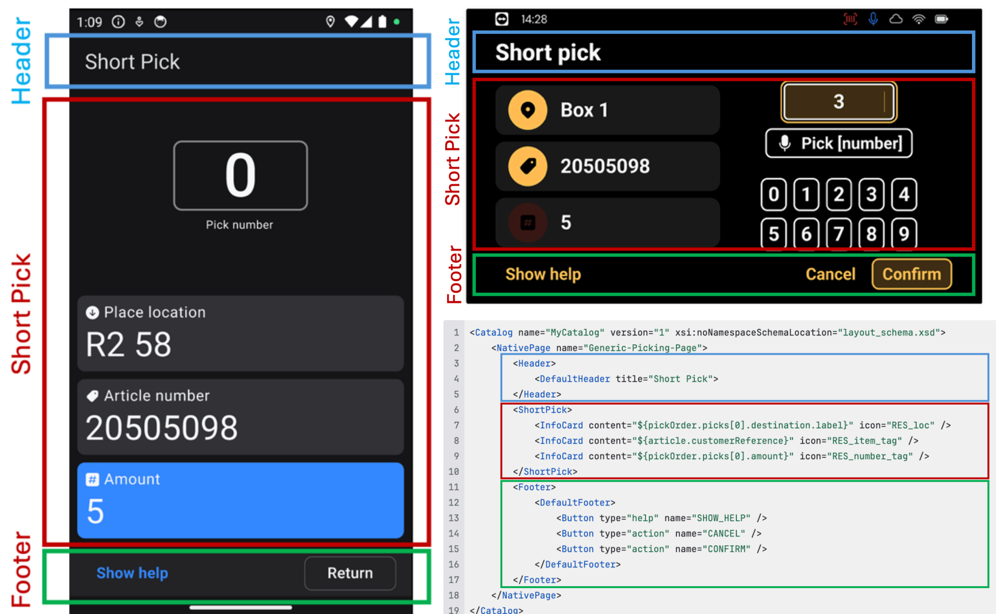


Figure 24. Example of a NativePage definition and its rendered interfaces.

Together, Layout Pages and Native Pages form a dual-structure model that supports both declarative and dynamic UI behavior. Developers can choose the most appropriate structure for each workflow scenario, balancing flexibility and efficiency. Dynamic content binding is supported throughout, enabling real-time updates based on user interaction or backend responses.

4.5 Interaction Methods

The application supports two primary interaction methods: voice commands and physical button navigation. These interaction methods are critical to ensuring accessibility, efficiency, and adaptability of the user interface across different hardware form factors, including smart glasses and wearable devices.

4.5.1 Voice Control Integration

Voice commands play a central role in the hands-free operation of the application, particularly in industrial environments where users often wear gloves or operate machinery. For this purpose, we integrated a voice recognition package provided by Frontline, which simplifies the process of listening for voice input and handling recognized commands through a built-in event handler.

4.5.1.1 Command Categorization

To accommodate the voice recognition system within the application architecture, a dedicated module named SpeechRecognitionModule was developed. This module manages and organizes the voice commands into four distinct categories:

- **Global Commands**: These commands are always active, regardless of the current screen or context. They include universal actions such as “go back” or “open help.”
- **Local Commands**: Commands that are only available within the current screen or UI context. For example, commands specific to a particular form or selection screen.
- **Workflow Commands**: Commands that are only valid during the execution of a workflow. These facilitate operations like scanning an item, confirming an action, or progressing to the next step.
- **Temporary Commands**: These are context-sensitive commands, typically activated when transient UI components such as drawers or dialogs are visible. Once the temporary component is dismissed, these commands are removed from the registry.

This categorization enables a clear and predictable interaction model, ensuring that users can rely on consistent voice behavior throughout the application while minimizing command collisions or ambiguities.

4.5.1.2 Command Management and Execution

The SpeechRecognitionModule is responsible for interfacing with the voice recognition package provided by Frontline. While the package is capable of detecting when a spoken command matches a registered grammar entry, it does not contain any built-in mechanism to determine which action should be executed in response.

To address this, the SpeechRecognitionModule introduces a command registration mechanism that not only adds voice commands to the grammar of the speech recognition engine but also associates each command with:

- An *onClick* callback: This represents the action that should be triggered when the voice command is recognized.
- A *MutableInteractionSource* [71]: This is the same interaction source used by the corresponding UI element (e.g. a button) and is used to simulate a press interaction, providing visual feedback to the user on which component is being pressed.

This design ensures that when a voice command is spoken and matched, the system can both:

1. Trigger the associated action logic seamlessly.
2. Provide immediate visual feedback by simulating the component's press state, mirroring the behavior of a physical interaction, and thereby reinforcing clarity and user confidence in the voice-driven interface.

This approach not only bridges the gap between voice input and UI logic but also ensures that the application's interaction model remains consistent across both voice and traditional input methods.

4.5.1.3 User Feedback

To further enhance the user experience and ensure transparency, the application presents a snackbar notification (Chapter 4.2.2.7) every time a command is successfully recognized and triggered. The snackbar displays the spoken command, offering the user reassurance that the system interpreted their input correctly.

4.5.2 Physical Button Navigation

In addition to voice input, the application supports navigation and interaction via physical buttons. This is particularly important for devices with limited or no touch capabilities, such as smartglasses, wearable scanners, and rugged handhelds, where reliable operation must be ensured even in demanding industrial environments.

4.5.2.1 Button Mapping and Event Handling

The application supports interaction through physical hardware keys, providing an alternative or complementary input modality to touch-based navigation. Key events, such as those generated by Volume buttons and directional pads (D-Pads), are captured at the UI layer using Jetpack Compose's *Modifier.onKeyEvent* [56]. This approach enables the system to interpret hardware input in a consistent and platform-independent manner.

To maximize flexibility, the application does not enforce fixed mappings between keys and actions. Instead, the associations are defined externally in the configuration file (*app_config.yaml*). This allows physical keys to be reassigned according to the requirements of a particular device, environment, or user preference.

The available navigation actions that can be mapped to hardware inputs cover the essential interaction primitives required for efficient navigation within the interface. Currently it supports the following actions:

- **Directional navigation**: moving focus *up*, *down*, *left*, or *right*.
- **Sequential navigation**: advancing to the *next* or returning to the *previous* interactive element.
- **Selection and confirmation**: triggering a focused element, equivalent to an *enter* or *select* operation.

An example of button mapping configuration is shown below:

```yaml
Input:
  Keys:
    500: SELECT
    KEYCODE_VOLUME_DOWN: PREVIOUS
    KEYCODE_VOLUME_UP: NEXT
    KEYCODE_DPAD_RIGHT: RIGHT
    KEYCODE_DPAD_LEFT: LEFT
```

Code Block 12. example of button mapping configuration in *app_config.yaml*

In this example, the custom key with code *500* is mapped to the *SELECT* action, functioning as a confirmation key. The Volume Down and Volume Up buttons are mapped to *PREVIOUS* and *NEXT*, enabling sequential backward and forward navigation respectively within the user interface. Finally, the D-Pad Left and D-Pad Right keys are mapped to *LEFT* and *RIGHT*, which enable a more horizontal directional navigation.

By decoupling key events from their resulting actions, the system ensures adaptability across different device classes. For instance, a handheld scanner equipped with volume keys may use them for sequential navigation, while a device with a D-Pad may map its directional controls directly to focus movement. This design promotes configurability, accessibility, and device-

agnostic interaction, making the application suitable for heterogeneous hardware environments.

4.5.2.2 Ensuring Focus Consistency

It was observed that *focusManager.moveFocus(...)* only functions correctly when a UI element is already focused. In cases where no element initially has focus, focus movement fails silently, resulting in unresponsive behavior.

To resolve this, the application follows Android's Compose focus best practices:

- All interactive elements are explicitly marked as focusable using *Modifier.focusable().*
- Logical focus scopes are defined using *FocusTarget* and *FocusGroup*.
- Custom directional behavior is handled through *focusProperties*, allowing more intuitive navigation flows in complex layouts.

These improvements ensure that the Compose focus system can accurately track and manage the currently active element, providing a consistent experience across different screens and contexts.

4.5.2.3 Managing Repeated Key Events

One other challenge encountered with physical button handling was managing repeated key events that occur when a button is held down. Without additional control, this can lead to rapid, unintended navigation or multiple interactions being triggered.

To address this, a key state tracking system was implemented. When a key is first pressed, it is marked as "active" and handled once. Any subsequent key events for that same key are ignored until the key is released and re-pressed. This prevents duplicate actions during prolonged presses and ensures a more predictable and user-friendly experience.

4.5.2.4 Fallback Mechanism for Initial Focus

In scenarios where the focus chain has not yet been initialized, such as immediately after launching a screen or dialog, a fallback mechanism is required to initiate navigation.

To handle this, the application programmatically dispatches synthetic *KEYCODE_TAB* events directly to the Android view layer. This mimics the behavior of tab-based navigation on traditional platforms and allows focus traversal to begin even if no element is currently active.

4.5.2.5 Resulting Interaction Experience

With the integration of these mechanisms, physical button interaction is now fully supported in a consistent and intuitive manner. Users can traverse interface elements and trigger actions

reliably using only the physical controls available on the device. This significantly improves usability in environments where touch or voice input may be unavailable and contributes to the overall adaptability and accessibility of the application across a wide range of hardware platforms.

5 Testing and Validations

This chapter focuses on the evaluation processes implemented to ensure the reliability, efficiency, and overall quality of the developed solution. It outlines the systematic approach taken to validate both the technical performance and user experience aspects of the application. The goal is to confirm that the solution meets its intended objectives and performs optimally under various conditions.

5.1 Performance Benchmarks

Evaluating the performance of mobile applications across different devices and platforms is a critical step in understanding their usability and suitability for real-world deployment. This evaluation focuses on four key factors: startup time, application size, CPU usage, and memory usage.

Startup time directly affects the perceived responsiveness of the application, while application size influences storage requirements, download times, and deployment feasibility. These two metrics were chosen not only for their impact on user experience but also to allow a comparison with earlier studies discussed in Chapter 3.3.3, providing insight into how the technology has evolved and whether recent advancements translate into tangible performance gains.

CPU and memory usage extend this analysis by capturing the runtime efficiency of the applications. CPU utilization reflects the computational effort required to execute workflows, with implications for responsiveness and battery life, while memory usage highlights how heavily the application relies on device resources.

5.1.1 Startup Performance Comparison

The startup time was measured between the app developed in this project and the current app in production, Workplace App, across multiple devices, including Samsung A52, a Vuzix M400,

and an iPhone 16e (Figure 25). These measurements provide a comparative basis to assess cross-platform performance and identify potential bottlenecks. To ensure accurate and consistent results, platform-specific profiling tools were used:

- On Android, the Android Profiler [72] within Android Studio was employed to measure startup time. This tool records the lifecycle of the application, from process creation to the rendering of the first frame, offering precise insight into perceived launch latency.
- On iOS, startup times were measured using Instruments [73] in Xcode, specifically the Time Profiler instrument, which provides a detailed timeline of execution during the application launch phase.

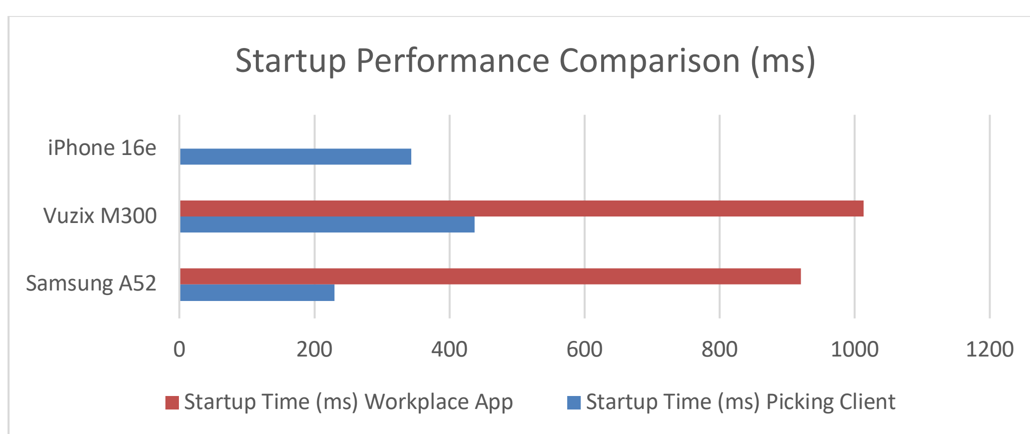


Figure 25. Startup performance comparison between iPhone 16e, Vuzix M300 and Samsung A52

The experiments were conducted under consistent conditions, with each application launched multiple times to minimize the impact of external variability. The results demonstrate that the Picking Client is consistently faster to start than the Workplace App across all tested Android devices. For the iPhone 16e, only results for the Picking Client are available, since there is no support for iOS from the Workplace App side.

These findings align with Jacob Ras's earlier evaluation [35], which showed that Compose Multiplatform's startup times were already close to those of native applications and faster than Flutter on Android. The results in this study suggest that the technology has continued to improve since then, with the Picking Client demonstrating consistently shorter startup times than the production Workplace App and competitive performance on iOS. This progression reflects the ongoing optimization of the KMP runtime and its integration with platform-specific APIs, underscoring its growing maturity as a cross-platform solution.

5.1.2 Application Size Comparison

Application size is another important dimension of performance evaluation, as it affects download times, installation feasibility, and storage consumption on end-user devices. Large

applications can create barriers for users with limited device storage or constrained network connectivity.

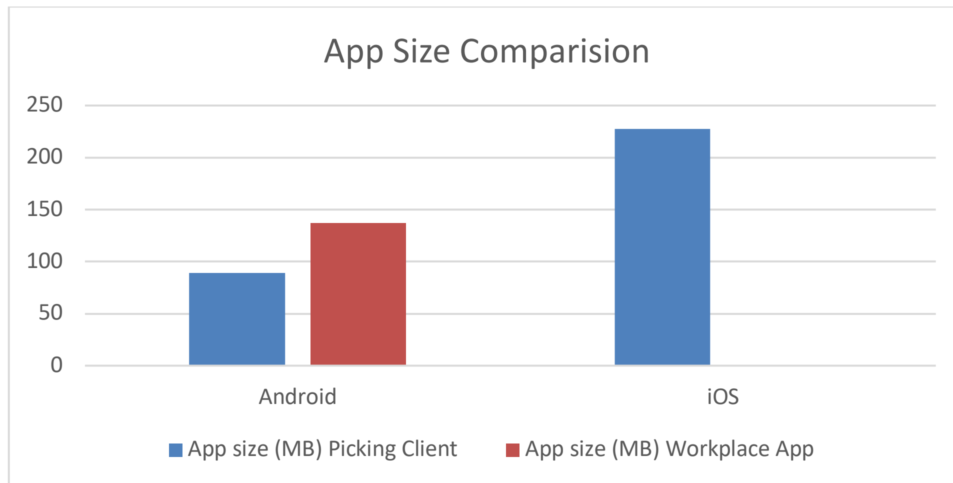


Figure 26. App size performance comparison (in MB) between Android and iOS

The results show that application size varies both between the two apps and across platforms (Figure 26). On Android, the Workplace App is heavier, mirroring its poorer startup performance. On iOS, however, the Picking Client is considerably larger than on Android. This discrepancy is influenced by platform-specific factors: iOS applications often include multiple architectures, bitcode, and additional resources to support a wide range of devices and app thinning, resulting in larger binaries.

Although the Picking Client has a smaller binary size, it also includes fewer modules and functionalities compared with the Workplace App. The teams responsible for the Workplace App have invested significant effort in optimizing its package size, which is crucial in real-world scenarios where devices are managed through Mobile Device Management (MDM) where application sizes are very important. Maintaining a compact application footprint ensures faster deployment, easier updates, and smoother operation on constrained devices, whereas larger applications can face challenges in enterprise environments with strict storage and management requirements.

Jacob Ras [35] highlighted the larger binary size of KMP applications on iOS due to the inclusion of the Skia rendering library, compared with both native and Flutter equivalents. The results presented here confirm that iOS binaries for KMP remain heavier than their Android counterparts, though the impact is now framed in the context of enterprise deployment, where package size directly affects device management and update processes. On Android, however, the Picking Client's smaller footprint compared to the Workplace App suggests that the overhead of using KMP is minimal, and in some cases even advantageous. This indicates that

while some of the challenges identified in earlier studies persist, particularly on iOS, the overall footprint of KMP-based applications is increasingly competitive in real-world scenarios.

5.1.3 CPU Usage Comparison

CPU utilization is a key performance indicator that directly affects both battery life and device responsiveness. Applications that require excessive processing power may not only drain the battery faster but also degrade the overall user experience, particularly on lower-end or wearable devices with limited processing capabilities.

To ensure realistic results, CPU usage was measured while running a basic workflow (Figure 27). This workflow consisted of triggering a notification, displaying a dialog, and transitioning to the next step. During execution, the highest CPU usage value observed was recorded for both applications. In both the Picking Client and the Workplace App, the peak in CPU consumption occurred during the loading of the workflow, representing the most resource-intensive part of the run.

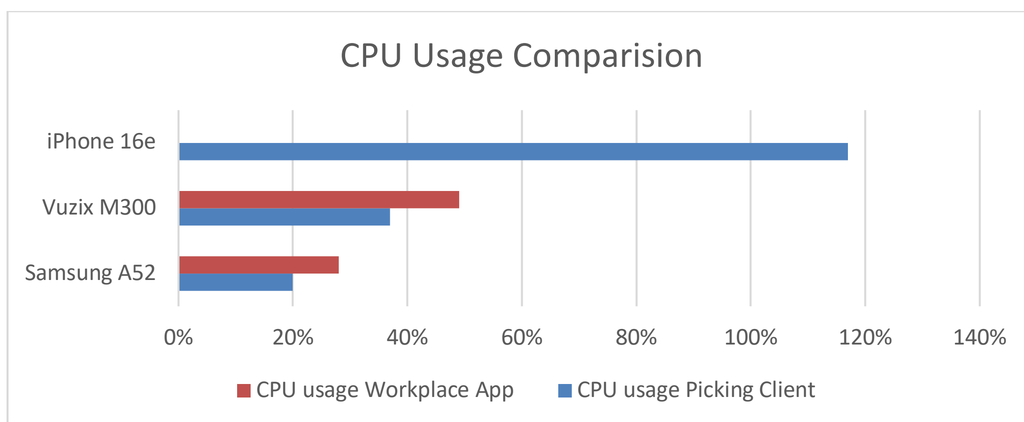


Figure 27. Peak CPU utilization across devices during workflow execution

The results clearly indicate that the Picking Client is more efficient across all tested devices, consistently requiring fewer CPU resources than the Workplace App. The difference is especially pronounced on the Vuzix M400, where the Picking Client demonstrates a 12% reduction in CPU usage, a significant factor on wearables with constrained resources.

On the iPhone 16e, the CPU usage exceeded 100%. This is not an anomaly but rather a result of how iOS reports CPU consumption. On iOS, CPU percentages are calculated per core; thus, a value of 117% indicates that the application utilized more than one logical core during execution, specifically about 1.17 cores. This finding reflects the high level of parallelism leveraged by Kotlin Multiplatform on iOS hardware, suggesting efficient multithreading capabilities rather than inefficiency.

5.1.4 Memory Usage Comparison

Memory usage is another critical dimension of performance evaluation, as high memory consumption can lead to slower performance, increased likelihood of application termination by the operating system, and reduced multitasking capability. This is especially relevant for wearables and lower-end mobile devices, where memory resources are often limited.

Memory profiling was conducted in the same basic workflow scenario as CPU testing (Figure 28), involving a notification trigger, a dialog, and a transition to the next step. For each run, the highest memory usage observed during execution was collected.

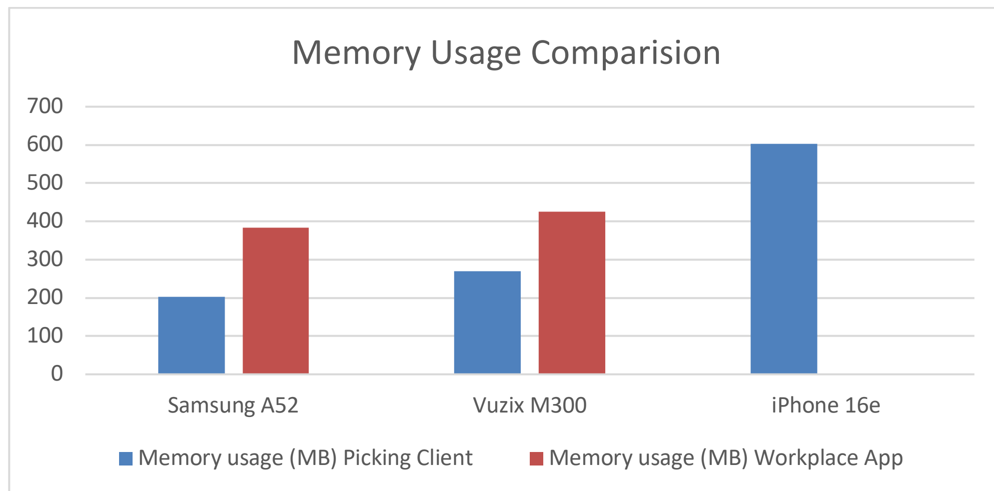


Figure 28. Peak memory usage across devices during workflow execution (in MB)

The Picking Client demonstrates significantly lower memory usage compared to the Workplace App across all shared platforms. On the Samsung A52, the memory footprint of the Picking Client is almost half that of the Workplace App. Similarly, on the Vuzix M300, the difference is substantial, with the Picking Client consuming 155 MB less memory. These results indicate that the Picking Client is more memory-efficient, which translates into improved stability and smoother performance on constrained devices.

On iOS, the Picking Client shows relatively high memory usage (603 MB). This elevated value can be attributed to multiple factors, including the Kotlin Multiplatform runtime overhead, the inclusion of multiple platform-specific frameworks, and possibly less aggressive memory optimization compared to mature native iOS applications. However, when viewed alongside the improved CPU and startup performance, this trade-off appears acceptable, particularly given the memory capacity of modern iOS devices.

5.2 Comparative Evaluation

This section provides a comparative evaluation of the redesigned application and workflow definition model against the legacy system. The objective is to demonstrate how the unified

schema and adaptive design principles improve usability, maintainability, and cross-device consistency. The evaluation is divided into two parts. The first part (Chapter 5.2.1) examines the application itself, comparing representative screens and interaction patterns across smartphones and smartglasses. The second part (Chapter 5.2.2) addresses the workflows, showing how the underlying schema resolves limitations of the legacy definitions and enables more reusable, flexible, and adaptive user interface descriptions.

5.2.1 Application Comparison

To validate the improvements introduced by the new design, several representative application screens were compared with their legacy counterparts: the Landing page, Settings (including the *Info* and *Device Settings* menus, as well as *Reset Configuration*), and the Login page. This comparison highlights how the unified schema adapts interfaces to the characteristics of smartphones and smartglasses while maintaining a consistent user experience.

5.2.1.1 Landing Page

The landing page was redesigned to improve clarity and usability across devices while maintaining the same core functionality (Figure 29). The layout now centers the login section with the application logo positioned above, moves the Settings button to the header for easier access, and introduces a Show Help button to inform users about available voice commands, reinforcing the multimodal interaction approach described in Chapter 4.5.1. The Exit App option remains in the footer, ensuring it is still easily accessible.

On smartglasses, the interface adopts a black background to enhance readability in near-eye displays, as discussed in Chapter 4.2.1 on theming and color adaptation. These adjustments ensure that the application remains visually clear and ergonomic on constrained screens while providing a consistent experience across platforms.

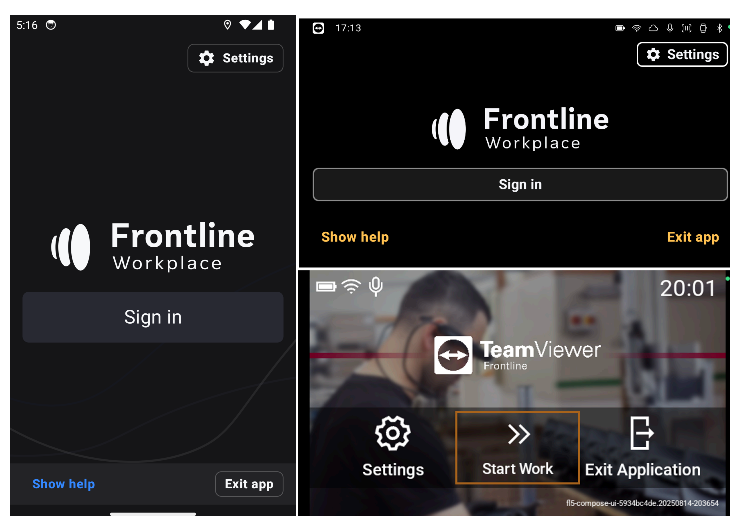


Figure 29. Landing Page on the Frontline Workplace app (bottom-right) and on the Picking Client app for smartphone (left) and smartglasses (top-right)

5.2.1.2 Login

The login page has been redesigned to expand on the functionality of the legacy system (Figure 30). In addition to the scanner-based login used previously, the new design introduces the option to log in via keyboard input on smartphones through a WebView component (Figure 19).

This enhancement takes advantage of the smartphone's touch interface, allowing workers to quickly type their credentials when convenient. In scenarios where typing is impractical, such as when wearing gloves or operating in hands-free environments, the user can still rely on the QR code scanning method. This dual approach ensures a seamless and adaptable authentication experience across different usage contexts.

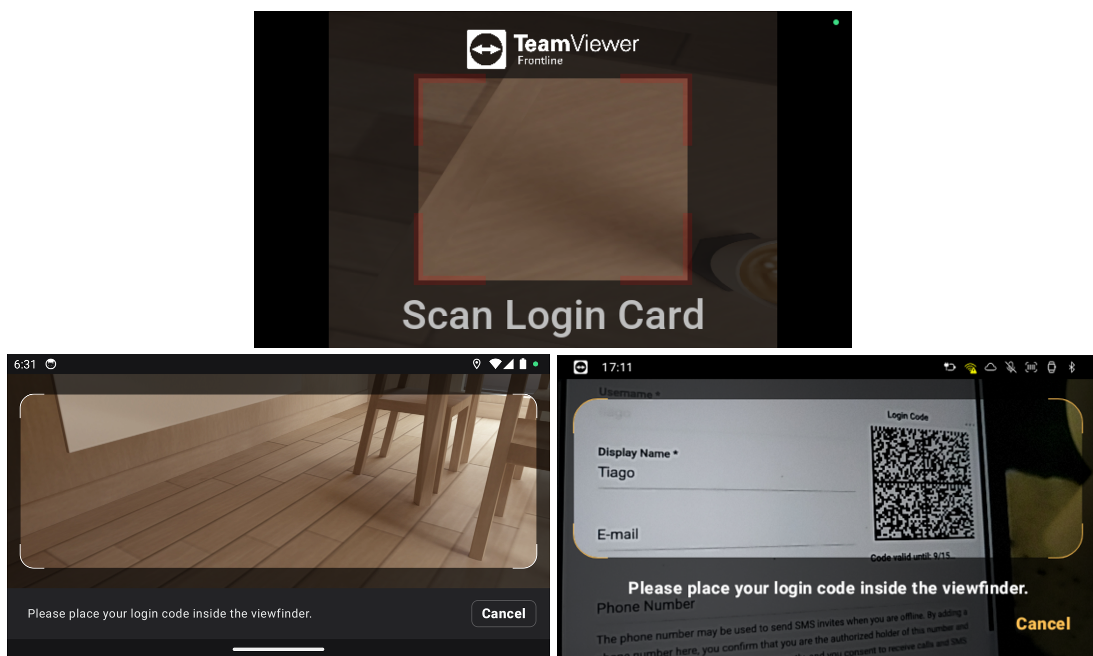


Figure 30. Login Scanner on the Frontline Workplace app (top) and on the Picking Client app for smartphone (bottom-left) and smartglasses (bottom-right)

5.2.1.3 Settings

The settings page highlights one of the most significant differences between smartphones and smartglasses (Figure 31). On smartphones, settings are presented in a continuous scrollable list, leveraging the larger screen and natural interaction model of touch devices. On smartglasses, only three settings are shown per page, with a pager used for navigation. This resembles the current approach but improves on it by dynamically adapting the presentation to the device type and screen characteristics, an ability absent from the legacy implementation.

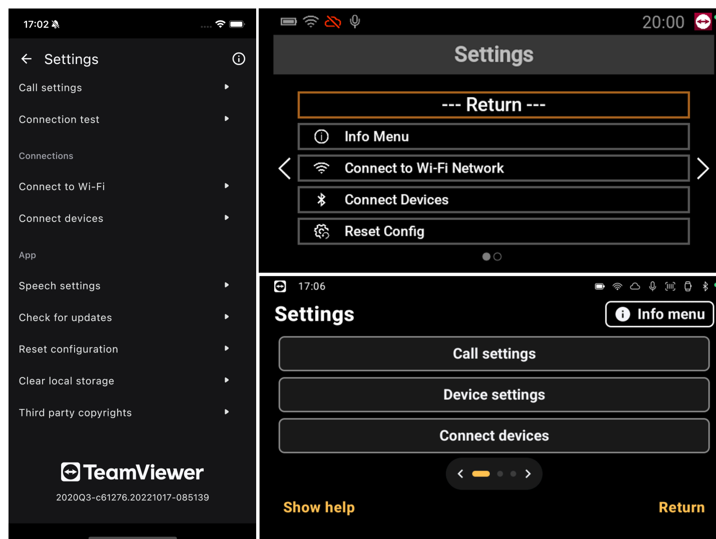


Figure 31. Settings Page on the Frontline Workplace app (top-right) and on the Picking Client app for smartphone (left) and smartglasses (bottom-right)

5.2.1.4 Info Menu

The Info section (Figure 32) is available on both platforms, but its navigation model adapts to the characteristics of each device. On smartphones, it uses a conventional tab-based menu for quick access through touch interaction, while on smartglasses it employs a side drawer to optimize limited screen space and support hands-free navigation. This design ensures that the application maintains a consistent visual identity while applying interaction patterns that respect the ergonomics and usability requirements of each form factor, improving discoverability and overall user experience.

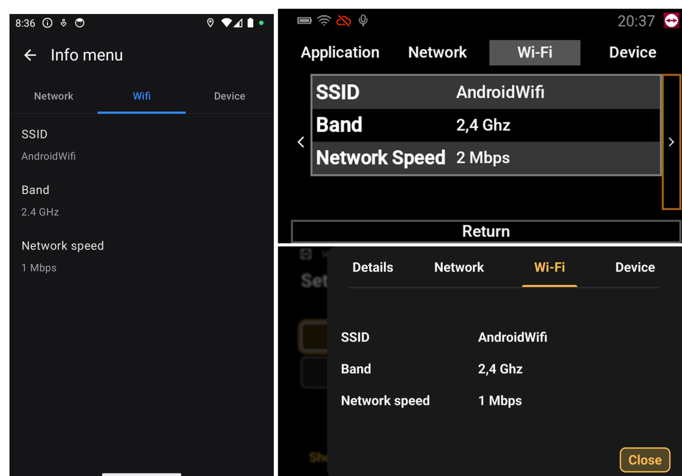


Figure 32. Info Menu on the Frontline Workplace app (top-right) and on the Picking Client app for smartphone (left) and smartglasses (bottom-right)

5.2.1.5 Device Settings

Device-specific configurations, such as brightness, volume, and screen margins, are exposed only on smartglasses. On smartphones these options are redundant, since such adjustments can be handled directly by the operating system. On smartglasses, however, providing these options inside the application improves usability, as these settings are less accessible through the native system.

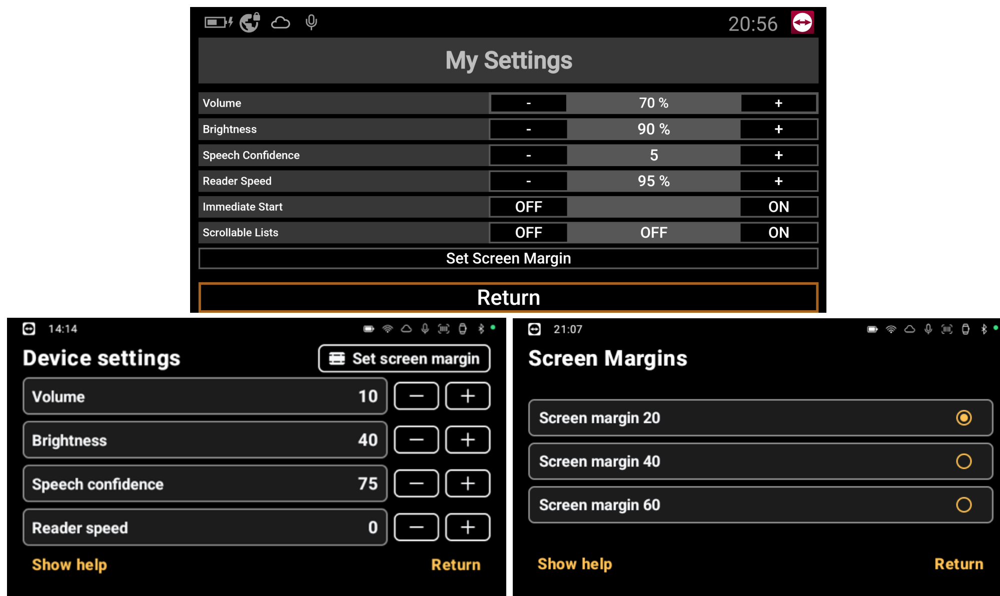


Figure 33. Device Settings Page on the Frontline Workplace app (top) and on the Picking Client and smartglasses (bottom)

5.2.1.6 Reset Configuration

The Reset Configuration option (Figure 34), located within the settings menu, provides an additional case where adaptive design is validated. Upon selection, a confirmation dialog is displayed before resetting the app's configuration (*app_config.yaml*), which contains parameters such as the connected FCC, button mappings, and screen margins. As described in Chapter 4.2.2.5 the dialog adapts its presentation to the device type.

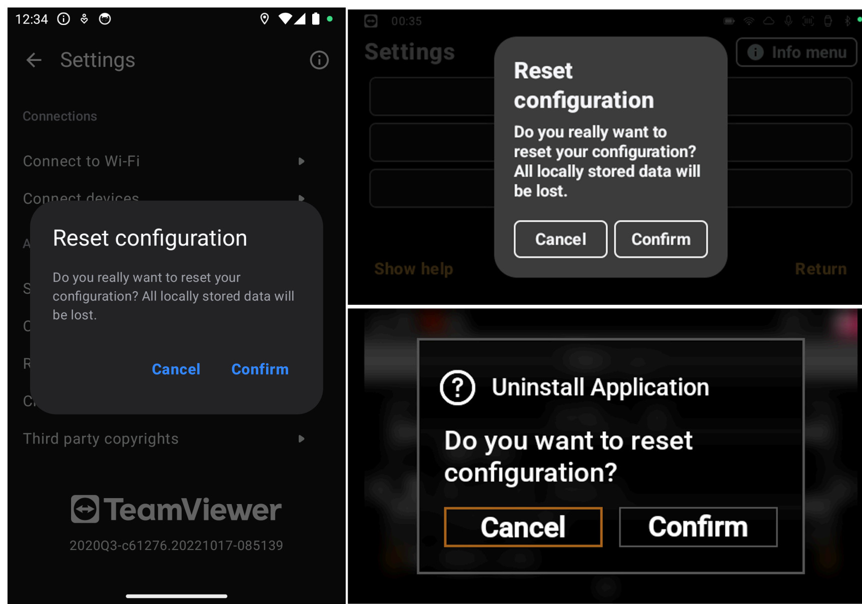


Figure 34. Reset Configuration Dialog on the Frontline Workplace app (bottom-right) and on the Picking Client app for smartphone (left) and smartglasses (top-right)

5.2.2 Workflow Comparison

As discussed in Chapter 3.1.1.2, the legacy Frontline UI system was based on a layered architecture that separated styling, layout, and content logic across multiple XML files. While this modularity enabled reusability, it also introduced substantial complexity. Developers and PEs were required to manage styles, part templates, layout pages, and layout models independently (Figure 35). Consequently, even minor modifications to the UI became time-consuming and error-prone.

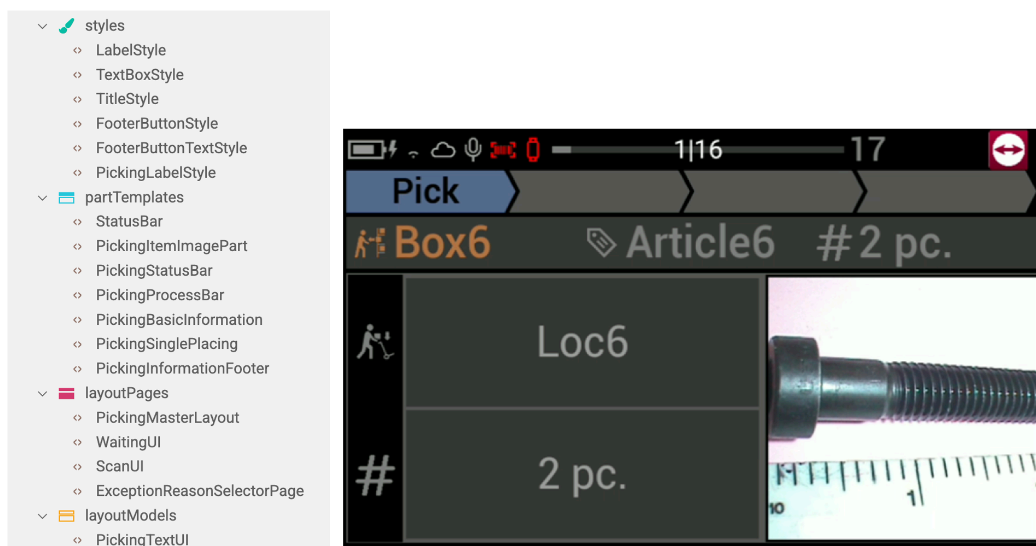


Figure 35. Files required to define the UI shown in the workflow on the right

The unified schema addresses these challenges by consolidating UI definitions into a single XML structure. Instead of coordinating multiple interdependent files, developers now describe entire screens covering structure, content, and behavior in one place (Figure 36). This not only reduces development overhead but also enhances maintainability and ensures consistency across workflows.

```xml
<LayoutPage name="Generic_Picking">
        <Header>
            <PickingProgressBar name="picking_progress_bar" orderId="${pickOrder.id}" />
        </Header>
        <MainContent>
            <Panel weight="1">
                <Image name="article_image" content="${pickOrder.groups[activeGroup].source.image}" />
            </Panel>
            <Panel weight="2">
                <InfoCard content="${pickOrder.groups[activeGroup].source.pick_location}"  label="Pick Location" style="highlighted" icon="RES_arrow_circle_up" />
                <Grid orientation="horizontal">
                    <InfoCard weight="2" content="${pickOrder.groups[activeGroup].source.article_code}"  label="Article Number" icon="RES_tag" />
                    <InfoCard weight="1" content="${pickOrder.groups[activeGroup].source.pick_amount}"  label="Amount" icon="RES_number_symbol_square" />
                </Grid>
                <InfoCard  content="${pickOrder.groups[activeGroup].source.deposit_location}"  label="Place Location" icon="RES_arrow_circle_down" />
            </Panel>
        </MainContent>
        <Footer>
            <DefaultFooter>
                <Button type="help" name="show_help" text="Show Help" />
                <Button type="help" name="select_exceptions" text="Select Exception" />
                <Button type="action" name="pick_action" />
            </DefaultFooter>
        </Footer>
    </LayoutPage>
```

Figure 36. UI definition of the workflow shown in Figure 37

In the legacy approach, creating a screen involved:

1. a layout page to define structure,
2. a layout model to populate it, and
3. several part templates and styles to control appearance.

This fragmentation often led to duplication and made it difficult to trace how a UI was assembled. By contrast, the unified schema enables developers to define all aspects of the UI within a single framework (Figure 37). Both LayoutPages and NativePages play a role in this system: LayoutPages offer the flexibility to create bespoke designs, while NativePages provide predefined templates for recurring interaction patterns. Together, they create a more streamlined and scalable development process.

Figure 37 illustrates the user interface rendered from the UI definition presented in Figure 36. This replicates a similar layout to the one shown in Figure 35, but consolidated into a single file while still enabling flexibility in adapting both the content and its presentation across different devices running the workflow.

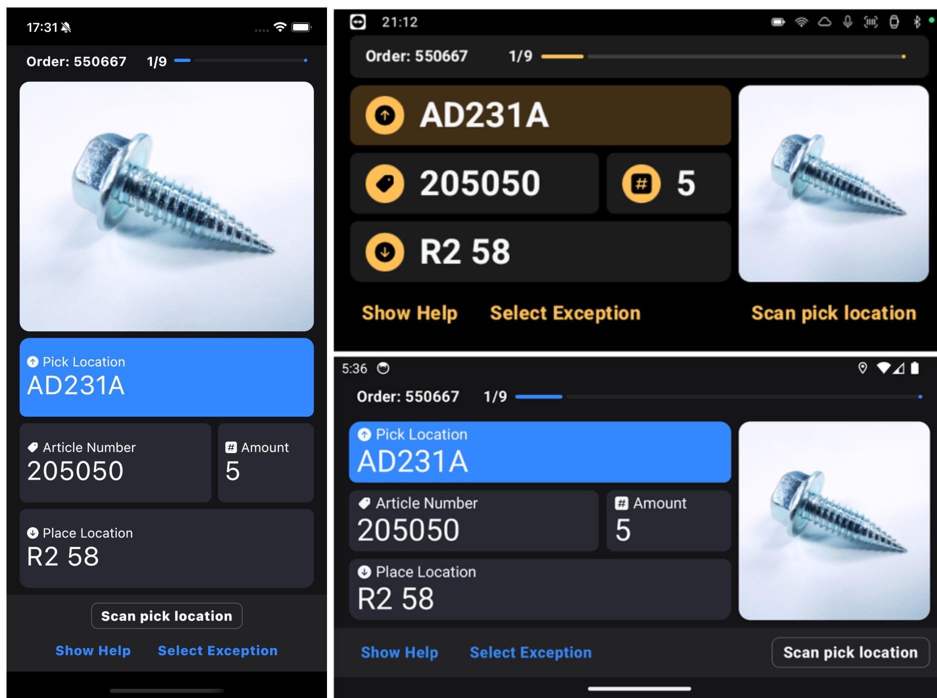


Figure 37. Workflow example running on iOS (left), smartglasses (top right), and Android phones (bottom right)

Another important improvement is the way the new schema manages limited screen space. When the available display area is insufficient to present all desired content simultaneously, the interface can be divided into panels. These panels can then be presented either as scrollable content (a natural fit on smartphones) or as paged views, where each panel is shown individually, and the user navigates between them sequentially (Figure 38). This ensures that workflows remain functional and accessible even on devices with constrained screen sizes.

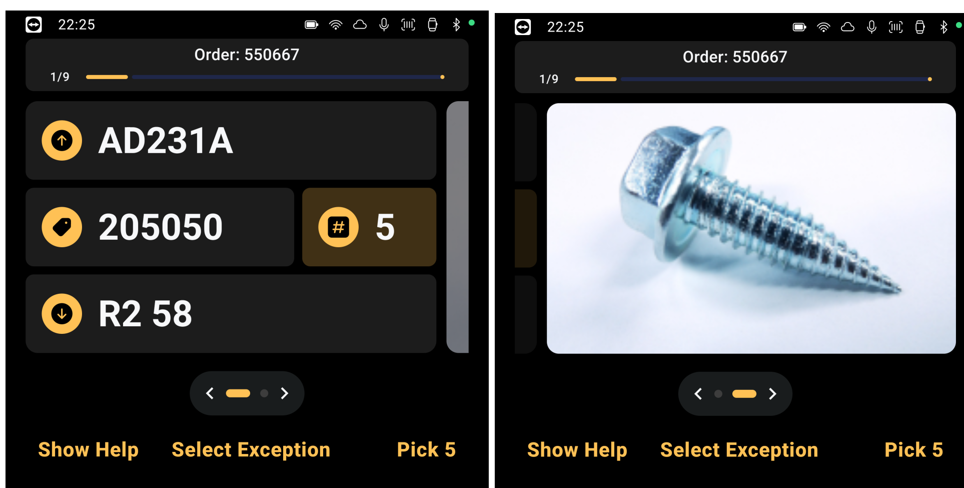


Figure 38. Example of paged content navigation

The comparison with the legacy architecture demonstrates that the unified schema effectively addresses the shortcomings of the previous system. The validation of the unified schema can

be discussed across several dimensions that correspond to the main challenges of the legacy approach.

5.2.2.1 Maintainability

One of the persistent issues in the legacy system was the high maintenance effort caused by the distribution of styles, templates, and layout definitions across multiple files. The unified schema consolidates these elements into a single source of truth. Changes to shared components, such as headers, buttons, or footers, are automatically applied across all workflows. This reduces the likelihood of inconsistencies and confirms that the new approach lowers the overall maintenance burden.

5.2.2.2 Structural Clarity

The unified schema simplifies the development process by combining styles, templates, layout models, and content definitions into a single XML structure. This eliminates unnecessary fragmentation, reduces the risk of configuration errors, and improves readability. As a result, workflow definitions are easier to understand, maintain, and extend compared to the legacy approach.

5.2.2.3 Cross-Device Adaptability

Testing across smartphones and smartglasses shows that workflows render correctly on a diverse set of devices. Responsive layout attributes ensure that screens adapt to different resolutions and aspect ratios, maintaining both usability and visual consistency. This validates the schema’s ability to support the heterogeneous device landscape required in frontline operations.

5.2.2.4 Reusability

The new system eliminates the need to repeatedly define styling and structure for each workflow by introducing pre-styled, adaptive components and standardized templates. Elements such as buttons, dialogs, picking cards, lists are now set and ready to be used on workflows. This approach significantly reduces duplication, accelerates workflow creation, and ensures consistent visual and functional behavior across devices.

5.2.2.5 Customizability

In addition to standardization, the schema preserves flexibility by allowing developers to combine predefined templates (NativePages) with fully customized layouts (LayoutPages). This balance ensures that workflows can be developed efficiently while still supporting unique requirements when needed. The validation confirms that the unified schema achieves both standardization and customization.

5.3 Feedback Validation

This chapter examines the user experience validation of the Multi-platform Adaptive UI System. The validation form was developed to gather critical feedback from a diverse group of

stakeholders to ensure the system meets the project's goals of improving adaptability and streamlining the UI creation process for interactive systems. Conducted during the first half of August, the form aimed to assess the system's performance across devices such as Android smartphones, iPhones, and smartglasses, identifying strengths and areas for refinement. This evaluation is essential to validate the redesigned client interface and the new, more efficient method for declaring workflows, ensuring the system aligns with the needs of developers, designers, and operational teams.

5.3.1 Demographic

A total of 21 participants answered a google form (Figure 39) essential for evaluating the Multi-platform Adaptive UI System, including 5 QAs (23.8%), 10 Developers (47.6%), 4 PEs (19%), and 2 UI/UX Designers (9.5%). This group tested the system during the first half of August to assess its usability and redesign of the client interface, as well as to validate a more efficient method for declaring UI workflows, core objectives of the project aimed at enhancing multi-platform adaptability in interactive systems. The demographic's significant Developer representation (47.6%) ensures robust feedback on implementation and optimization, a critical aspect of UI declaration efficiency, while Project Experts and QA professionals provide valuable operational and quality assurance perspectives, enhancing the system's practical applicability across these devices. The inclusion of UI/UX Designers, though limited to 9.5%, offers essential design insights for the client redesign, contributing to a balanced evaluation.

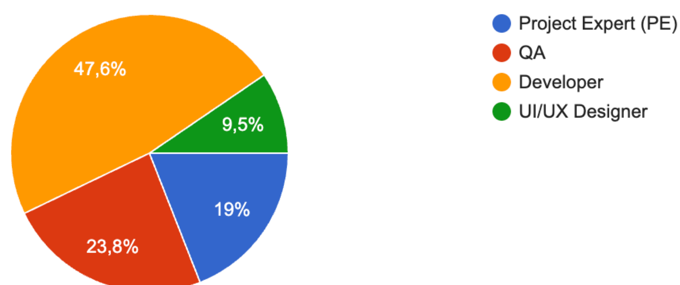


Figure 39. Demographic from the feedback form

Experience with Frontline varied (Figure 40), with 38.1% having over one year, 23.8% over five years, 19% between six and twelve months, 9.5% less than six months, and 9.5% with no experience. This distribution brings depth to the evaluation, with the majority (61.9%) possessing at least one year of experience ensuring informed critiques, though the minimal novice presence (9.5%) may slightly underrepresent entry-level usability challenges.

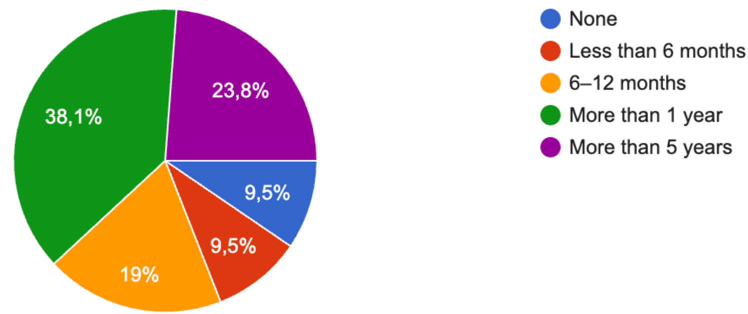


Figure 40. Level of experience with Frontline on the feedback form

The diverse device usage, Android smartphones, iPhones, and smartglasses, mirrors the project's multi-platform focus, as depicted in Figure 41, and is well-suited for testing adaptability across varied screen sizes and interaction methods. The inclusion of smartglasses by a subset of participants highlights their relevance for validating emerging technologies, though it also underscores the need for targeted performance assessment to address potential limitations on these devices.

Figure 41. Devices used for the feedback form

5.3.2 Methodology

The validation form was designed to evaluate multiple aspects of the Multi-platform Adaptive UI System, including usability, adaptability, performance, efficiency, and overall satisfaction. Participants responded to each statement on a 5-point Likert scale [74] (1 = strongly disagree, 5 = strongly agree) and provided open feedback to supplement the numeric ratings. The results are organized into application-related and workflow-related aspects, with both quantitative outcomes and qualitative interpretations discussed.

5.3.3 Application related feedback

The first area of evaluation focused on the application itself, including its interface, behavior, and cross-device consistency. Participants responded to the following statements:

1. *“The UI was easy to understand and navigate.”*
2. *“Visual consistency across devices (Android, iOS, Smartglasses) was maintained.”*
3. *“Button mapping for navigation worked as expected.”*
4. *“Speech recognition responded accurately to my voice commands.”*
5. *“Screens loaded in a reasonable amount of time.”*
6. *“The app responded quickly to my actions (e.g. button presses, navigation).”*

5.3.3.1 Usability and Navigation

The application achieved an average usability and navigation rating of **4.7/5**, showing that users found the interface coherent and easy to follow. The consistent structure across screens allowed participants to quickly understand the app’s flow, making it straightforward to locate features and move between different sections.

Open feedback highlighted appreciation for the adaptation to different devices, particularly smartphones and smartglasses. The Settings page (Figure 20) was frequently mentioned as an example, where the layout and interaction model adjust depending on the device type. This effort to tailor the experience reinforced usability, ensuring smooth and intuitive navigation across platforms.

5.3.3.2 Visual Consistency

Most participants rated the application as either **Consistent (42.9%)** or **Very Consistent (38.1%)**, indicating strong cross-platform visual integrity across Android smartphones, iPhones, and smartglasses. A smaller portion (19%) found it **Neutral**, while no participants reported it as **Very Somewhat Inconsistent** or **Inconsistent**. These results suggest that the current implementation effectively maintains a coherent layout and style, enhancing confidence and usability when switching devices.

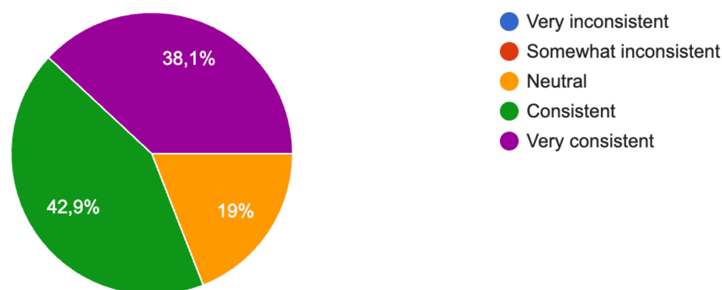


Figure 42. Visual Consistency chart from the feedback form

5.3.3.3 Button Mapping

Button mapping received an average rating of **4.4**, with participants highlighting the *app_config.yaml* approach as a major advantage. This configuration-based method allowed keys to be mapped without code changes, making the process simple and highly adaptable across devices. One example came from RealWear smartglasses, which use some custom key events; with this approach, these keys were easily assigned to actions demonstrating strong flexibility for heterogeneous hardware environments.

Despite the positive feedback, some participants reported occasional inconsistencies. In one example, pressing NEXT was expected to move to the following element but instead focused on a different one. These cases were rare but highlight the need for further refinement of focus management to ensure predictable navigation across all supported devices.

5.3.3.4 Speech Recognition

Speech recognition was rated **4.3**, the lowest among application-related metrics. Participants noted sporadic delays or missed commands, especially on smartglasses. Potential causes include background noise, hardware limitations, or insufficient recognition thresholds. Suggested improvements include adaptive signal processing, distinct voice command notifications, and enhanced feedback mechanisms to confirm successful recognition.

5.3.3.5 Performance

Participants rated the application's performance highly, with response speed with **4.6** and screen load times with **4.7**. Feedback emphasized that the app felt smooth and responsive, with quick reactions to user input and minimal waiting when moving between screens. On smartglasses, some users noticed occasional slowdowns, which is expected given their more limited hardware, but the application still maintained overall stability and usability.

These impressions are consistent with the results obtained in the earlier benchmarks (Chapter 5.2), which also showed improvements over the Workplace app. The alignment between feedback and measurements reinforces that the Picking Client offers a more efficient and reliable experience across devices, while remaining well-suited for both smartphones and smartglasses.

5.3.4 Workflow related feedback

The second area of evaluation focused on workflow related aspects, particularly the system for creating adaptive workflows using NativePages and LayoutPages. Participants rated the following statements:

1. *"The new system is faster to configure UIs."*

2. *“The new system is more reusable (layouts/components).”*
3. *“The new system adapts to different devices well.”*
4. *“Workflow screens loaded correctly without layout issues.”*
5. *“I like the new design system and feel it supports workflows well in real use scenarios.”*
6. *“Overall, the new system improves the workflow creation process.”*

5.3.4.1 Configuration Speed

Configuration speed received an average rating of **4.6**, indicating that participants found the new declaration method efficient. Open feedback emphasized that workflow creation is now substantially faster, especially for experienced users. The presence of prebuilt templates and streamlined UI definitions contributes to these efficiency gains.

Although the Creator tool does not currently support the new layout schema, resulting in the absence of features such as autocomplete, participants were still able to configure interfaces efficiently by relying solely on the accompanying documentation. Achieving a high rating under these conditions demonstrates the inherent clarity and usability of the proposed layout approach, even in the absence of integrated tooling support.

5.3.4.2 Reusability of Layouts/Components

Reusability received a rating of **4.3**, indicating that the system effectively supports component reuse, but with reduced customization compared to the previous implementation. This limitation reflects a deliberate design trade-off: prioritizing simplicity and efficiency over extensive configurability. While the current approach may not offer the same level of flexibility, it significantly streamlines the workflow creation process, reducing complexity for most use cases.

Frontline provides a generic picking workflow template that serves as a base for PEs to construct the desired workflows for customers. This template significantly reduces development effort by offering a predefined base that typically requires only minor adjustments. Almost every scenario from this template is supported by the existing components and NativePages, ensuring broad applicability without extensive customization.

5.3.4.3 Device Adaptability

Device adaptability scored **4.4**, indicating that workflows consistently rendered and behaved correctly across smartphones and smartglasses. Participants highlighted that layouts adjusted appropriately to different screen sizes, aspect ratios, and interaction models, ensuring usability regardless of device type. This confirms that the adaptive design principles implemented in the system effectively support heterogeneous hardware environments, maintaining both functional integrity and visual consistency across platforms.

5.3.4.4 Workflow Accuracy

Workflow accuracy reached **95%**, confirming that screens rendered correctly without layout issues in most cases. This result demonstrates the reliability of the unified schema in ensuring consistent and predictable workflow presentation across devices.

5.3.5 Open feedback

Open-ended feedback provided further insights into how participants experienced the workflow system. Developers, in particular, praised NativePages for significantly reducing the time required to define workflows. Because these participants were not always deeply familiar with workflow intricacies, they valued the simplicity and immediacy of creating UIs that "just work" without needing extensive customization. Project Experts also recognized the benefit of having both NativePages and LayoutPages: NativePages are efficient when a predefined template meets the workflow's needs, while LayoutPages offer additional flexibility for more complex or specialized interfaces. This combination was highlighted as a strong design choice, allowing the system to balance speed and adaptability depending on the context.

The Workflow UI Schema's documentation was widely regarded as extensive, clear, and well-structured, providing a solid foundation for understanding the workflow UI definition. Some participants suggested that its effectiveness could be further enhanced by including practical examples, templates, and sample configurations. Such additions would likely accelerate onboarding for new users and provide guidance for implementing less common or more advanced workflows.

Participants also noted a few areas where improvements could enhance overall usability. These included differentiating voice command notifications from standard UI notifications to prevent overlap, expanding the library of available UI components, and adding more NativePages to cover a broader range of workflow scenarios. Collectively, these suggestions point to opportunities for fine-tuning the system, ensuring it remains both efficient for typical workflows and flexible enough for advanced or specialized cases.

6 Conclusions

This chapter presents the final reflections on the work developed throughout this thesis. It begins by summarizing the main contributions of the proposed multi-platform adaptive UI system, highlighting its relevance in addressing the challenges identified in the initial stages of the project. The chapter then evaluates the extent to which the defined goals were achieved, providing a clear link between the objectives and the implemented solution. Following this, the limitations encountered during the development and validation phases are discussed, offering a critical perspective on areas that require further improvement. Finally, the chapter explores potential directions for future work, outlining enhancements that could extend the system's capabilities and applicability. These considerations aim to provide a foundation for continued research and development in the field of adaptive, cross-platform user interfaces.

6.1 Contributions

The contributions of this work extend across academic research, industrial application, and organizational impact. From an academic perspective, this thesis introduces and validates an architecture for adaptive multi-platform user interfaces, leveraging Kotlin Multiplatform and Compose Multiplatform. This architecture addresses the inefficiencies identified in Chapter 1 by reducing complexity in UI definition, ensuring styling consistency, and introducing mechanisms for reusability across workflows. In addition, the work establishes a set of best practices for multi-platform UI development, including strategies for modular layout design, adaptive interaction handling, and performance optimization for resource-constrained devices. These guidelines provide a methodological foundation that can inform future research and development in cross-platform UI engineering.

From an industrial standpoint, the project delivers a functional and reusable UI module capable of adaptive rendering and modular component reuse. This module significantly reduces workflow creation time, improves maintainability, and integrates seamlessly into existing systems. Its design ensures scalability and adaptability for heterogeneous devices, including

smartphones and smartglasses. Furthermore, the dissertation introduces the new workflow layout schema for Frontline, consolidating layout and content definitions into a single structure. This innovation replaces the fragmented, multi-file approach of the legacy system and combines LayoutPages for flexibility with NativePages for rapid development, achieving a balance between customization and efficiency while improving maintainability and reducing redundancy.

The organizational impact of this work is demonstrated by its adoption beyond the scope of the initial project. An unexpected yet highly significant outcome was the integration of the developed UI module by other TeamViewer teams during the course of this dissertation. This adoption highlights the practical relevance and scalability of the solution, as several teams expressed interest in using the smartglasses UI implementation to modernize existing Android-based applications. Delivered as a Kotlin Multiplatform library, the module integrates seamlessly with Android code without requiring a complete rewrite, reducing migration costs and accelerating time-to-market. This interoperability reinforces the architectural robustness and adaptability of the system, confirming its long-term value for the organization.

6.2 Achieved Goals

The objectives defined in Chapter 1.3 were successfully achieved, confirming the alignment between the initial research goals and the outcomes delivered.

The first objective, **developing a multi-platform solution** (Goal 1.3.1), was realized through the integration of Kotlin Multiplatform and Compose Multiplatform, as described in Sections 4.1 and 4.2. This approach enabled the sharing of business logic and user interface components across heterogeneous devices, including smartphones and head-mounted displays. It significantly reduced code duplication and streamlined the development process while preserving the flexibility required for device-specific adaptations. Performance benchmarks presented in Chapter 5.1 further validate that the solution maintains competitive startup times, efficient resource usage, and overall responsiveness across platforms.

The second objective, **simplifying the workflow UI definition process** (Goal 1.3.2), was one of the most critical challenges addressed by this work. The new schema replaced the fragmented, multi-file approach of the legacy system with a centralized structure that unifies layout, content, and styling definitions. This innovation reduces redundancy, improves clarity, and accelerates workflow creation, as demonstrated in the comparative evaluation in Chapter 5.2.2.

The third objective, **enabling adaptive and modular layouts** (Goal 1.3.3), was achieved through the introduction of a unified layout schema that supports both LayoutPages and NativePages (Chapter 4.4.2). This schema consolidates layout and content logic into a single structure, eliminating the need for developers to manually define styling and layout elements for each workflow. As a result, visual consistency is ensured, configuration effort is minimized, and workflows can adapt seamlessly to different devices without requiring multiple versions for

various form factors. Validation results in Chapter 5.2.2 confirm that this adaptability significantly improves maintainability and scalability.

The fourth objective, **integrating advanced interaction methods** (Goal 1.3.4), was successfully accomplished through the implementation of voice commands and hardware navigation, as detailed in Chapter 4.5. These features enable hands-free operation and alternative input methods, enhancing usability in industrial environments where traditional touch interaction is often impractical. Feedback from user validation (Chapter 5.3) highlights the positive impact of these features on operational efficiency in contexts characterized by physical constraints and high mobility.

Finally, the fifth objective, **establishing best practices for multi-platform development** (Goal 1.3.5), was achieved through the definition of guidelines for modular UI design and adaptive interaction handling (Sections 4.2 and 4.5). These practices not only guided the implementation of this solution but also provide a methodological foundation for future projects within and beyond the organization.

All objectives outlined in Chapter 1.3 were achieved. The resulting system not only meets the functional and technical requirements but also introduces innovations that improve workflow creation efficiency, cross-device adaptability, and user experience, while laying the groundwork for future developments in multi-platform UI systems.

6.3 Limitations

Although the proposed solution achieved its primary objectives, some limitations remain that highlight opportunities for improvement. On iOS, Kotlin Multiplatform introduces additional overhead due to the Skia rendering engine, which increases application size and memory usage (Chapter 5.1). This is a known limitation of KMP rather than the application itself. Despite this, the solution remains highly performant and fully usable in production. With the release of Compose Multiplatform 1.8.0 [75], iOS support is now stable, and future updates are expected to further optimize rendering and reduce overhead.

Performance on smartglasses, while generally very good, can be affected when rendering workflow pages with a high number of components, complex layouts, or frequent state changes that require constant UI updates. In such cases, slight slowdowns may occur (Chapter 5.3.4), which can impact responsiveness. Further optimization of rendering and state management could help mitigate these occasional performance drops.

Finally, voice recognition, although functional, showed occasional inaccuracies in noisy industrial environments (Chapter 5.3.3.4). While this limitation is primarily due to the external voice recognition package provided by the Frontline platform, working closely with the team maintaining this package could significantly improve accuracy and robustness in future iterations.

6.4 Future Work and Potential Enhancements

Apart from the limitations identified in Chapter 6.3, future work should prioritize extending the system's flexibility for workflow design. A key improvement is the expansion of the workflow component library, allowing new elements to be directly integrated into the new layout schema. This would provide PEs with more options to design workflows that are both efficient and tailored to specific use cases, reducing the need for custom development. Establishing a continuous feedback loop with PEs will be essential to identify gaps in the current design and determine whether additional structural variations, such as headers and footers, are needed to improve navigation and usability.

Finally, the system should incorporate dedicated UI patterns for other wearable devices, which are increasingly prevalent in warehouse and logistics environments. While the current implementation supports smartglasses, extending the design to other wearables, such as wrist-mounted displays or rugged handhelds, would enhance the system's applicability and ensure consistent user experiences across all device types used in industrial operations.

6.5 Final Remarks

This dissertation set out to address the challenges of UI fragmentation, device heterogeneity, and workflow complexity in industrial environments. Through the integration of Kotlin Multiplatform and Compose Multiplatform, it introduced a unified approach to building adaptive, multi-platform user interfaces that significantly improves development efficiency and ensures a consistent user experience across diverse devices.

A central contribution of this work is the new workflow UI schema for Frontline, which consolidates layout and content definitions into a single structure and introduces reusable components. This innovation simplifies workflow creation, enhances maintainability, and enables seamless adaptability across smartphones, smartglasses, and other platforms. Together, these advancements provide a scalable and future-ready foundation for industrial applications.

The practical relevance of the solution is reinforced by its adoption within the organization, demonstrating its value beyond the scope of this research. It not only delivers a functional system but also establishes best practices and design principles that can guide future developments in multi-platform UI engineering.

Ultimately, this work positions the proposed approach as a robust and flexible framework capable of supporting the evolving demands of industrial software, paving the way for continued innovation in adaptive user interface systems.

References

[1] “TeamViewer remote control software: Secure device access anywhere.” Accessed: Dec. 16, 2024. [Online]. Available: https://www.teamviewer.com/en/solutions/use-cases/remote-control/

[2] “TeamViewer Frontline.” Accessed: Dec. 16, 2024. [Online]. Available: https://www.teamviewer.com/en/products/frontline/

[3] “Frontline Help & Learning.” Accessed: Dec. 09, 2024. [Online]. Available: https://www.teamviewer.com/en/global/support/knowledge-base/teamviewer-frontline/frontline-help-learning/

[4] “TeamViewer Customer Success story: Samsung SDS.” Accessed: Dec. 16, 2024. [Online]. Available: https://www.teamviewer.com/en/success-stories/samsung-sds/

[5] “Frontline - Creator - Knowledge Base.” Accessed: Dec. 16, 2024. [Online]. Available: https://www.teamviewer.com/en/global/support/knowledge-base/teamviewer-frontline/frontline-command-center/creator/

[6] “Frontline - Actions - Knowledge Base.” Accessed: Dec. 16, 2024. [Online]. Available: https://www.teamviewer.com/en/global/support/knowledge-base/teamviewer-frontline/developer-guide-workflow-engine-reference/actions/

[7] “Frontline - Transitions - Knowledge Base.” Accessed: Dec. 16, 2024. [Online]. Available: https://www.teamviewer.com/en/global/support/knowledge-base/teamviewer-frontline/developer-guide-workflow-engine-reference/actions/transitions/

[8] “Frontline - Layouts and Mapping - Knowledge Base.” Accessed: Dec. 16, 2024. [Online]. Available: https://www.teamviewer.com/en/global/support/knowledge-base/teamviewer-frontline/developer-guide-creator-developer-training/layouts-and-mapping/

[9] “Frontline - UI Update - Knowledge Base.” Accessed: Dec. 16, 2024. [Online]. Available: https://www.teamviewer.com/en/global/support/knowledge-base/teamviewer-frontline/developer-guide-workflow-engine-reference/actions/ui/

[10] “Frontline - UI Concept - Knowledge Base.” Accessed: Dec. 17, 2024. [Online]. Available: https://www.teamviewer.com/en/global/support/knowledge-

base/teamviewer-frontline/developer-guide-workflow-engine-reference/ui/ui-concept/

[11] O. D. Segun-Falade, O. S. Osundare, W. E. Kedi, P. A. Okeleke, T. I. Ijomah, and O. Y. Abdul-Azeez, "Developing crossplatform software applications to enhance compatibility across devices and systems," *Computer Science & IT Research Journal*, vol. 5, no. 8, pp. 2040–2061, Aug. 2024, doi: 10.51594/CSITRJ.V5I8.1491.

[12] S. Liao-Mäkinen, "Developing a full stack mobile application," 2023, Accessed: Dec. 11, 2024. [Online]. Available: http://www.theseus.fi/handle/10024/795514

[13] A. Biørn-Hansen, T. M. Grønli, G. Ghinea, and S. Alouneh, "An Empirical Study of Cross-Platform Mobile Development in Industry," *Wirel Commun Mob Comput*, vol. 2019, no. 1, p. 5743892, Jan. 2019, doi: 10.1155/2019/5743892.

[14] A. Holzinger, P. Treitler, and W. Slany, "Making Apps Useable on Multiple Different Mobile Platforms: On Interoperability for Business Application Development on Smartphones," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7465 LNCS, pp. 176–189, 2012, doi: 10.1007/978-3-642-32498-7_14.

[15] A. Biørn-Hansen, C. Rieger, T. M. Grønli, T. A. Majchrzak, and G. Ghinea, "An empirical investigation of performance overhead in cross-platform mobile development frameworks," *Empir Softw Eng*, vol. 25, no. 4, pp. 2997–3040, Jul. 2020, doi: 10.1007/S10664-020-09827-6.

[16] S. Mascetti, M. Ducci, N. Cantù, P. Pecis, and D. Ahmetovic, "Developing Accessible Mobile Applications with Cross-Platform Development Frameworks", doi: 10.48550/ARXIV.2005.06875.

[17] M. Martinez and S. Lecomte, "Towards the Quality Improvement of Cross-Platform Mobile Applications," *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 184–188, Jul. 2017, doi: 10.1109/MOBILESOFT.2017.30.

[18] C. Wu, J. M. Pérez-Álvarez, A. Mos, and J. M. Carroll, "Codeless App Development: Evaluating A Cloud-Native Domain-Specific Functions Approach", doi: 10.48550/ARXIV.2210.01647.

[19] L. Albesher, R. Aldossari, and R. Alfayez, "An Observational Study on React Native (RN) Questions on Stack Overflow (SO)," *IET Software*, vol. 2023, no. 1, 2023, doi: 10.1049/2023/6613434.

[20] L. Albesher, R. Aldossari, and R. Alfayez, “An Observational Study on React Native (RN) Questions on Stack Overflow (SO),” *IET Software*, vol. 2023, no. 1, 2023, doi: 10.1049/2023/6613434.

[21] K. Wasilewski and W. Zabierowski, “A comparison of java, flutter and kotlin/native technologies for sensor data-driven applications,” *Sensors*, vol. 21, no. 10, May 2021, doi: 10.3390/S21103324.

[22] “Flutter vs React Native in 2024: End-To-End Comparison.” Accessed: Nov. 28, 2024. [Online]. Available: https://markovate.com/blog/flutter-vs-react-native/

[23] V. NUTALAPATI, “Concept to Completion - Android Apps and Kotlin Multi Platform,” 2024, doi: 10.61909/AMKEDTB082431.

[24] “KMP - flexible multiplatform development (part 1) - Application Engineering.” Accessed: Nov. 28, 2024. [Online]. Available: https://blog.nashtechglobal.com/kmp-flexible-multiplatform-development-part-1/

[25] I. E. KHO, A. W. ALIYAZIS, and M. GALINIUM, “Front-End Application for Multiple Storefronts Ecommerce Using Cross-Platform Technology,” *Business Excellence and Management*, vol. 12, no. 1, pp. 93–104, Mar. 2022, doi: 10.24818/BEMAN/2022.12.1-07.

[26] E. GÜLCÜOĞLU, A. B. USTUN, and N. SEYHAN, “Comparison of Flutter and React Native Platforms,” *Journal of Internet Applications and Management*, Dec. 2021, doi: 10.34231/IUYD.888243.

[27] Y. KOZUB and H. KOZUB, “Features of Multiplatform Application Development on Kotlin,” *Herald of Khmelnytskyi National University. Technical Sciences*, vol. 317, no. 1, pp. 224–229, Feb. 2023, doi: 10.31891/2307-5732-2023-317-1-224-229.

[28] B. Suri, S. Taneja, I. Bhanot, H. Sharma, and A. Raj, “Cross-Platform Empirical Analysis of Mobile Application Development frameworks: Kotlin, React Native and Flutter,” *Proceedings of the 4th International Conference on Information Management &Amp; Machine Intelligence*, Dec. 2022, doi: 10.1145/3590837.3590897.

[29] D. Wheeler and J. I. Olszewska, “Cross-Platform Mobile Application Development for Smart Services,” *2022 IEEE 22nd International Symposium on Computational Intelligence and Informatics and 8th IEEE International Conference on R*, pp. 203–208, 2022, doi: 10.1109/CINTI-MACRO57952.2022.10029466.

[30] A. Nedyak, O. Rudzeyt, A. Zainetdinov, and P. Ragulin, “Mobile cross-platform app development tools,” *Russian Journal of Resources, Conservation and Recycling*, vol. 7, no. 4, Dec. 2020, doi: 10.15862/13INOR420.

[31] J. Bieniek, M. Rahouti, and D. C. Verma, “Generative AI in Multimodal User Interfaces: Trends, Challenges, and Cross-Platform Adaptability,” 2024.

[32] M. Guśpiel, “Mobile App Development Using Kotlin Multiplatform,” 2024, Accessed: Dec. 10, 2024. [Online]. Available: http://www.theseus.fi/handle/10024/852201

[33] “FAQ | Kotlin Multiplatform Development Documentation.” Accessed: Dec. 10, 2024. [Online]. Available: https://www.jetbrains.com/help/kotlin-multiplatform-dev/faq.html#kotlin-multiplatform

[34] “Compose Multiplatform UI Framework | JetBrains.” Accessed: Dec. 10, 2024. [Online]. Available: https://www.jetbrains.com/compose-multiplatform/

[35] “Android & iOS native vs. Flutter vs. Compose Multiplatform.” Accessed: Dec. 10, 2024. [Online]. Available: https://www.jacobras.nl/2023/09/android-ios-native-flutter-compose-kmp/

[36] “Expected and actual declarations | Kotlin Documentation.” Accessed: Dec. 10, 2024. [Online]. Available: https://kotlinlang.org/docs/multiplatform-expect-actual.html

[37] “Write automated tests with UI Automator | Android Developers.” Accessed: Dec. 15, 2024. [Online]. Available: https://developer.android.com/training/testing/other-components/ui-automator

[38] G. Fantini, “Continuous integration for End-to-End testing of mobile applications,” Apr. 2022.

[39] “Espresso | Android Developers.” Accessed: Dec. 15, 2024. [Online]. Available: https://developer.android.com/training/testing/espresso

[40] “Intro to Appium - Appium Documentation.” Accessed: Dec. 15, 2024. [Online]. Available: https://appium.io/docs/en/2.0/intro/

[41] “What is an Android Emulator? - GeeksforGeeks.” Accessed: Dec. 15, 2024. [Online]. Available: https://www.geeksforgeeks.org/what-is-an-android-emulator/

[42] “Emulator Vs Simulator Vs Real Device: Mobile Testing.” Accessed: Dec. 15, 2024. [Online]. Available: https://www.accelq.com/blog/emulator-vs-simulator-vs-real-device/

[43] “What is Mobile Application Testing? - GeeksforGeeks.” Accessed: Dec. 15, 2024. [Online]. Available: https://www.geeksforgeeks.org/what-is-mobile-application-testing/

[44] A. Biørn-Hansen, C. Rieger, T. M. Grønli, T. A. Majchrzak, and G. Ghinea, “An empirical investigation of performance overhead in cross-platform mobile development frameworks,” *Empir Softw Eng*, vol. 25, no. 4, pp. 2997–3040, Jul. 2020, doi: 10.1007/S10664-020-09827-6/TABLES/26.

[45] K. Wasilewski and W. Zabierowski, “A Comparison of Java, Flutter and Kotlin/Native Technologies for Sensor Data-Driven Applications,” *Sensors 2021, Vol. 21, Page 3324*, vol. 21, no. 10, p. 3324, May 2021, doi: 10.3390/S21103324.

[46] V. Vidmark, “Performance Evaluation of Kotlin Multiplatform on iOS,” vol. TRITA – EECS-EX.

[47] J. R. Lewis and J. Sauro, “Usability and User Experience: Design and Evaluation,” *Handbook of Human Factors and Ergonomics*, pp. 972–1015, Aug. 2021, doi: 10.1002/9781119636113.CH38.

[48] M. Hertzum, “Usability Testing,” 2020, doi: 10.1007/978-3-031-02227-2.

[49] G. Joyce, M. Lilley, T. Barker, and A. Jefferies, “Mobile application usability: Heuristic evaluation and evaluation of heuristics,” *Advances in Intelligent Systems and Computing*, vol. 492, pp. 77–86, 2016, doi: 10.1007/978-3-319-41935-0_8.

[50] “State and Jetpack Compose | Android Developers.” Accessed: Sep. 04, 2025. [Online]. Available: https://developer.android.com/develop/ui/compose/state

[51] “NavHost | API reference | Android Developers.” Accessed: Aug. 06, 2025. [Online]. Available: https://developer.android.com/reference/androidx/navigation/NavHost

[52] “Compose Multiplatform – Beautiful UIs Everywhere.” Accessed: Sep. 04, 2025. [Online]. Available: https://www.jetbrains.com/compose-multiplatform/

[53] “Jetpack Compose | Android Developers.” Accessed: Aug. 06, 2025. [Online]. Available: https://developer.android.com/develop/ui/compose/components/scaffold?hl=pt-br

[54] “WS50 Programmer’s Guide - TechDocs.” Accessed: Sep. 02, 2025. [Online]. Available: https://techdocs.zebra.com/emdk-for-android/13-0/guide/ws50_programming/

[55] "InteractionSource | API reference | Android Developers." Accessed: Aug. 24, 2025. [Online]. Available: https://developer.android.com/reference/kotlin/androidx/compose/foundation/interaction/InteractionSource

[56] "Compose modifiers | Jetpack Compose | Android Developers." Accessed: Sep. 04, 2025. [Online]. Available: https://developer.android.com/develop/ui/compose/modifiers

[57] "Side sheets – Material Design 3." Accessed: Aug. 24, 2025. [Online]. Available: https://m3.material.io/components/side-sheets/overview

[58] "Navigation drawer – Material Design 3." Accessed: Aug. 06, 2025. [Online]. Available: https://m3.material.io/components/navigation-drawer/overview

[59] "Snackbar | Jetpack Compose | Android Developers." Accessed: Aug. 06, 2025. [Online]. Available: https://developer.android.com/develop/ui/compose/components/snackbar?hl=pt-br

[60] "CameraX overview | Android media | Android Developers." Accessed: Sep. 04, 2025. [Online]. Available: https://developer.android.com/media/camera/camerax

[61] "ML Kit | Google for Developers." Accessed: Sep. 04, 2025. [Online]. Available: https://developers.google.com/ml-kit?hl=pt-br

[62] "PreviewView | API reference | Android Developers." Accessed: Sep. 04, 2025. [Online]. Available: https://developer.android.com/reference/androidx/camera/view/PreviewView

[63] "Permissions on Android | Privacy | Android Developers." Accessed: Sep. 04, 2025. [Online]. Available: https://developer.android.com/guide/topics/permissions/overview

[64] "AVFoundation | Apple Developer Documentation." Accessed: Sep. 04, 2025. [Online]. Available: https://developer.apple.com/documentation/avfoundation/

[65] "AVCaptureSession | Apple Developer Documentation." Accessed: Sep. 04, 2025. [Online]. Available: https://developer.apple.com/documentation/avfoundation/avcapturesession

[66] "AVCaptureMetadataOutput | Apple Developer Documentation." Accessed: Sep. 04, 2025. [Online]. Available: https://developer.apple.com/documentation/avfoundation/avcapturemetadataoutput

[67] "AVMetadataObject | Apple Developer Documentation." Accessed: Sep. 04, 2025. [Online]. Available: https://developer.apple.com/documentation/avfoundation/avmetadataobject

[68] “UIView | Apple Developer Documentation.” Accessed: Sep. 04, 2025. [Online]. Available: https://developer.apple.com/documentation/uikit/uiview

[69] “AVCaptureDevice | Apple Developer Documentation.” Accessed: Sep. 04, 2025. [Online]. Available: https://developer.apple.com/documentation/avfoundation/avcapturedevice

[70] “SharedFlow | kotlinx.coroutines – Kotlin Programming Language.” Accessed: Aug. 06, 2025. [Online]. Available: https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-shared-flow/#

[71] “MutableInteractionSource | API reference | Android Developers.” Accessed: Sep. 08, 2025. [Online]. Available: https://developer.android.com/reference/kotlin/androidx/compose/foundation/interaction/MutableInteractionSource

[72] “Profile your app performance | Android Studio | Android Developers.” Accessed: Sep. 05, 2025. [Online]. Available: https://developer.android.com/studio/profile

[73] “Instruments Tutorials | Apple Developer Documentation.” Accessed: Sep. 05, 2025. [Online]. Available: https://developer.apple.com/tutorials/instruments

[74] A. Joshi, S. Kale, S. Chandel, and D. Pal, “Likert Scale: Explored and Explained,” *Br J Appl Sci Technol*, vol. 7, no. 4, pp. 396–403, Jan. 2015, doi: 10.9734/BJAST/2015/14975.

[75] “Compose Multiplatform 1.8.0 Released: Compose Multiplatform for iOS Is Stable and Production-Ready | The Kotlin Blog.” Accessed: Sep. 07, 2025. [Online]. Available: https://blog.jetbrains.com/kotlin/2025/05/compose-multiplatform-1-8-0-released-compose-multiplatform-for-ios-is-stable-and-production-ready/

Appendix A. Project Risks

Risk ID	Description	Cause	Effect	Risk Owner	Probability (1-5)	Impact (1-5)	PI Score	Expected Result, No Action	Risk Response Type	Response Description
R1	Delayed Project Completion	Complex customization requirements	Missed deadlines, stakeholder dissatisfaction	Project Manager	3	4	12	Extended development timelines impacting delivery dates	Mitigation	Perform detailed requirement analysis, use Agile Scrum to address complexity in iterations, and involve stakeholders in regular progress reviews.
R2	Inconsistent Styling	Manual application of styles across workflows	Increased maintenance effort and user confusion	UI Designer	4	3	12	Lack of a cohesive UI leading to poor user experience	Mitigation	Introduce a centralized design system and use modular styling components to ensure consistency across workflows.
R3	Device Compatibility Issues	Lack of dynamic UI adaptation for devices	Poor user experience, increased development effort	Tech Lead	4	5	20	Workflows need extensive rework for different device types	Mitigation	Implement adaptive layouts using Compose Multiplatform to handle diverse screen

Risk ID	Description	Cause	Effect	Risk Owner	Probability (1-5)	Impact (1-5)	PI Score	Expected Result, No Action	Risk Response Type	Response Description
										sizes and interaction methods.
R4	Limited Reusability of Components	Workflow-specific shared layouts and components	Duplication of efforts, higher development costs	Development Lead	3	4	12	Repetition of UI elements across workflows	Mitigation	Enable reuse of components across workflows by enhancing shared resource handling in the platform.
R5	Technology Limitations	Immaturity of Kotlin Multiplatform	Compatibility or performance issues on some devices	Technical Architect	3	5	15	Unreliable application performance, increased debugging time	Mitigation	Conduct extensive testing, use stable releases of Kotlin Multiplatform, and involve the Kotlin community for support.
R6	Gaps in Product Knowledge	Limited understanding of platform capabilities	Inefficient feature usage and suboptimal design	Product Owner	4	4	16	Ineffective workflows and wasted development effort	Mitigation	Provide comprehensive product training, create detailed documentation, and schedule knowledge-sharing sessions with

Risk ID	Description	Cause	Effect	Risk Owner	Probability (1-5)	Impact (1-5)	PI Score	Expected Result, No Action	Risk Response Type	Response Description
										experienced team members.
R7	Misaligned User Expectations	Lack of understanding of end-user workflows	Poor adoption and user dissatisfaction	UX Researcher	3	4	12	Low user engagement and increased support requirements	Mitigation	Conduct user research sessions, gather feedback, and adjust workflows to align with actual user needs and preferences.
R8	Stakeholder Misalignment	Differing priorities among stakeholders	Conflicting requirements, delayed approvals	Project Manager	3	4	12	Unclear scope and scope creep	Mitigation	Schedule regular stakeholder meetings to align priorities, document agreements, and ensure transparency in progress updates.
R9	Limited Platform Knowledge	Lack of familiarity with new features	Delayed implementation and poor feature integration	Development Lead	4	4	16	Incomplete feature set and inefficient workflows	Mitigation	Establish regular team sync-ups, provide access to detailed feature guides, and assign exploratory tasks

Risk ID	Description	Cause	Effect	Risk Owner	Probability (1-5)	Impact (1-5)	PI Score	Expected Result, No Action	Risk Response Type	Response Description
										for new platform capabilities.
R10	Workflow Scalability Issues	High complexity in workflow configurations	System performance degradation with larger workflows	System Architect	3	5	15	Reduced system performance affecting usability	Mitigation	Optimize backend architecture, introduce performance testing for large-scale workflows, and implement caching where applicable.

Appendix B. Project Planning

Gantt diagram to represent all tasks that will be performed during the project.