Final Project Report: Development and Architectural Evolution of the Ros2BridgeMobile Application

Winston Har - November 16 2025

1.0 Executive Summary

This report provides a comprehensive overview of the progress and strategic evolution of the Ros2BridgeMobile project over the last six months. The primary objective of this initiative is to deliver a robust, user-friendly Android application for interfacing with and controlling ROS 2 robots via the rosbridge WebSocket protocol.

To go into more detail the application supports, at the highest level, a connection between a physical or virtual controller to Ryan's ROS 2 environment. The application allows for the creation of not only generic ROS messages, actions and services, but also customized ones as well. This application is meant to connect via an open lan port on the robot.

The project's journey was split into two distinct phases. Phase 1 resulted in the creation of TestRos2JSBridge (V1), a fully-featured prototype that successfully validated the core concept of mobile-based ROS 2 interaction. It provided complete functionality, including controller mapping, custom message publishing, and topic subscription. However, its monolithic architecture, centered around a single ViewModel(or arguably 2) and reliance on SharedPreferences for all data persistence, presented significant long-term challenges related to scalability, maintainability, and testability. This is also a fundamental issue with the way messages are imported into the codebase leading to a lack of modularity.

Recognizing these limitations as a barrier to future growth, Phase 2 was initiated: a complete architectural overhaul resulting in Ros2BridgeMobile (V2). The key achievement of this period was the strategic migration to a modern, layered architecture utilizing industry-standard libraries such as Room for database persistence, Dagger Hilt for dependency injection, and Jetpack Compose for the UI. This re-architecting effort has established a resilient and scalable foundation for the application.

Currently, the V2 architecture is successfully in place, with foundational modules and several core features, such as connection management and custom action creation, fully implemented. While a temporary and planned gap in feature parity with V1 exists, the new architecture ensures that the remaining features can be implemented more efficiently, reliably, and with a higher degree of quality. This report details the technical decisions, challenges, and strategic direction of this critical transformation. Supplemental materials: kotlin_file_documentation.md for version 2, and Repository_Architecture_Details.md for version 1 outline the exact feature discrepancy and project structure to allow for easier future development.

2.0 Introduction & Project Goals

2.1 Problem Statement

The ROS 2 ecosystem, while powerful, often necessitates interaction through command-line tools or complex desktop applications like RViz. This paradigm poses challenges for robotics engineers, researchers, developers, and operators who require mobility and ease of use in real-world environments. Simple tasks like robot teleoperation, on-the-fly diagnostics, or demonstrating robot capabilities to non-technical stakeholders(buyers) are hindered by the need for a tethered laptop. This project addresses the need for a mobile-first solution that democratizes ROS 2 interaction, providing an intuitive and powerful interface directly on an Android device. Theoretically parts of this codebase can be transferred to a standalone controller for simpler controls.

2.2 Initial Project Objectives

The project was chartered with a clear set of initial objectives to create a minimal viable product that would be genuinely useful:

1. Establish Stable Communication: Implement a reliable, bidirectional communication channel with a ROS 2 rosbridge_server, handling connection, disconnection, and error states gracefully.
2. Support Diverse Messaging: Enable the publishing and subscribing of messages to ROS 2 topics, with support for both standard message types (e.g., geometry_msgs/msg/Twist) and user-defined custom message types (messages for pose server).
3. Implement Real-Time Control: Integrate external gamepad/controller input(via bluetooth) for intuitive robot teleoperation, including logic for joystick deadzones, sensitivity, and mapping to appropriate geometry messages.
4. Provide User Customization: Build functionality for users to create, save, and manage custom message publishing configurations ("AppActions") and assign them to specific controller buttons or UI elements. Allow for export/import of profiles for quick deployment.

3.0 Phase 1: Initial Implementation (V1 - TestRos2JSBridge)

https://github.com/WinstonHar/Ros2BridgeMobile

3.1 Architecture Overview

TestRos2JSBridge was developed as a pragmatic, feature-focused application. The architecture was straightforward, built as a standard Android Studio project with Kotlin.

- Centralized Logic (RosViewModel): The application's brain was a single, large RosViewModel. This component was responsible for the entire ROS 2 communication lifecycle: managing the WebSocket connection, advertising topics, serializing and publishing messages, and processing incoming subscriptions. All fragments and UI components directly observed and interacted with this single ViewModel.

- Traditional UI Framework: The user interface was constructed using the classic Android UI toolkit(xml), consisting of Fragments to represent different screens (e.g., ControllerSupportFragment, CustomPublisherFragment) and RecyclerViews to display lists of topics and protocols.
- Key-Value Data Persistence (SharedPreferences): All application state and user-generated content, including saved IP addresses, controller presets, and complex button-to-action mappings, were serialized into JSON strings and stored in SharedPreferences. This approach was simple to implement, works, but lacked the structure and integrity of a true database.

3.2 Key Accomplishments & Features
Despite its architectural simplicity, Phase 1 was a critical success, delivering a proof-of-concept that was not only functional but feature-complete according to the initial objectives. The application supported a wide array of features, including preset controller schemes (to allow for multiple remaps), dynamic creation of custom publishers, and live monitoring of ROS 2 topics. This version unequivocally proved the viability and value of a mobile ROS 2 interface and provided invaluable insights that informed the decision to evolve the architecture.

4.0 Phase 1 Evaluation & Motivation for Architectural Redesign

The success of V1 also highlighted its limitations. As a prototype, its architecture was not designed for long-term growth, and several critical issues became apparent.

- 4.1 Scalability and Maintainability Challenges:
    - Brittle Data Persistence: SharedPreferences proved to be the architecture's primary weakness. To store a controller configuration, the entire object graph, including nested button maps and action details, had to be manually serialized into a single JSON string. This approach offered no data integrity, type safety, or querying capabilities. Modifying any data model required careful, error-prone manual updates to the serialization and deserialization logic.
    - Monolithic ViewModel: The centralized RosViewModel became a significant bottleneck. With dozens of responsibilities, it grew to thousands of lines of code, making it exceedingly difficult to debug and maintain(although having 10 200 line files is arguably more difficult to debug in my opinion). Any change, no matter how small, carried the risk of creating unintended side effects in unrelated features, requiring a full manual regression test of the entire application.
- 4.2 UI/UX Limitations: The imperative nature of the traditional Android UI toolkit, coupled with the single ViewModel, made it difficult to manage complex UI state. This resulted in a user experience that was functional but lacked the polish, reactivity, and responsiveness of modern applications.
- 4.3 Lack of Testability: The architecture's tight coupling made automated testing nearly impossible. Components had hard dependencies on the Android OS context, and the monolithic ViewModel could not be instantiated in isolation for unit testing. This reliance on manual testing was time-consuming and unreliable.

- 4.4 Custom Message Support: One of the key issues with the design for V1 was how the ros message creation lifecycle was hardcoded into the viewmodels. This means that if someone wanted to add support for new messages, or modify existing messages, during the creation process, they'd have to dive into the app codebase. This needs to be fixed via dynamic message import support (via adding ros message files to assets folder in V2 current implementation).
- 4.5 Strategic Decision: These challenges made it clear that continuing to build upon the V1 foundation would be inefficient and lead to a fragile, unmaintainable product. The strategic decision was made to invest in a complete architectural redesign to ensure the project's long-term viability, extensibility, and quality.

5.0 Phase 2: Architectural Overhaul (V2 - Ros2BridgeMobile)

https://github.com/WinstonHar/Ros2BridgeMobile/tree/master

Phase 2 represents a fundamental shift from a prototype to a production-quality application, rebuilt from the ground up using a modern, scalable architecture.

- 5.1 Core Architectural Principles: The new architecture is strictly layered, enforcing a clear separation of concerns as seen in the project's file structure:
    - Presentation Layer: Contains all UI-related logic (Jetpack Compose screens, ViewModels) and is responsible for displaying data and capturing user input.
    - Domain Layer: Contains core business logic, models, and use cases, and is independent of any specific framework or data source.
    - Data Layer: Responsible for all data operations, abstracting data sources (network and local database) through the Repository pattern.
- 5.2 Data Persistence Layer Modernization: The cornerstone of the new architecture is the replacement of SharedPreferences with a Room Database. A comprehensive relational schema was designed, as detailed in the provided diagrams(last page).
    - Relational Entities: Tables like AppAction, ButtonMap, Controller, and ButtonPresets now exist as type-safe Room entities.
    - Data Integrity: The relationships between these entities are now enforced at the database level. For example, the diagrams show how a Controller owns ButtonPresets, which in turn own collections of ButtonMaps, each linking to a specific AppAction. This structure guarantees data integrity and allows for powerful, efficient querying.
    - DAOs (Data Access Objects): Each entity is paired with a DAO (e.g., AppActionDao, ControllerDao), which provides a clean, abstract interface for all database operations.
- 5.3 Backend and Networking:
    - Repository Pattern: The application now uses repositories (e.g., AppActionRepositoryImpl, ControllerRepositoryImpl) as the single source of truth for data. These repositories abstract away the complexities of whether data is coming from the local Room database or the remote rosbridge server.

    - Dependency Injection (Dagger Hilt): Hilt is used to manage object lifecycles and provide dependencies throughout the application. This decouples components from their concrete implementations, dramatically improving modularity and making the codebase easily testable.
    - Structured Networking: Network logic is now encapsulated within dedicated classes like RosbridgeClient and ConnectionManager, which are provided as singletons by Hilt.
    - Dynamic message support: Instead of having ros message types defined in the app codebase all messages for the creation lifecycle process are dynamically created. All standard libraries for ros exist as packages, which along with custom message packets, can be put in a designated folder in the codebase, and then generated into the UI. This allows for support for any message, action or service, via just adding more packages and a recompile.
- 5.4 UI/UX Modernization: The entire UI has been rewritten in Jetpack Compose. This declarative UI toolkit allows for the creation of a better, responsive, and maintainable user interface with significantly less boilerplate code. Reusable components like RosConnectionCard and TopicSelector have been created to ensure a consistent look and feel.
- 5.5 State Management: The application now uses a modern, reactive approach to state management. Each screen has its own dedicated ViewModel which exposes UI state via Kotlin's StateFlow. UI components observe this flow and automatically update when the state changes. Asynchronous operations, like network requests or database queries, are safely handled in the background using structured concurrency with Kotlin Coroutines.

6.0 Project Status & Feature Implementation Comparison

6.1 V2 Feature Status

The architectural migration is a significant undertaking, and progress is being made systematically.

- Completed Features: The foundational elements are in place. The Connection screen (ConnectionScreen.kt and ConnectionViewModel.kt) successfully manages connection state and user input. The Publisher screen allows for the creation, editing, and deletion of custom AppActions, which are correctly persisted in the Room database. The Controller screen framework is built, supporting the creation and selection of different configurations.
- Features In Progress: Development was focused on re-implementing the more complex, interactive features. The Subscriber screen, which requires a robust real-time data flow from the WebSocket through the entire stack to the UI, is a top priority. Advanced controller preset management (removing and updating presets) and the YAML-based import/export functionality are also under active development.

6.2 V1 vs. V2 Functionality Gap

Currently there is a gap in the functionality between V1 and V2. This centers around the complexity of how I decided to implement the navigation between the different compose view

screens. As a quick explainer there was no way to sync the data between two separate screens on the nav bar which led to the data creation in one screen not being viewable in a different screen. As the screens rely on each other's data, this created a black box basically causing troubleshooting the database to be a massive problem, which was overcome with sheer development time. This trade-off, sacrificing short-term feature parity for long-term architectural integrity, is essential for a more maintainable application. The new foundation will dramatically accelerate possible future development, allowing this gap to be closed and surpassed with a higher standard of quality and stability.

7.0 Challenges Encountered & Solutions

- Technical Challenge 1: Data Model Migration: The most complex task was translating the unstructured, conceptual data model from V1 into a normalized, type-safe relational schema for the Room database. This was overcome by dedicating significant time to data modeling and diagramming upfront, ensuring the database structure was sound before writing any implementation code(as well as a lot of trial and errors).
- Technical Challenge 2: Mastering a New Technology Stack: The transition involved a steep learning curve, requiring the team to master the paradigm shifts associated with Jetpack Compose (declarative UI), Dagger Hilt (dependency injection), and the structured concurrency of Kotlin Flows. This was addressed through prototyping, thorough documentation review, and iterative implementation. The original design had a lot of unnecessary boilerplate which was left behind and iterated upon during the development process.
- Project Management Challenge: Balancing Refactoring and Feature Parity: The primary project management challenge was to maintain focus on building the new architecture correctly, resisting the temptation to take shortcuts to port features more quickly. This was managed by adopting a phased approach, prioritizing foundational modules first (database, DI, networking) before moving on to feature-specific modules.

8.0 Future Work & Recommendations

- 8.1 Immediate Next Steps:
    1. Complete Subscriber Functionality: Prioritize the full implementation of the Subscriber screen, including the SubscriberViewModel, to allow for real-time topic subscription and message history display. The easiest way to do this would be importing the subscriber logic in fragments over to the new DI/DB structure from the old V1 to V2 (this might seem easy but it's trivial to fall into hours or days of troubleshooting due to incorrectly linked logic).
    2. Implement Import/Export Logic: Develop the logic within the ConfigurationRepositoryImpl to handle the serialization and deserialization of controller configurations to and from YAML files, fixing the known issue. This was not implemented yet due to the database structure of the application not being finalized, for controller preset (below).

3. Finalize Controller Preset Management: Implement the "remove" and "update" logic for controller presets, completing the CRUD (Create, Read, Update, Delete) operations for controller configuration. Currently the controller preset logic does not seem to work correctly for the controller configurations, this feature needs a lot of close debugging and tracing to determine the real source of the DB sync issue (what I was doing for the last few weeks, still didn't manage to determine the source of the issue though)

- 8.2 Long-Term Roadmap:
    1. Develop a Comprehensive Test Suite: Leverage the new testable architecture by writing a full suite of tests, including local unit tests for ViewModels and Repositories, and instrumented tests for Room DAOs and Jetpack Compose components.
    2. Establish a CI/CD Pipeline: Configure a continuous integration/continuous deployment pipeline using a tool like GitHub Actions to automate the process of building, running tests, and deploying new versions of the application.
    3. Expand ROS 2 Feature Support: Extend the application to support ROS 2 services and actions, building upon the foundational work already in place.

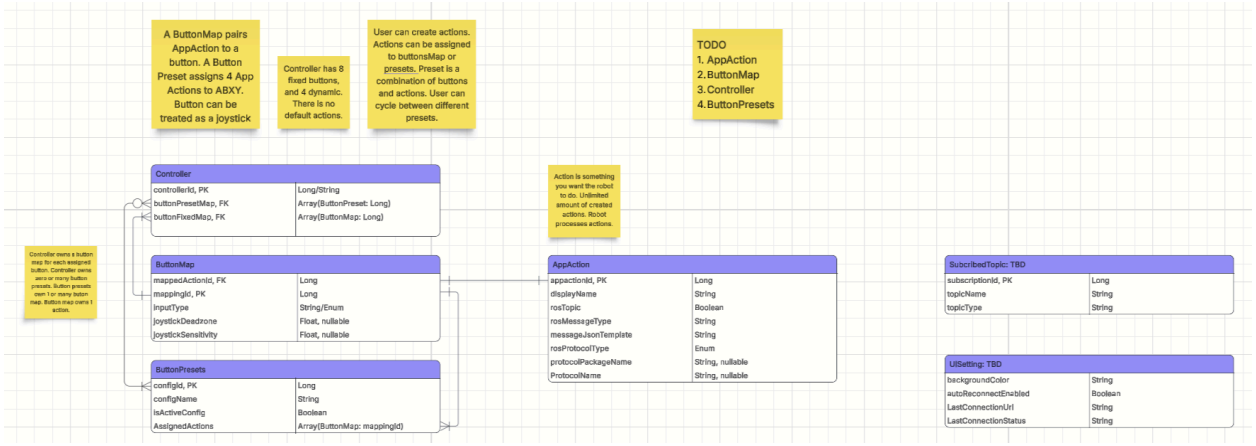
9.0 Conclusion

The last six months have marked a period of profound and positive transformation for the Ros2BridgeMobile project. It has matured from a monolithic proof-of-concept into a modular, enterprise-grade application built on a modern and sustainable foundation. The strategic decision to invest in a complete architectural overhaul has successfully addressed the critical limitations of the initial prototype, paving the way for future growth. The project is now equipped with a resilient backend, a reactive UI, and a testable codebase. While work remains to achieve full feature parity with the original version, the current trajectory is strong, and the project is exceptionally well-positioned to become a solid tool for mobile interaction within the ROS 2 ecosystem.

I definitely learned a lot about project management, full stack planning for a development process, and time management. I would encourage a team instead of a single person to be assigned for a project of this scope if further development is intended. The sheer number of files, 94 to be exact, is very imposing and challenging to manage with one person at the helm. With a team the modular parts of the codebase can be assigned to specific members to allow for parallel development (and troubleshooting), as well as easier discussions between members to reduce the amount of time wasted trying to implement features in a working but not optimal manner.

10.0 Assets

10.1 V2 ERD



10.2 V2 Database flow diagram