Politecnico di Torino

Politecnico di Torino

Master's Degree in Ingegneria Informatica (Computer Engineering)

A.y. 2025/2026

Graduation Session July 2026

Modernizing the PoliTO Mobile Ecosystem

A Unified DevOps Strategy for CI/CD Optimization and Custom Over-The-Air Updates

Supervisors:

Marco Torchiano

Cristina Ferrian

Federico Cucinella

Candidate:

Emanuele Coricciati

Summary

In the current mobile ecosystem of the Politecnico di Torino, the introduction of the new Faculty application alongside the existing Students app presents significant operational challenges. Maintaining an increasing number of applications running in the same technical environment raises the risk of increased costs and complexity due to long build times, unstable Continuous Integration (CI) pipelines, and duplicate code and configurations. This thesis explores a design-based transformation of the current codebase into a monorepo architecture with an Over-The-Air (OTA) update model. The first part of the thesis involves an architectural consolidation phase where the distinction between shared logic and application-specific code is formalized. In addition to creating a solid basis for eliminating duplicated complexity from expanding application portfolios, a well-defined separation and a reusable common set of components and services are created. The second part of the thesis addresses the introduction of a dynamic distribution system through the OTA update process. The dynamic distribution system enables instant deployment of non-native changes, such as bundles and assets, which bypasses traditional app store release processes while providing robust release channels and automated roll back mechanisms. Finally, the existing Continuous Integration and Continuous Delivery (CI/CD) pipeline based on GitHub Actions is transformed to reflect the new monorepo architecture. As a result of the transformation, the system will be able to determine whether infrastructure changes require complete rebuilds or if they only require publishing bundles. The effectiveness of this method will be verified through a qualitative and quantitative impact assessment. Comparison metrics are based on lines of code saved, release velocity, OTA update costs, and the total amount of time spent on unnecessary builds before and after implementation.

Acknowledgements

First off, a huge thank you to my supervisor, Marco Torchiano, and ISIAD — especially Cristina Ferrian and Federico Cucinella — for giving me the chance to work on this project, and for all their incredible help and support along the way. I also want to thank my family, my brother, and my friends for their support, for keeping me motivated, and for being there for me every step of the way.

Table of Contents

List of Figures

Glossary

ISIAD
: Infrastrutture Servizi Informatici e Amministrazione Digitale

OTA
: Over-The-Air

UI
: User Interface

API
: Application Programming Interface

CI/CD
: Continuous Integration and Continuous Delivery

QA
: Quality Assurance

POI
: Point of Interest

APK
: Android Package Kit

CLI
: Command Line Interface

SaaS
: Software as a Service

EAS

Expo Application Services

MAU

Monthly Active Users

LoC

Lines of Code

Chapter 1

Introduction

1.1 Context

Mobile Apps are now considered the most common way students and faculty can interface with the institution's overall ecosystem. In the educational setting, the need for rapid access to numerous services - ranging from class scheduling to administration - has created a large number of mobile-based services. Due to cross-platform development, React Native has established itself as one of the current standard for developing applications at a high performance level from one shared code base. Despite the ever-growing number of special purpose apps, the core technical problem of engineering sustainable and operationally efficient mobile ecosystems has shifted from the initial design phase to long term sustainably.

At Politecnico di Torino, the digital ecosystem is experiencing its largest transformation. The Student App, developed together with Infrastrutture Servizi Informatici e Amministrazione Digitale (ISIAD), represents the principal point of contact for students. It provides fundamental services such as course management, a personal calendar, and campus navigation. ISIAD acts as the university's central IT support unit responsible for managing this infrastructure; currently supporting approximately 38,000 active users and having released software 13 times in the last year. In order to provide additional digital services, a new Faculty App was launched to address the unique administrative and instructional needs of faculty members. Although both applications utilize the same React Native technology stack and communicate with exactly the same back-end systems, the two applications have historically been treated as completely separate entities. This polyrepo treatment resulted in numerous serious bottlenecks.

Both applications rely upon the same Design System; therefore, the underlying User Interface (UI) code had to be replicated across different repositories, thereby significantly increasing the potential for divergence in configurations. When a

developer desired to modify a core component of one application, they would typically manually replicate and synchronize those modifications in both projects — clearly an error prone practice. These redundancies extended well beyond basic buttons and styles to include complex functional components. An excellent example of this is the Campus Map. With spatial data visualization capabilities and Point of Interest (POI) management, the map is a sizeable, self-sufficient module. Regardless if you are a student or a teacher utilizing the University's campus map, there are no variations in how the campus layout is arranged; consequently, duplicating this engine in two separate codebases was inherently wasteful. Redundancy similarly affected the developmental life-cycle. Both projects utilized the same tool chains used to manage their respective codebases; therefore, identical pre-commit hooks for TypeScript static type-checks, linting, and formatting existed independently in each repository. Updating validation logic required hand updating of each repository individually. Additionally, managing third party libraries - particularly native modules in the React Native ecosystem - was a major obstacle. Repositories for isolated projects tended to diverge regarding the versions of dependencies; accordingly, sustaining native iOS and Android files was complicated. Framework upgrades –which generally require substantial effort on the native side – were doubly burdensome and threatened to produce inconsistent deployment methodologies.

Furthermore, as described above, the rigidity of the deployment methodology caused problems. Traditional binary deployments via the Apple App Store and Google Play Store introduce considerable latency. Even minor non-native updates - such as UI tweaking or emergency JavaScript bug fixes - necessitated full recompilation followed by extensive review processes in stores; therefore, greatly hindering the development teams' ability to respond quickly. At an infrastructure level, separate, duplicate build cycles were generated by continuous integration/continuous deployment (CI/CD) pipelines based on GitHub Actions; therefore, creating a highly cumbersome task to maintain synchronization.

The ultimate goal of this work is to detail the restructuring of PoliTO's mobile ecosystem to resolve these overlapping challenges. The work describes transitioning from a fragmented polyrepo structure to a unified monorepo using npm workspaces to significantly reduce duplicated code. Concurrently with this structural transition, the project introduces a revised CI/CD pipeline and implements a custom OTA update server hosted locally. The updated infrastructure dynamically directs updates - prompting either a native deployment or immediate OTA delivery based on what was modified in the codebase. Lastly, the dissertation evaluates this architectural transition through evaluating qualitative enhancements in developer workflow along with quantifiable reductions in pipeline build time. Together, these metrics illustrate substantial reductions in maintenance overhead.

The new repository is available on GitHub as an open-source project for further

reference. [1]

1.2 Goal

The primary objective of this thesis is the architectural and operational optimization of the Politecnico di Torino's mobile ecosystem, specifically targeting the integrated management of the official Students and Faculty applications. This ecosystem will transform from separate development initiatives into a single, scalable and fault-tolerant application. This work also demonstrates how leveraging the React Native technology stack along with contemporary DevOps best-practices can significantly improve the cost-effectiveness of maintaining the Mobile Application and remove roadblocks that exist today in deploying new versions of the Mobile Applications.

Three coordinated interventions are developed to reach these goals:

- **Architectural Consolidation:** Establishing formal separation between shared logic and application-specific code . Through formal establishment of defined boundaries and a reusable common-base (components, services and modules) it is possible to remove redundant complexity due to duplicated functionality when expanding the number of supported applications.
- **Dynamic Distribution:** Developing an OTA update flow to publish bundles and assets in near-real-time to the Mobile Applications. This allows for non-native updates to be deployed outside of the traditional app-store release-cycle. Additionally, release channels and automatic-rollback mechanisms are introduced.
- **CI/CD Optimization:** Revising GitHub Actions workflows so they reflect a Monorepo structure. This enables the workflow automation to recognize whether changes made by developers require rebuilding the native infrastructure binaries or if the changes made are simply a logical update which can be published as a bundle. Also, this revised workflow includes additional Preview Channels to allow developers and designers to isolate their testing of specific OTA Deployments before releasing them to production.

1.3 Structure of the thesis

To provide a comprehensive view of the work performed, this thesis is organized into seven distinct chapters. Each section details a fundamental phase of the project,

[1] Repository URL: `https://github.com/polito/rn-apps`

from the initial architectural analysis to the practical deployment of the new mobile infrastructure. The content is structured as follows:

Chapter 2, Background and State of the Art: This chapter offers a general view on technologies used. It compares monorepo versus multi-repo architectures and examines the development of OTA Distribution within the React Native framework. It look at continuous integration and continuous deployment pipelines today in mobile workflows; also current digital ecosystem of Politecnico di Torino is evaluated for technical gap that this work tries to closing.

Chapter 3, Architectural Consolidation (The Monorepo): This chapter describes the core restructuring of the codebase. It details the design and implementation of the monorepo environment by focusing on the logic division between shared modules and application specific logic to ensure scalability and reusability.

Chapter 4, Custom Over-the-Air (OTA) Distribution Infrastructure: This chapter presents the design and deployment of the self-hosted server engineered to deliver application updates. It details the implementation of the Expo Updates protocol and the mechanics of handling runtime bundles and managing channels, branches, rollback directives to guarantee absolute data sovereignty and institutional control over asset distribution.

Chapter 5, Context-Aware Pipeline Redesign and Mobile DevOps Automation: This section show how the CI/CD flows have been redisegned using GitHub Actions. It's focused on optimizing code quality checkpoints and native fingerprinting utilities which are used to detect when task route towards native binary compile or ephemeral preview deployment.

Chapter 6, Evaluation and Impact Analysis: This section dedicates to discuss results through a complete comparison of "before" and "after" situations. The evaluation focuses on build time optimization, release cycles reliant on store decreased and global stability of unified infrastructure while capturing qualitative improvements in developer experience and efficiency of workflow.

Chapter 7, Conclusion and Future Developments: This final chapter summarizes results obtained by reflecting on the impact of unified management system for university services and identify potential futuristic extensions of the Politecnico's mobile ecosystem.

Chapter 2

Background and State of the Art

2.1 The React Native Framework

Among many mobile app development frameworks, React Native has become one of the most popular options for developing apps with high performance using a shared JavaScript/TypeScript codebase.

Unlike traditional hybrid solutions that rely on WebViews (rendering HTML/CSS inside a mobile container), React Native allows developers to write applications in JavaScript using the React component model while translating that code into native components. This ensures that the application interacts directly with the mobile OS, overcoming the performance limitations of older hybrid frameworks.

Recent versions of React Native have introduced a "New Architecture" written in C++, which includes the JavaScript Interface (JSI), Fabric renderer, and TurboModules, improving communication between JavaScript and native layers while reducing runtime overhead. These advancements further enhance React Native's ability to meet the needs of an institutional ecosystem looking to develop a responsive application rapidly and efficiently.

The main advantages of using this framework are:

- Cross-Platform Efficiency: A large portion of the codebase is reused across iOS and Android, reducing maintenance costs.
- Near-Native Performance: By using actual native UI elements, the framework provides a high-fidelity experience in terms of animations and scrolling.
- Fast Refresh: This feature allows developers to see code changes instantly on a device or simulator, significantly accelerating the development cycle.

- Modular Architecture: The component-based approach allows for breaking down complex applications into independent, reusable pieces.

A standard React Native project follows a structured hierarchy to manage both JavaScript logic and platform-specific native file:

```
MyApp/
├── android/  Android-specific native project files
├── ios/  iOS-specific native project files
├── node_modules/  JavaScript dependencies and libraries
├── App.js  Entry point component of the application
└── package.json  Project metadata, scripts, and dependencies
```

This standard folder structure with its distinct `/ios`, `/android`, and `/node_modules` directories—provides a clear boundary for a single mobile application. However, as software ecosystems scale, the development focus often shifts from managing a single codebase to coordinating multiple interdependent projects. This transition necessitates a strategic decision on how to organize the source code within version control systems.

2.2 Codebase Governance and Architectural Patterns

The long-term sustainability of multi-application mobile portfolios depends heavily on structural organization and operational consistency. As institutional digital frameworks expand, the choice of repository layout fundamentally shapes how engineering teams handle source control architectures and reconcile the evolving trade-offs between isolated and centralized development models. This strategic choice directly dictates the complexity of the underlying dependency graph, where the management of shared platform configurations, version alignment, and automated toolchains serves as the primary defense against technical debt and systemic architectural erosion.

2.2.1 Repository Management Strategies

As mobile ecosystems evolve from single applications to portfolios of interconnected applications, ecosystem coordination and long-term maintainability become key success factors.

Software ecosystems are coordinated collections of interdependent software projects requiring shared governance and evolution strategies [1]. Ecosystem governance and coordination are often reflected into how repositories are structured and governed. At a high level, repository management strategies generally follow either a monorepo or polyrepo model.

At its core, a Monorepo is a strategy where a single repository contains the source code for multiple projects [2]. For a React Native ecosystem, this might mean having one repository that houses a directory for the Students App, a directory for a second Faculty App, and a directory for shared UI Components.

In contrast, a Polyrepo (or multi-repo) approach assigns a dedicated repository to each individual project. Under this model, the Students App and the Faculty App would exist as entirely separate entities in the version control system, each with its own independent history and lifecycle.

A literature review [3] highlighted that the main benefits of Monorepo adoption include simplified dependencies, coordination of cross-project changes, and easy refactoring; although code health, codebase complexity, and tooling investments for both development and execution are important challenges.

The two alternative architectures encourage very different ways of thinking about software development:

- Holistic vs. Independent Management: A monorepo manages projects together holistically. Changes can be tracked, tested, and released across all apps simultaneously. A polyrepo manages projects separately; changes affect only one project at a time, ensuring they can be released independently without affecting the rest of the ecosystem.
- Conjoins vs. Contracts: Monorepo environments encourage developers to think about how projects "conjoin" or overlap. Polyrepos, by nature of their isolation, force developers to think in terms of "contracts"—the formal interfaces and Application Programming Interface (APIs) that allow separate repositories to communicate [4].
- Atomic State: In a monorepo, the current state of the entire ecosystem is captured by a single commit hash. In a polyrepo, the state of the ecosystem is a fragmented collection of commits across various repositories.

One of the most practical benefits of a monorepo is the ease with which components can share common code. When multiple React Native apps require the same authentication logic or branding components, a monorepo facilitates sharing without the overhead of external versioning. This approach is invaluable: if a developer needs to modify a function signature or an interface used across apps, they can perform a global search-and-replace to update the entire ecosystem in a single operation.

Furthermore, monorepos reduce "tribal knowledge." By having "THE" repository, the service and ownership graphs become incredibly clear, making code discovery easier compared to the "adventure" of searching through multiple cryptic tomes in a polyrepo setup.

While a monorepo simplifies coordination, it eventually hits a scaling limit. When an organization reaches large sizes (millions of lines of code and hundreds developers), standard Git workflows can become impractical. Problems arise in terms of space (the repository exceeds the storage of a developer's laptop) and time (operations like fetching or pruning become sluggish).

2.2.2 Dependency and Configuration Management

A central challenge in multi-application ecosystems concerns dependency management and configuration consistency. Modern JavaScript and TypeScript ecosystems rely heavily on extensive dependency graphs composed of both application-level and native platform libraries. In React Native environments, these dependencies are tightly coupled with Android and iOS native layers, making version synchronization particularly critical.

Duplicated configurations across distributed repositories progressively diverge over time, increasing maintenance effort and operational fragility. This phenomenon is especially problematic in polyrepo architectures where linting rules, TypeScript configurations, CI/CD workflows, and dependency versions must be manually synchronized across independent repositories.

Research on technical debt and architecture erosion further identifies dependency divergence and duplicated infrastructure as common indicators of long-term maintainability degradation [5].

To mitigate these issues and maintain the benefits of a unified repository, specialized tooling becomes necessary:

- npm/yarn Workspaces: The foundational mechanism for managing multi-package repositories. They enable cross-linking between local projects (e.g., sharing a UI library with the mobile apps) without the overhead of external package registries [6].
- Nx and Bazel: These systems track internal dependencies and ensure that only the parts of the code affected by a change are rebuilt or tested [7].
- Turborepo: A high-performance build system for JavaScript and TypeScript monorepos that utilizes advanced caching and task execution graphs to minimize redundant work and speed up CI/CD pipelines [8].
- Lerna: Specifically optimized for Node.js environments, it helps manage the workflow and versioning of multi-package repositories.

- Virtual File Systems (VFS): Advanced setups (like those used by Google or Microsoft) allow developers to work on a portion of the code locally without downloading the entire multi-gigabyte repository.

While academic literature on monorepo architectures provides foundational theoretical grounding, a substantial body of practitioner-oriented guidance has emerged from the React Native community itself. Notable examples include the react-native-monorepo-tools package [9] and the Expo monorepo documentation [10]. However, these resources share a common limitation: they are prescriptive in nature, offering configuration templates and recommended practices without empirical grounding in production deployments. They do not report measured outcomes nor do they document the architectural decisions and tradeoffs. The present work complements this practitioner knowledge base by providing evidence-backed documentation of a complete migration lifecycle — from problem identification through architectural design to quantitative evaluation — on two production-grade applications.

2.3 Dynamic Application Distribution and Over-the-Air (OTA) Infrastructures

Traditional deployment workflows for compiled mobile applications present a significant point of friction within fast-paced software engineering lifecycles. When code changes are made, developers must generate a complete platform-specific binary file, sign it with cryptographic certificates, and submit it to the Apple App Store and Google Play Store. This process forces the deployment to undergo vendor review phases that introduce unpredictable delays.

To overcome these delays and achieve continuous delivery, modern mobile engineering relies on dynamic distribution strategies. Over-the-Air (OTA) update models allow development teams to push application updates directly to users' devices. This cuts delivery times down from days to seconds, radically altering how critical patches, user interface adjustments, and business logic evolutions are distributed in production.

2.3.1 Architectural Prerequisites within the React Native Runtime

The implementation of an OTA framework is not universally applicable to all mobile development environments; it requires a specific architectural separation between the native platform wrapper and the execution engine. React Native

is suited for this model because it handles compilation and execution through a decoupled dual-layer structure.

A standard React Native application binary consists of a native platform shell (written in Java/Kotlin for Android and Objective-C/Swift for iOS) that embeds a specialized JavaScript runtime engine, such as Hermes. When the operating system initializes the application container, the native runtime layer boots up and immediately requests an underlying asset pack known as the JavaScript bundle. This bundle is a single minimized file that contains the entire scope of the application's business logic, component definitions, and static UI configurations.

Crucially, this architecture means that the application's dependency graph - housed within the `node_modules` directory - is also fundamentally split. While many JavaScript libraries exist entirely within the bundle layer, an extensive category of third-party dependencies includes native code bridges. Through the framework's autolinking mechanism, installing a node module that targets native capabilities causes the local build toolchain to automatically inject platform-specific source files, compile static libraries, and bind native APIs into the underlying iOS and Android project layers during binary compilation.

Because the native binary acts as an abstract rendering container executing instructions read from the decoupled JavaScript bundle, the user interface and logic can be modified dynamically at runtime. An updates client can intercept the boot sequence, check a remote server for a newly compiled bundle, swap the local asset references on the device's storage, and trigger an engine reload—all without requiring a full binary recompilation or forcing an operating system-level application update.

2.3.2 Operational Boundaries of Bundle-Isolated Updates

While OTA mechanics grant immense deployment velocity, they operate under strict constraints defined by the boundaries of the native binary compilation layer. OTA updates can only be done if all modifications have been made strictly within the JavaScript or TypeScript source code which compiles solely into the application bundle. Code modification that causes a change in the application's native compiled layers or how the native platform layer interacts with the JavaScript runtime is also unable to be distributed using OTA updates. The above restrictions on code modifications are as follows:

- Adding, removing, or changing versions of third-party node modules containing native extensions, which alter the compiled dependency graph by changing the native iOS `Podfile` or Android `build.gradle` structures through autolinking configurations.

- Altering primary application metadata, device access permissions, or platform-level configuration files such as `AndroidManifest.xml` and `Info.plist`.

- Modifying native source files or asset registries compiled directly into the binary container, such as the application launcher icon or native splash screen configurations.

If an update containing native alterations—whether introduced via manual file configuration or implicitly pulled in by upgrading a native node module—is distributed to an existing client binary via an OTA update, the application layer will experience an architectural mismatch. The new JavaScript bundle will try to call native functions that are not actually located in the pre-installed native runtime container. Because this is incompatible with the physical existence of these native functions it causes an error within the bridge abstraction layer (a fatal runtime) and thus the application crashes. Therefore, isolating dependencies from each other through clearly defining what specific areas of code have changed is key to maintaining structural integrity across the mobile ecosystem.

2.3.3 The Expo Updates Client Lifecycle: Channels, Branches, and Updates

To manage update execution within bare React Native applications, developers often integrate the `expo-updates` native client module. This library introduces a formal taxonomy to structure how updates are targeted, validated, and applied on user devices, utilizing an abstract data pipeline consisting of channels, branches, updates, and runtime directives.

- **Embedded Bundle:** The baseline JavaScript bundle and asset mapping compiled and packaged directly into the application binary filesystem during the initial build phase. This serves as the immutable fallback state; if no remote updates are available or if downloaded bundles are corrupted, the client runtime engine defaults to executing this local embedded bundle.

- **Update:** A single immutable deployment group containing a newly compiled JavaScript bundle, source maps, and static assets generated at a specific point in time from a developer's workstation or a CI/CD pipeline. Every update is associated with a unique identifier.

- **Branch:** An ordered, linear historical timeline of updates, conceptually equivalent to a branch within a Git version control system. A branch acts as a conceptual folder where new update groups are sequentially appended.

- **Channel:** A permanent deployment designation embedded directly into the native binary configuration at compile time (e.g., `production`, `staging`, or `preview`).

This architectural model allows for extreme operational agility. If a critical bug is discovered in production, an engineering team does not need to rush a new binary build through store reviews. Instead, they can adjust the channel-to-branch mapping on the server, publish a clean bundle update, or issue an immediate rollback directive straight to the production channel, modifying or resetting thousands of client application states instantly.

2.4 Continuous Integration, Deployment, and Quality Assurance Lifecycles

Automated governance of multiple applications will be needed in future mobile software development environments where the ability to compile on a developers machine (locally) is eliminated and replaced by centralized automation for compiling as part of an application's build process. This will allow for enforcement of strict quality gates at scale while eliminating manual compilation bottlenecks in the developer's environment and allowing for centralized management of code health across all related applications.

2.4.1 The Challenge of Automated Validation at Scale

Collaborative mobile application development is complicated when gatekeepers are manual, as this causes a breakdown in code quality through anti-patterns, architecture violations and runtime errors, also due to style divergences and broken type contracts. This instability is addressed using static analysis validation tools that run global validation on the entire project:

- **Static Linting:** Global scanning for anti-patterns, violation of architectures, and possible runtime errors found in all changed source files prior to compilation.
- **Code Formatting Standardisation:** Maintaining a uniform styling structure across all projects to prevent trivial style-related merge conflict and maintain readibility of your code.
- **Type Contract Verification:** Running static type compilers across shared workspaces to ensure that common interface contracts, function signatures, and data models have not been broken by incoming changes.

Although each of these static validation processes is lightweight and takes only minutes to complete they are only the initial layers of mobile quality assurance and serve as a foundation for the more time-consuming build phases.

2.4.2 The Mobile Build Bottleneck and Fingerprinting Theory

Unlike web applications, compiling mobile binaries (such as iOS `.ipa` or Android `.aab` packages) is highly resource-intensive, expensive, and time-consuming. In standard multi-application architectures, trivial modifications frequently trigger redundant native recompilations of the entire platform wrapper, creating massive inefficiencies in development velocity.

To optimize these workflows, modern CI/CD theory introduces context-aware routing based on the concept of *native fingerprinting*. A fingerprinting engine generates a deterministic cryptographic hash representing the exact state of the native platform configurations, manifests, and the underlying dependency tree within a project.

This state-of-the-art approach allows a pipeline to evaluate incoming code modifications and split the integration workflow into two theoretical paths based on structural impact:

- **Runtime-Isolated Changes:** If the native fingerprint remains unchanged, modifications are strictly limited to the high-level application logic (e.g., JavaScript or TypeScript components). The pipeline can theoretically bypass native compilation entirely, relying instead on lightweight Over-the-Air (OTA) bundle distribution.
- **Native Configuration Drift:** If the native fingerprints diverge, it indicates that native dependencies, platform-specific code, or native libraries have been modified. This state necessitates a full, heavy binary compilation runner to produce a new native testing asset capable of supporting the updated underlying architecture.

Chapter 3

Architectural Consolidation (The Monorepo)

The transition to a monorepo architecture required a rigorous re-evaluation of the existing codebase to decouple shared assets from application-specific business logic. This process was not merely a file-migration task but involved a strategic modularization aimed at establishing clear architectural boundaries across the entire mobile ecosystem. To achieve a sustainable, multi-package infrastructure, the intervention was executed through a multi-layered engineering approach, addressing code boundaries, dependency management, compilation bundling, and repository governance.

3.1 Definition of Common Logic and TypeScript Boundaries

The foundational step in our migration was identifying the "*Common*" layer: a set of packages designed to be consumed by both the Students and Faculty applications.

The primary component of this layer is the UI Library, which serves as the concrete implementation of the Politecnico di Torino's official Design System.

Prior to the structural shift, UI components resided within a local directory (`lib/ui`) of a monolithic repository. While structurally separated, these components were strongly coupled [11] with the business logic of the host application. For instance, basic components frequently imported React Contexts (such as user preferences) or specialized hooks (such as `useDeviceOrientation`) directly from the app's core logic.

In React, a Context is a mechanism designed to share data that can be considered "global" for a tree of components, avoiding the need to pass props manually through

every level—a challenge often referred to as "*prop drilling*" or pass-through variables [12]. This system relies on a Provider, a specialized component that wraps a portion of the application and "provides" the state to any descendant component that needs to consume it.

To reduce the existing coupling, we introduced a shared package where UI components are strictly presentational or depend only on shared "Core" providers. This necessitated a systematic analysis of the Context Providers to determine which could be generalized for the entire ecosystem. We identified four critical contexts and their relative implementations of Providers for synchronization:

- `PreferencesContext` (managed by `PreferencesProvider`): handling user-specific settings and persistence.
- `SplashContext` (managed by `SplashProvider`): controlling the initialization phase and the visibility of the launch screen.
- `FeedbackContext` (managed by `FeedbackProvider`): managing the display of bottom snackbars triggered by user interactions through Call-to-Action (CTA) buttons.
- `ThemeContext` (managed by `ThemeProvider`): enforcing the visual identity and dark/light mode switching across the applications.

The `PreferencesContext` presented a distinct challenge: both apps share a "Common" core of settings (accessibility, language, theme) but maintain unique app-specific flags. To avoid duplicating context logic while ensuring strict type safety, we combined TypeScript Generics with Interface Merging.

We defined a `CommonPreferences` type to include universal settings, such as `accessibility`, `campusId`, and `colorScheme`. To allow for extensibility, we implemented the context using a generic `Extra` parameter. As shown in figure 3.1.

The consuming applications can then instantiate the specialized provider by passing their unique preference shapes and configuration keys. This ensures that the `updatePreference` function remains fully type-aware, preventing the use of invalid keys or incorrect data types during runtime. For example, the initialization in either application would follow the pattern shown in listing 3.1.

```
1 <PreferencesProvider<AppPreferences>
2   extraEditableKeys={appEditablePreferenceKeys}
3   extraObjectKeys={appObjectPreferenceKeys}
4   initialPreferences={initialAppPreferences}
5 >
6 ...
7 </PreferencesProvider>
```

Listing 3.1: Preference initialization

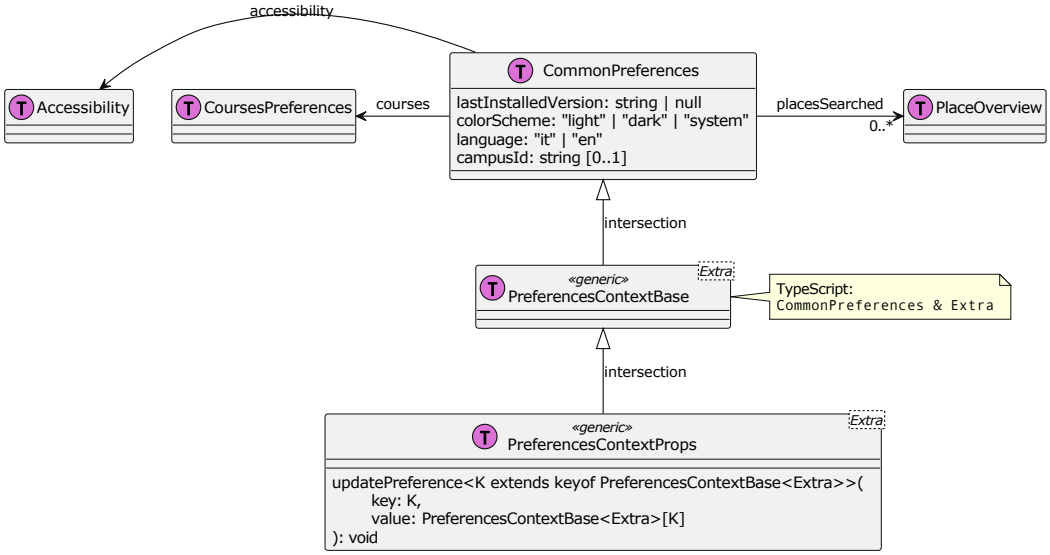


Figure 3.1: Generic types structure used to represent flexible preferences context

A significant portion of the shared logic involved the Places feature. Unlike simple UI primitives, this is a high-level functional module responsible for rendering interactive maps via Mapbox. It handles complex operations such as fetching custom vector tiles from Politecnico di Torino's servers to display indoor floor plans and managing POIs navigation.

Given that the physical infrastructure of the university is a constant across all user roles, the "Places" module was extracted into a standalone shared package. This ensures that improvements in indoor navigation or updates to the mapping engine are instantly available to both students and faculty, eliminating the risk of data divergence.

Finally, we centralized a suite of React Hooks and Utility functions—ranging from device state observers to API formatters—further reducing the footprint of the individual application repositories and ensuring a unified development experience.

3.2 Shared Library Architecture and API Definition

Once the shareable logic was identified, the next phase involved organizing the internal structure and defining the public API of the library. To ensure a smooth transition for the development team and maintain architectural consistency, the internal layout of the `lib` directory was modeled after the established conventions of the original Students App.

The internal organization follows a modular pattern within the `src/` directory, ensuring that both the shared library and the main application projects speak the same "architectural language." This symmetry reduces the cognitive load for developers switching between the shared core and app-specific code. The structure is organized as follows:

```
src/
├── core/      Foundational logic (Contexts, Hooks, Providers, Utils)
├── features/  High-level functional modules (Courses, Places, Maps)
└── ui/        Design System components and UI-specific logic
```

By mirroring the `core` and `features` directories in both the library and the individual app projects, we established a predictable environment where shared assets are easily discoverable. This structure allowed us to define a clear and scoped public API, enabling the applications to interface with the library through three main entry points:

- `@polito/lib/ui`: For Design System components.
- `@polito/lib/core`: For shared business logic and state management.
- `@polito/lib/features`: For complex, standalone functional modules.

A critical aspect of designing a shared library in a React Native environment is the classification of dependencies. Unlike standard web packages, mobile libraries often interact with native code, requiring a strict distinction to avoid build conflicts and duplicate symbols. We categorized the library's requirements into three tiers:

1. Dependencies: Essential logic-based libraries required for data processing, such as `luxon` for date manipulation.
2. DevDependencies: Tools strictly necessary for development, primarily consisting of TypeScript type definitions (e.g., `@types/`) for the project's internal libraries.
3. PeerDependencies: This category is crucial for native modules. Libraries that require native linking—such as `react-native-device-info` or `react-native-localize`—are declared as peer dependencies. This ensures that the consuming application provides the actual native implementation, preventing the "duplicate library" errors that frequently plague mobile monorepos.

During the architectural design, we evaluated whether to implement a formal versioning system for the shared package. While versioning (e.g., following SemVer)

offers stability, it introduces significant operational friction in a fast-paced development environment. Each change in the library would require a version bump, a new release, and a subsequent update in both applications.

To prioritize real-time development velocity, we opted to maintain the library at a static version (`1.0.0`). This choice allows the React Native bundler to treat the library as a "live" part of the source tree. Changes made within the `lib` directory are instantly reflected in both the Students and Faculty apps without the overhead of a release cycle. Furthermore, this strategy ensures that both projects are always perfectly synchronized with the latest shared logic, eliminating the risk of "version desynchronization" or more in general structural inconsistencies where one app might lag behind the other in terms of core features or bug fixes.

Achieving this seamless integration, however, required a specialized configuration of the React Native packager, which will be detailed in the following section regarding Metro's role in the monorepo.

3.3 Bundle Orchestration

In the React Native ecosystem, Metro serves as the specialized JavaScript bundler responsible for the critical task of resolution and transformation. Its primary function is to crawl the dependency graph, transpiling thousands of individual JavaScript files into a single, optimized bundle that the mobile application can execute efficiently via the Hermes engine.

While a standard React Native project typically relies on a pre-configured `metro.config.js` that requires little to no intervention, a monorepo structure introduces significant path resolution challenges. By default, Metro operates under a "project-root" assumption, meaning it is essentially blind to any resources located outside the application's immediate directory. To enable the seamless integration of the shared `lib` package and hoisted dependencies, we implemented a custom configuration designed to bridge the gap between the individual app directories and the monorepo root.

To grant Metro the visibility required to resolve shared assets and modules, we focused on three primary configuration properties:

1. watchFolders: This property allows Metro to monitor directories outside the local project root. In our implementation, we extended this to include the `monorepoRoot/node_modules` (where dependencies are hoisted by npm workspaces), the `lib` directory (containing our shared logic), and a centralized `assets` folder.

2. extraNodeModules: To ensure that imports such as `@polito/lib` are correctly resolved during the bundling process, we utilized this property to create a

logical mapping. This acts as an alias system, informing the resolver exactly where to find the physical source code for our shared library and assets.

3. Server Port Allocation: To facilitate an efficient Developer Experience (DX), we modified the default server port. By assigning unique ports to the Students and Faculty applications (e.g., `8081` and `8082`), developers can run and debug both applications simultaneously without encountering "address already in use" conflicts.

The implementation show in listing 3.2 illustrates the structural logic applied to the `metro.config.js` file.

```
const projectRoot = __dirname;
const monorepoRoot = path.join(projectRoot, '..');
const config = getSentryExpoConfig(projectRoot);
// 1. Expand visibility to include the library and hoisted modules
config.watchFolders = [
  projectRoot,
  path.join(monorepoRoot, 'node_modules'),
  path.join(monorepoRoot, 'lib'),
  path.join(projectRoot, 'assets'),
];
// 2. Map library aliases to their physical paths
config.resolver.extraNodeModules = {
  ...config.resolver.extraNodeModules,
  '@polito/lib': path.join(monorepoRoot, 'lib'),
  'assets': path.join(projectRoot, 'assets'),
};
// 3. Define unique ports for parallel development
config.server.port = 8081;
```

Listing 3.2: Metro configuration for monorepo support

By customizing the bundler in this manner, we effectively neutralized the physical distance between the applications and the shared code. This configuration ensures that despite the complex directory structure, the bundling process remains as fast and reliable as it would be in a monolithic setup, while gaining the scalability benefits of a multi-package architecture.

3.4 Dependency Management

Efficient dependency management is a cornerstone of an operational monorepo architecture. In a traditional polyrepo setup or non optimized multi-package structure, shared libraries must be independently resolved and downloaded inside each project sub-folder (e.g., `students/node_modules` and `faculty/node_modules`).

This approach results in significant storage redundancy and introduces the risk of version misalignment across the ecosystem.

To mitigate these drawbacks, this work leverages dependency hoisting—an optimization technique natively managed by package managers to consolidate third-party libraries. Instead of duplicating identical assets across individual subprojects, the package manager aggregates shared dependencies and lifts them into a single, centralized `node_modules` directory in the monorepo root. This centralized strategy yields three primary benefits: minimize disk footprint by removing duplicate packages, accelerate installation workflows since dependencies are downloaded only once, and provide structural guarantee that identical libraries share exact semantic versions across all execution environments.

In our architecture, hoisting is orchestrated via npm workspaces. This native mechanism was chosen primarily to maintain structural and operational continuity, leveraging the fact that the pre-existing standalone projects relied exclusively on the npm ecosystem. When evaluated against the alternative orchestration strategies reviewed in Section 2.2.2, npm workspaces offered the most balanced solution for our deployment scale. While enterprise-grade build tools such as Bazel or Nx provide sophisticated caching and dependency graphing for massive, multi-tiered codebases, they introduce significant configuration complexity. For our specific multi-application landscape, npm workspaces deliver a lightweight yet highly effective abstraction layer that achieves the desired modular integration without the operational friction or steep learning curve of more complex build engines.

However, npm implements hoisting conservatively. If two workspaces explicitly request divergent semantic versions of the same dependency, npm resolves the conflict by isolating the secondary version inside that specific workspace's local directory. Left unmanaged, this edge case introduces configuration drift and desynchronizes the environment. To enforce structural alignment without forcing developers to manually inspect deep dependency trees, we implemented a custom suite of automation scripts. These tools provide a standardized API for developers, dividing operations into two main branches:

1. Lifecycle Management: Wrapped commands—namely `npm run add`, `npm run remove`, and `npm run update`—abstract package operations. When executed within a specific workspace directory, the script targets only that micro-application or shared library. Conversely, when invoked from the root directory, the action is automatically mapped across the entire monorepo asset portfolio.

2. Consistency Audits: The utility script `npm run deps:check` programmatically parses the root-level `package-lock.json` (which serves as the single source of truth for the entire environment) to detect duplicate trees, unhoisted nodes, or version mismatches between workspaces. If an anomaly is identified, developers

can execute `npm run deps:fix`, which automatically restructures the local manifests to ensure uniform version compliance.

While hoisting optimizes dependency trees, it introduces a critical architectural vulnerability known as phantom dependencies. A phantom dependency occurs when a project successfully imports an undeclared third-party package simply because the module was hoisted to the root directory by a sibling workspace. This introduces significant fragility: if the sibling workspace removes or updates that dependency, the dependent project will encounter compilation or runtime failures without any local code modifications.

To neutralize this risk and safeguard the integrity of our codebase, we integrated Knip, an advanced static analysis tool specialized for modern JavaScript and TypeScript environments. Knip crawls the project infrastructure starting from designated entry points, systematically mapping import trees to evaluate resource utilization across two critical vectors:

- **Package Manifests:** It performs differential analysis to expose both *unused dependencies* (packages declared in a local manifest but never imported) and *missing/phantom dependencies* (packages imported in the code but omitted from the explicit dependencies array).
- **Dead Code Extermination:** By tracking the export-import pipeline, it isolates orphaned source files, dead variables, or unconsumed functions and TypeScript types that have become obsolete, keeping the overall application bundle lean.

The standardization of the codebase was finalized by hoisting all developer tooling configurations to the monorepo root. Rather than maintaining separate validation rule sets, a unified configuration matrix was established for TypeScript compilation targets, the Ruby Gemfile (governing native iOS CocoaPods dependencies), ESLint for static code analysis, and Prettier for automated code style enforcement as illustrated in the project layout of Figure 3.2.

To guarantee that these quality thresholds are rigidly observed, we utilized Husky to configure centralized pre-commit Git hooks. Before any local modifications can be permanently committed to the version control history, Husky intercepts the workflow to trigger automated linting, formatting checks, and TypeScript type verification.

Crucially, to prevent developers from bypassing these local protections via the `--no-verify` flag, identical automated verification workflows are mirrored directly within the remote continuous integration and deployment (CI/CD) pipeline. These validation checks are configured as mandatory quality gates within Pull Request (PR) reviews. A PR cannot be merged into the primary branches unless the

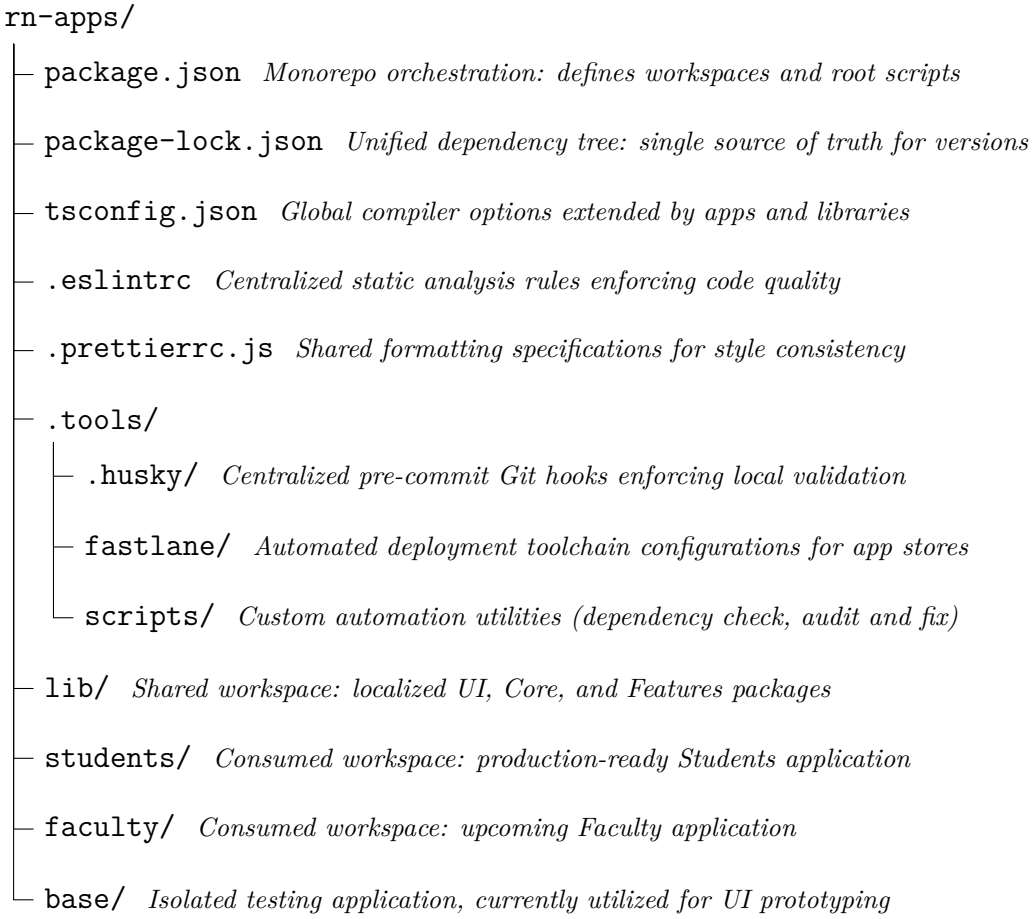


Figure 3.2: Monorepo directory structure.

automated pipeline validates that the code satisfies the unified linting policies, type declarations are sound, and all active dependencies are correctly hoisted and declared. This dual-layer validation loop ensures a highly stable, uniform, and predictable development environment across the institution's mobile ecosystem.

3.4.1 Version Control Management Strategies

The final stage of the architectural transition focused on establishing a sustainable version control strategy within Git. Rather than continuously modifying the pre-existing, fragmented repositories—which introduces operational risks and disrupts active workflows—we elected to initialize a completely clean, unified repository named `rn-apps`. The legacy repositories will be subsequently archived to serve as immutable historical points of reference.

A primary engineering constraint during this migration was the preservation of the extensive commit histories from both legacy projects. Maintaining this historical data is important for tracking regressions origins, understanding architectural choices, and retaining authors' attribution.

To achieve this without creating a disconnected or corrupted commit graph, we adopted a targeted multi-remote strategy. The process began with the repository containing the highest volume of historical data, the Students App. A new remote pointing to the unified `rn-apps` repository was added to the local configuration of the Students App repository. The entire source tree of the Students App was then shifted into a designated subdirectory locally. Because Git tracks content snapshots rather than explicit file paths, moving these files within the same repository preserved their complete linear commit history.

To merge the secondary project (the Faculty App) into this new topology while keeping its independent history intact, we utilized the native `git subtree` toolset. This utility allows a repository to be injected as a subdirectory of another project while seamlessly merging their independent commit graphs. The integration was executed using the following command sequence:

```
git subtree add --prefix=faculty/  main
```

By specifying the `--prefix` flag, Git safely mapped the entire commit history of the Faculty App into the `faculty/` subdirectory of the monorepo root. This technique prevented the loss of historical context and avoided the messy commit graphs often generated by traditional merge or rebase operations on unrelated codebases.

To prevent governance fragmentation within a shared codebase, the historical branch naming conventions were standardized into a single, unified policy governing branches, commits, Pull Requests (PRs), and issue tracking. This systematic alignment ensures traceability across project boards and automated changelog generation.

The structural framework defines specific boundaries based on the monorepo directories, classifying changes into Allowed Scopes (`students`, `faculty`, `base`, `lib`, `tools`) and Common Types (`feat`, `fix`, `docs`, `chore`). If an intervention modifies the global infrastructure, the scope designation is omitted.

- **Branching Model**: Following a branching model derived from Git Flow, all feature and fix branches diverge directly from `main` using a strictly enforced hierarchical path format: `<type>/<scope>/<short-description>`. In scenarios modifying multiple scopes, the targets are joined using a hyphen.
  - *Example (Scoped):* `feat/students/add-sso-login`
  - *Example (Multi-Scope):* `fix/students-faculty/session-timeout`

  - *Example (Global):* `chore/update-monorepo-structure`

- **Commit Standardization**: Commits must strictly comply with the *Conventional Commits* [13] specification, adopting the structural template: `<type>(<scope>):<description>`. Commits that modify multiple scopes separate the entities with commas, whereas global changes omit the parentheses entirely. Issue tracker references are appended explicitly within the footer:

  ```
  fix(students,faculty): resolve core session timeout sync
  Refs #10, Fixes #123
  ```

- **Pull Requests and Issue Tracking**: To maintain visual clarity across organizational boards, Pull Requests and issues mirror the repository structure by pre-pending capitalized scopes in brackets:

  - `[Students] Implement login with SSO`
  - `[Lib] Refactor Design System: Typography & Colors`

- **Tagging and Release Management**: Because independent workspaces within the monorepo follow unique deployment lifecycles on the application stores, tags cannot rely on a single, global version number.

  Git tags are isolated by pre-pending the respective application scope to the semantic version string, formatted as `<scope>-v<major>.<minor>.<patch>` (e.g., students-v1.9.2). This formatting allows the automated continuous integration pipelines to identify exactly which application needs to be compiled and shipped to production during a release tag event.

Chapter 4

Custom Over-the-Air (OTA) Distribution Infrastructure

The evolution of cross-platform development frameworks has redefined deployment and maintenance methodologies for enterprise mobile applications. This chapter details the design, development, and integration of a proprietary, self-hosted Over-the-Air (OTA) distribution infrastructure. This system was engineered to overcome the economic and architectural constraints imposed by third-party Software as a Service (SaaS) solutions while ensuring complete data sovereignty for the university's digital ecosystem.

4.1 Introduction and Technological Foundations

As established in Chapter 2, the intrinsic architecture of React Native relies on a clean separation between a native execution engine (the runtime compiled for specific mobile platforms) and the application logic written in JavaScript. This logical layer is interpreted at runtime by a dedicated engine such as Hermes. This fundamental decoupling unlocks the ability to implement Over-the-Air (OTA) update mechanisms. When changes are made in non-native layers — i.e., JavaScript bundles, style sheets or static assets — then new versions of applications may be distributed and applied immediately. The above system has the advantage of completely avoiding long submission and review processes as imposed by the app stores, including Google Play Store and Apple App Store.

The adoption of an OTA distribution workflow introduces significant operational advantages within a dynamic university ecosystem:

- **Instant Hot-fixing:** In cases where critical bugs occur while the software is running in production mode, they can be fixed within minutes so as to ensure

continued operation of services for students and faculty.

- **Decoupled Feature Testing:** Developers can deploy and test specific feature branches or hot fixes to actual devices physically without having to rebuild native binary applications (Android Package Kits (APKs), Android App Bundles (AABs), or iOS App Store Packages (IPAs)).
- **Dynamic Environment Routing:** Client devices can be seamlessly routed across distinct release tracks (e.g., *alpha*, *beta*, *production*), avoiding client-side re-installations.

Within the React Native ecosystem, the standard tool for orchestrating client-side updates is the `expo-updates` library. In the context of the unifications undertaken for the Politecnico di Torino, both the *Students* and *Faculty* applications include `expo` as an external dependency within the monorepo framework. This layout allows the applications to leverage non-native update modules while preserving full autonomy over their respective native codebases.

Client-side update discovery and installation strictly comply with the **Expo Update Protocol**, an open specification defining the HTTP communication layer between mobile clients and remote distribution servers. The update lifecycle relies on two primary logical endpoints:

1. **Manifest Endpoint:** The mobile client dispatches a `GET` request embedding specific HTTP headers, including `Expo-Runtime-Version`, `Expo-Current-Update-ID`, and `Expo-Channel-Name`. The distribution server parses these attributes against its persistence layer and responds with a binding directive. The two core protocol directives are:
    - *No Update Available:* Notifies the client that its current execution state is up to date, instructing it to bypass asset fetching and proceed with initializing the existing local runtime.
    - *Rollback:* Evaluates as a stateless directive instructing the client to invalidate local cached updates and immediately revert to the static bundle embedded natively inside the binary package.
2. **Assets Endpoint:** A dedicated file-delivery endpoint optimized for transferring the physical binary assets (minified JavaScript execution files, images, fonts) declared within the explicit update configuration.

The specific assets required for an update are declared within a compiler-generated `metadata.json` file. This file is produced by the Expo bundling pipeline during the project export phase. Together with the raw asset files, this metadata schema must be securely stored on the server to handle incoming update requests accurately.

To prevent critical application crashes caused by runtime mismatches, updates must be tied to a strict native boundary known as the **Runtime Version**. Because an updated JavaScript bundle might invoke native methods exposed by underlying modules, an OTA update must only target native binaries equipped with compatible underlying libraries. The `expo-updates` client framework offers three primary configuration policies to govern this identifier:

- **appVersion:** The runtime version is bound statically to the semantic marketing version declared in the global configuration of the project (e.g., `1.0.0`).
- **NativeVersion:** Combines the semantic version with the platform-specific internal build number (e.g., `1.0.0(1)`), adding a granular validation layer to catch intermediate app store submissions.
- **Fingerprint:** Represents the most resilient and automated strategy, which has been implemented in this thesis work. Utilizing the `@expo/fingerprint` utility, the system generates a deterministic hash based exclusively on native source code files, native package dependencies, and build configuration scripts. The runtime version changes only when the underlying native structure is modified, decoupling pure JavaScript releases from arbitrary text or configuration changes.

The dynamic actor interactions, data exchanges, and operational workflows distributed among the mobile client application, the administrative automation Command Line Interface (CLI) tool, and the distribution server are illustrated in Figure 4.1. For a more comprehensive and technical breakdown of the underlying data structures, response handshake sequences, and caching primitives, detailed specifications regarding the Expo Updates protocol can be found within the official platform reference material.[1]

4.1.1 SaaS Constraints and Expo Protocol Specifications

While commercial cloud platforms like Expo Application Services (EAS) provide scalable hosting for OTA updates, their SaaS tiering models introduce severe financial constraints that are incompatible with the operational scale of the Politecnico di Torino.

The free tier of EAS caps update distribution at a maximum of 1 000 Monthly Active Users (MAU), applying metered, variable pricing past this threshold. Infrastructure metrics indicate that the *Students* application alone averages approximately 40 000 MAU. Transitioning to the commercial *EAS Production* tier to accommodate

[1]Available at: `https://docs.expo.dev/technical-specs/expo-updates-1/`

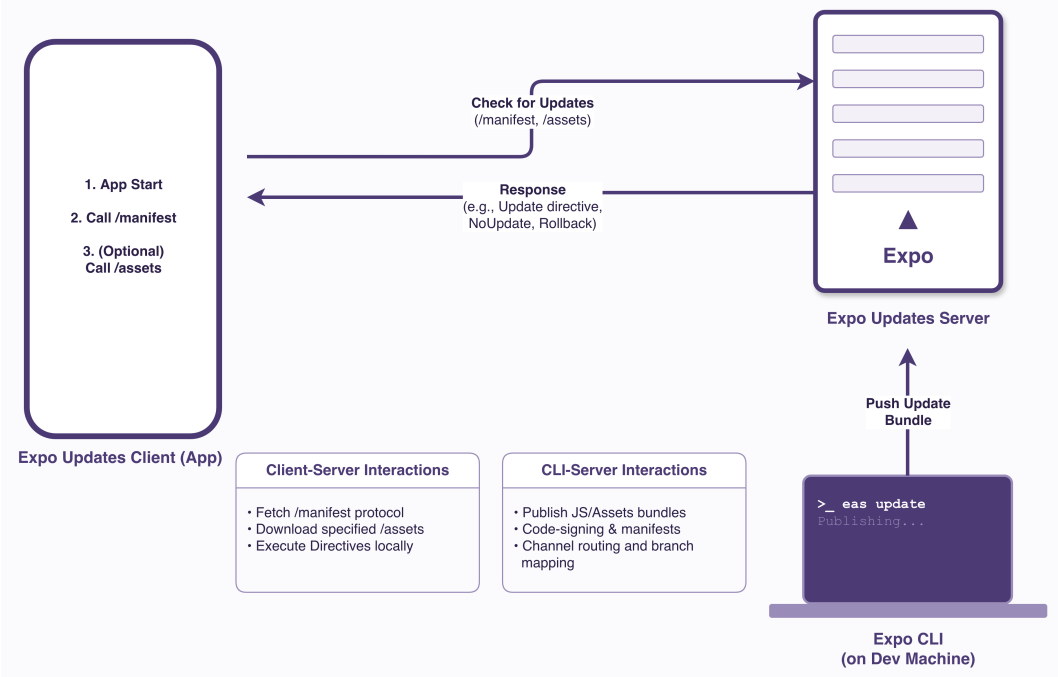


Figure 4.1: Expo Updates - Actor Interactions

this user base incurs a baseline cost of $199/month [14]. This fee is subject to additional variable surcharges tied to bandwidth utilization, storage allocation, and total cumulative update requests, making a fully cloud-hosted solution highly unpredictable and costly for institutional budgeting.

To circumvent these operational costs, the implementation of a self-hosted distribution server became an operational necessity. To facilitate this independent deployment, Expo provides an open specification for the Expo Update Protocol alongside an official reference implementation published on their GitHub repository [15]. However, this repository exists solely to provide a basic demonstration of how the protocol might be translated into executable code. It carries an explicit disclaimer from the maintainers stating that the codebase is not guaranteed to be complete, stable, or performant enough to serve as a production-grade backend.

Consequently, using the official demonstration server as a foundation for university infrastructure was unfeasible. During the initial design phase, however, a mature open-source project named `expo-open-ota` was discovered. With over 350 stars on GitHub, this repository offered an established foundational data plane. Contributing to and collaborating on this existing software presented a powerful opportunity: rather than developing an isolated, proprietary system, the engineering efforts expended during this thesis could be shared back with the community, benefiting not only the Politecnico di Torino but also a wider ecosystem

of developers facing identical infrastructure bottlenecks.

4.2 Evaluation and Collaborative Extension of Open-Source Infrastructure

The open-source repository `expo-open-ota` [16] is a specialized server architecture developed by software engineer Axel Marciano. Engineered entirely in Go to optimize memory allocation and maximize concurrent request-handling capacity, the platform introduces a high-performance alternative to standard node-based implementations. The server features several robust characteristics that make it suitable as a foundational data plane:

- **Object Storage Multi-providing:** Abstraction layers for file persistence supporting Amazon S3, Google Cloud Storage, and S3-compatible buckets.
- **Keystore Management:** High architectural flexibility in securing the asymmetric key pairs essential for generating the digital signatures required to sign update manifests cryptographically. The server natively accommodates diverse security environments by supporting file system execution through specific local paths, runtime injection via Base64-encoded environment variables, or advanced enterprise-grade protection at rest by delegating secret storage to dedicated cloud vaults, specifically integrating with AWS Secrets Manager.
- **Asset Delivery Performance:** Integrated routing mechanisms optimized for serving assets via CloudFront Content Delivery Networks (CDN) or pre-signed storage URLs.
- **Auxiliary Tooling:** A dedicated CLI (`eoas`), an initial monitoring user interface, and Prometheus/Grafana integrations to monitor bundle download distributions.

Although it has strong data-plane performance, a detailed architectural code inspection found that there is a fundamental structure issue in using this server for multi-app deployment. This server does not have either an internal relational database system or a centralized control plane. Instead of tracking state within itself, this server depends upon making an active call to the Expo API to determine channel/branch names and uses an automated session token from the Expo Dashboard. Additionally, all update definitions are written into static JSON files (`update-metadata.json` and `.check` markers), which reside directly inside object bucket path locations; these depend upon positional path look-ups to identify assets.

This architectural coupling compromised the objective of independent data sovereignty. By contacting the project's maintainer directly, a formal collaboration was established to redesign the core backend architecture, replacing the external dependencies with a relational database and an isolated, local Control Plane [17].

4.2.1 System Requirements and Data Modeling

The collaborative engineering process led to the definition of strict technical requirements to elevate the server to an enterprise-grade utility:

- **Resource Autonomy:** Complete isolation from third-party services. Applications, development branches, and distribution channels must be managed directly through local Control Plane APIs.
- **Database-Backed Keystore (Database KeysMode):** A secure operation mode where asymmetric private keys are stored sealed within the database using authenticated symmetric encryption (**AES-GCM**), driven by a protected `CONTROL_PLANE_MASTER_KEY`.
- **Persistence Layer Abstraction:** Structural isolation of storage operations within a dual-implementation framework. The `store` folder encapsulates object bucket interactions in files appended with `_bucket.go`, while relational database operations are isolated in files appended with `_postgres.go`, streamlining future dialect additions such as SQLite.
- **Scoped Access Control:** Elimination of global third-party session tokens. Introduction of high-entropy API keys (`EOO_TOKEN`) with an explicit `eoo_` prefix. These credentials are scoped per application with immutable, read-and-write privileges for version one, and feature immediate revocation capabilities. Keys are secured at rest using cryptographic hashing (**SHA-256**) and are displayed to administrators only once upon creation.
- **Type-Safe Data Operations:** Integration of the `sqlc` compiler alongside the native `pgx` driver in Go to compile raw SQL schemas into deterministic, type-safe data access code, optimizing transaction overhead against the PostgreSQL database instance.

The relational model illustrated in Figure 4.2 formalizes the domain hierarchy.

4.2.2 Architectural Implementation and Control Plane Endpoints

The server code was refactored into a clean 3-Tier Architecture [18], enforcing a strict boundary between protocol translation, business domain validations, and

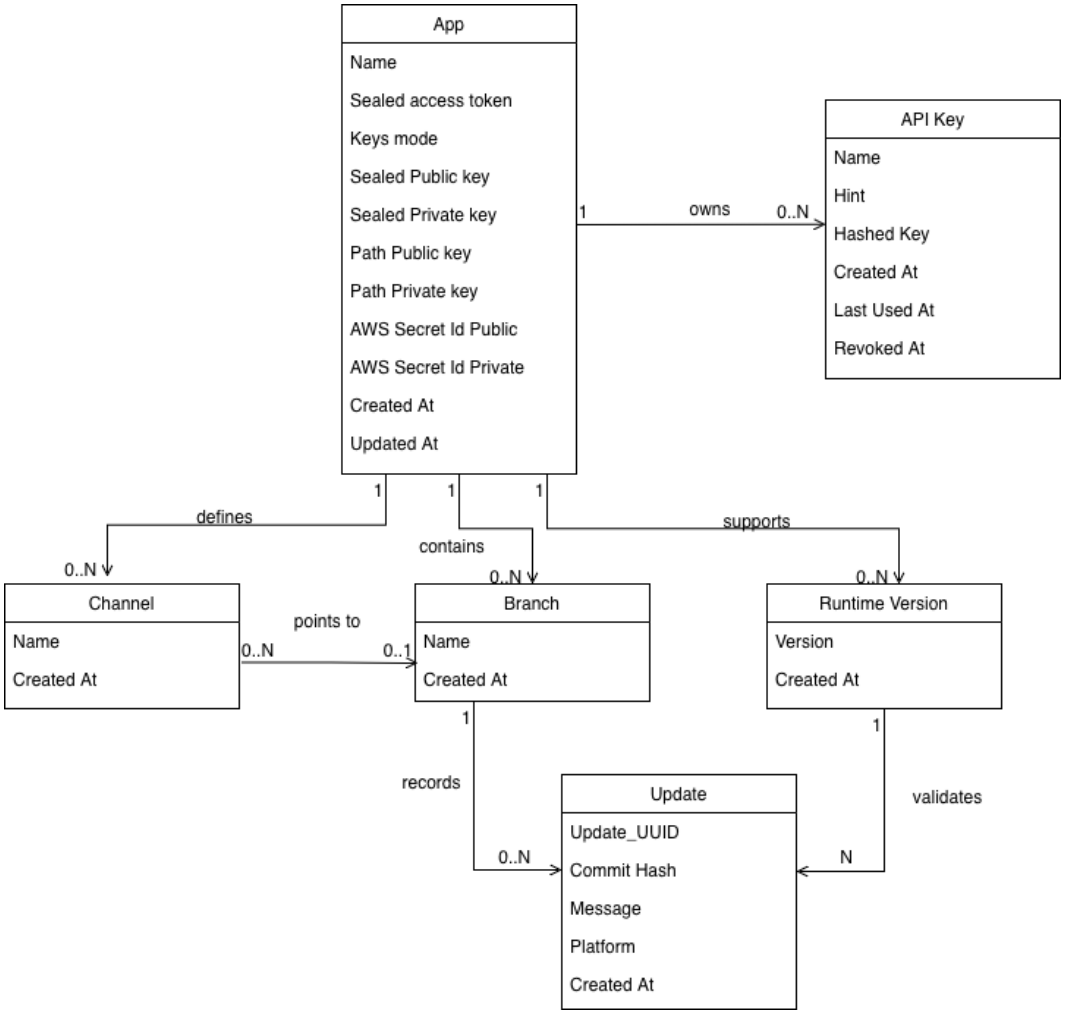


Figure 4.2: Relational data model schema introduced for local Control Plane state management.

persistence logic:

1. **Handler Layer:** Governs raw HTTP parameter parsing, query validation, and JSON content negotiation.

2. **Service Layer:** Executes pure business logic rules, such as verifying cryptographic signatures, evaluating release channel routing, and ensuring protocol header compliance.

3. **Repository/Store Layer:** Encapsulated within the `store` module, this tier abstracts file and database interactions. Coexistence between the original

architecture and the database-backed tier is achieved through dynamic feature-flagging. At boot time, the runtime checks for the presence of a `DB_URL` environment string. If discovered, the server instantiates the `PostgresStore` module to drive the relational Control Plane; otherwise, it activates a fallback loop to run in legacy bucket mode, as illustrated in Figure 4.3.

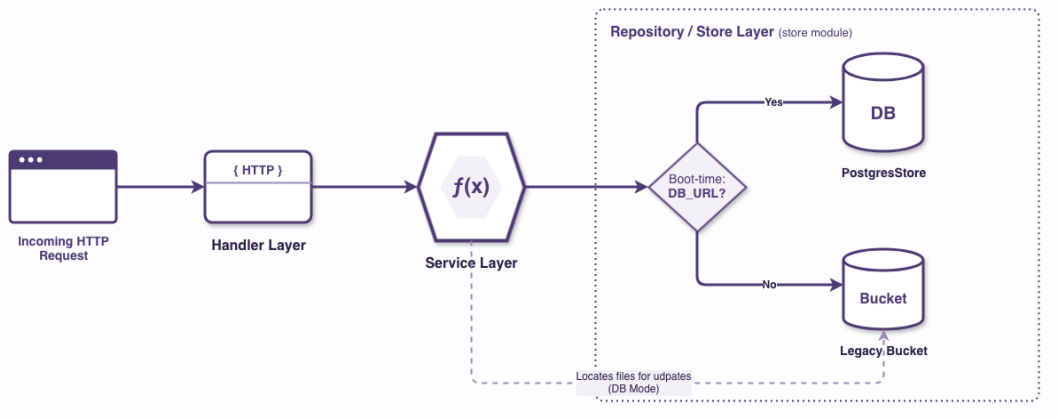


Figure 4.3: 3-Tier Architecture for Expo Open OTA Server

To expose administrative capabilities to the infrastructure, the following dedicated endpoints were integrated into the core HTTP routing multiplexer:

- `POST /apps` – Creating and configuring a new mobile application resource.
- `DELETE /apps/{APP_ID}` – Removing an existing application along with all of its associated resources.
- `PATCH /apps/{APP_ID}` – Changing the name of an individual application resource.
- `GET /apps/{APP_ID}/certificate` – Securely retrieving the public key certificate of the application being used by the client when in *Database mode*. This is so that the client may retrieve the appropriate validation keys for the public certificate used in validating signed manifest files.
- `POST /apps/{APP_ID}/branches` – Adding a new branch to an application.
- `DELETE /apps/{APP_ID}/branches/{BRANCH}` – Deleting an application branch.
- `POST /apps/{APP_ID}/channels` – Adding a new channel to an application.

- `DELETE /apps/{APP_ID}/channels/{CHANNEL}` – Deleting an application channel.
- `POST /apiKeys` – Generating cryptographically secured, app-scoped deployment tokens.
- `GET /apiKeys` – Retrieval of active API token metadata.
- `DELETE /apiKeys/{API_KEY_ID}/revoke` – Immediately invalidate and revoke a previously generated deployment token.

4.3 Dashboard Administrative Interface

To complement the newly engineered backend architecture, the web-based management dashboard was significantly extended to support the administrative capabilities introduced by the local Control Plane. This user interface is built on a responsive React framework and provides administrators with complete visibility and operational control over the application lifecycle, distribution paths, and security configurations.

A core enhancement of the frontend architecture is the implementation of a global React Context Provider situated at the root of the interface tree. During the initial authentication phase, this provider intercepts the global initialization configuration parameters directly from the server. If the server signals that it is running in the legacy bucket deployment mode without an underlying relational database, the dashboard dynamically adapts its execution state. It intercepts and restricts all state-mutating user actions—such as resource creation, metadata updates, or deletions—ensuring an explicit and error-free operational environment that prevents invalid operations on a stateless storage backend.

Conversely, when the relational database mode is active, the interface unlocks a comprehensive suite of administrative controls distributed across several specialized modules:

- **AppInfo Tab:** A new, dedicated management view implemented for both server execution modes (see comparison in Figure 4.4). This panel displays the active cryptographic keystore configuration currently utilized by the targeted application.
  - When operating in database mode (Figure 4.4b), it dynamically fetches and lists the authorized API tokens (`EOO_TOKEN`) associated with the application, rendering the last-used timestamps and providing administrative controls to instantly provision new keys or revoke compromised tokens.

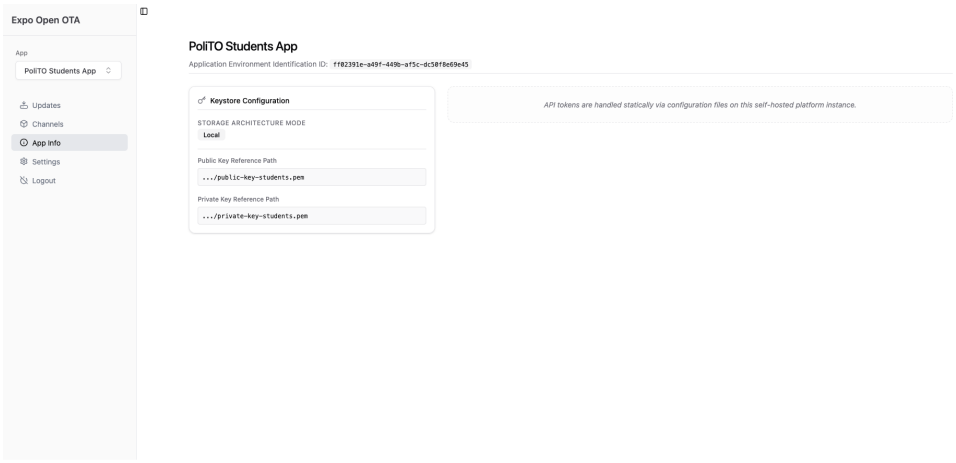
  - It exposes inline modification utilities, symbolized by an edit icon, to update the application name in the relational database.
  - It incorporates an isolated "Danger Zone" section that safely structures the deletion workflow for removing the entire application entity.
  - When operating in database mode, it features a download utility that allows to export a generated public security certificate with a fixed ten-year validity period. This file is directly integrated into the client application codebase to facilitate runtime cryptographic validation of signed manifests.

- **Unified Application Provisioning:** A "New Application" button equipped with a comprehensive input modal (illustrated in Figure 4.5). This interface guides the administrator through selecting the desired keystore residency model. While Database Mode is presented as the secure default, the modal dynamically adapts its fields to support alternative architectures:
  - Selecting *Local File Paths* requiring directory paths to the public and private key files on the host file system.
  - Selecting *AWS Secrets Manager Mode* dynamically updates the form requirements to request the explicit secret identifiers and resource names for both the public and private components within the cloud vault.

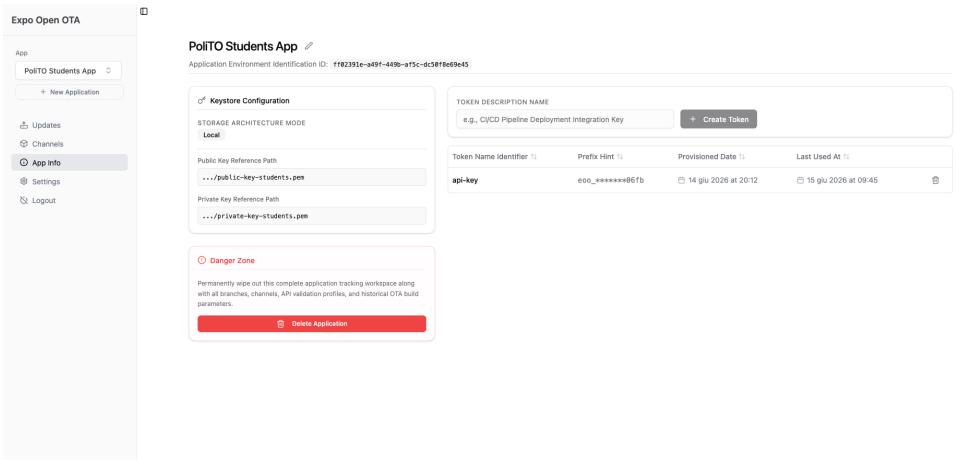
- **Resource Lifecycle Cards and Destructive Guardrails:** The orchestration of deployment tracks is structured into modular component cards dedicated to provisioning and mapping branches and distribution channels (see layout comparison in Figure 4.6). To mitigate the risk of catastrophic accidental data loss within a production environment, all destructive actions are bound to specialized safety modals. These warning systems explicitly outline the structural consequences of deleting applications, branches, or channels, requiring explicit multi-step user confirmation before propagating the deletion signals to the repository layer.

4.4 Command Line Automation: The `eoas` CLI tool

Developer workstations and automated Continuous Integration (CI) nodes interact with the self-hosted distribution server using the extended `eoas` command-line utility. The CLI tool has been re-engineered to authenticate with the new infrastructure using the application-scoped `EOO_TOKEN` system variable, removing the reliance on proprietary third-party session cookies. To ensure backward compatibility and a



(a) App Info: No-Database mode.



(b) App Info: Database mode.

Figure 4.4: Comparison of the App Info Tab.

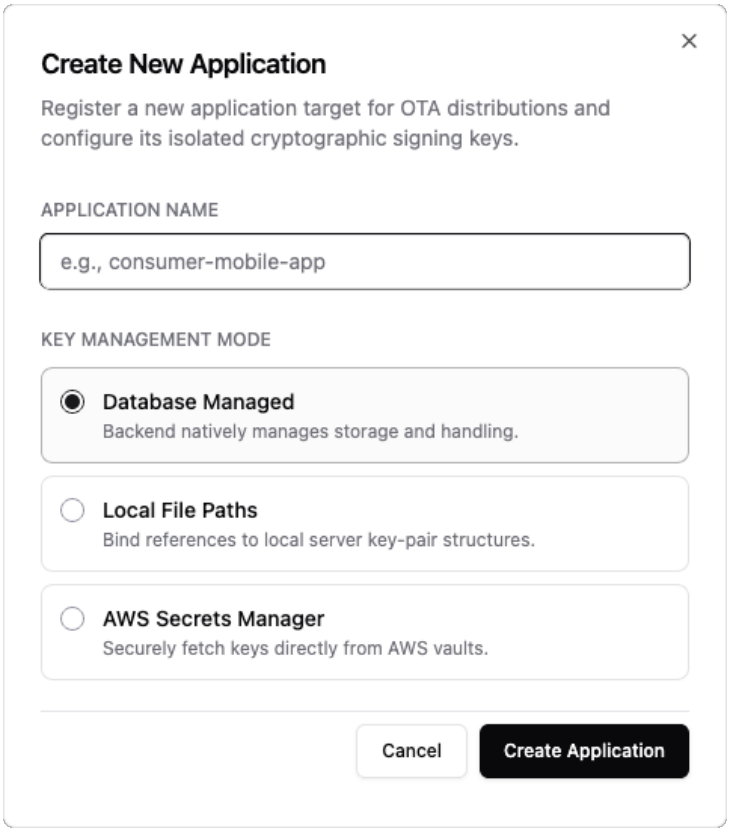
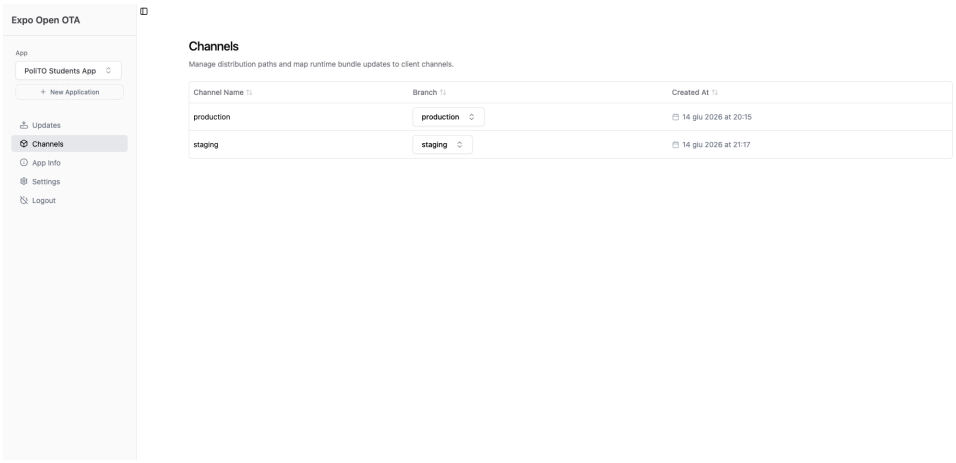
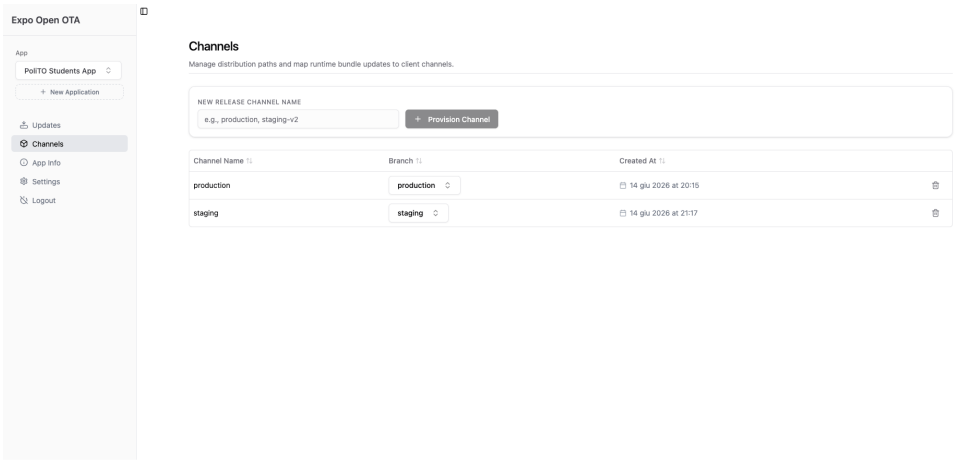


Figure 4.5: Application Creation Modal.



(a) Channels: No-Database mode.



(b) Channels: Database mode.

Figure 4.6: Comparison of the Channels Tab.

smooth transitional workflow, both authentication systems currently coexist within the client utility: the tool first attempts to resolve the local `EOO_TOKEN` to establish a session with the self-hosted database Control Plane; if this token is not found in the environment, the execution logic seamlessly falls back to standard Expo authentication utilizing either an `EXPO_TOKEN` or a localized session secret.

The utility orchestrates five primary commands to manage the software release lifecycle:

1. **`init`:** Generates and configures the local `app.config.js` file inside the client application directory, embedding custom server destination URLs and mapping unique local application identifiers.

2. **`generate-certs`:** Launches an interactive wizard that guides developers through the creation of asymmetric cryptographic key pairs, generating the required public and private certificate files locally within a specific destination folder. The developer can then manually configure the private signature key on the server. This setup enables public-key certificate verification on incoming manifest files at runtime, preventing Man-in-the-Middle (MitM) exploits or unauthorized update injections.

3. **`publish`:** Triggers the local compilation and transmission of JavaScript bundles using the following command structure:

   ```
   npx eoas publish -branch <branch-name> [-platform <platform>]
   ```

   The CLI tool calculates the current active *Runtime Version* from local native fingerprints, minifies source directories, produces the definitive `metadata.json` manifest, bundles graphical components, and transfers the package to the server via a signed multipart HTTP request.

4. **`rollback`:** Deploys an instant restoration directive to a targeted environment track using the following command structure:

   ```
   npx eoas rollback -branch <branch-name> [-platform <platform>]
   ```

   This directive does not erase server file records. Instead, it instructs the database to respond to incoming client updates with an explicit `Rollback To Embedded` instruction. This forces receiving mobile devices to purge local cached updates and immediately re-initialize the baseline native bundle distributed with the original app store installation file.

5. **`republish`:** Acts as a highly resilient downgrade mechanism for production tracks. Executed as an interactive command via `npx eoas republish`

`-branch <branch-name>`, the tool allows developers to query the server's database and browse the historical timeline of updates filtered by *Runtime Version* and *Update ID*. Once a stable historical update is identified, the command instructs the server to re-link that specific update metadata to the head of the deployment branch. Unlike a standard native rollback—which forces a retreat to the embedded bundle that may lack months of iterative non-native fixes—the `republish` command enables a controlled downgrade to the last known stable JavaScript bundle, resolving unexpected production regressions instantly.

4.5 Client-Side Integration: Expo Updates Client

The stabilization of the custom distribution infrastructure necessitates a robust client-side implementation within the mobile application portfolio. Currently, the `expo-updates` client library has been integrated and validated inside the *Students* application, as it represents the sole platform actively in production. However, due to the structural unifications achieved through the monorepo architecture detailed in Chapter 3, this identical setup can be instantly propagated to the upcoming *Faculty* application by sharing the foundational configuration logic.

4.5.1 Native Configuration Overrides in Bare React Native Workflows

When the Expo ecosystem is integrated into a pre-existing, bare React Native project rather than utilizing a default managed workflow template, the high-level configuration files—namely `app.json` and `app.config.ts`—are insufficient on their own to propagate OTA properties down to the compilation layers. To allow for full support of background update reconciliation, an explicit structural modification needs to be made to the native platform directories. This configuration defines critical parameter values including the targeted runtime version and the specified update server gateway URI.

The synchronization relies on modifying two platform-specific configuration manifests:

- **`Expo.plist` (iOS):** A property list configuration file located within the native iOS workspace directory. This file needs keys dictating the target URL of the custom self-hosted server (`EXUpdatesURL`) and the matching runtime identification policy (`EXUpdatesRuntimeVersion`).
- **`AndroidManifest.xml` (Android):** The primary application manifest inside the Android source tree. This file needs specific `<meta-data>` tags

within the main `<application>` boundary to declare properties such as `expo.modules.updates.EXPO_UPDATE_URL` and `expo.modules.updates.EXPO_RUNTIME_VERSION`.

4.5.2 Production Update Lifecycle Policy

Following architectural synchronization sessions involving UX designers, core members of ISIAD, and the thesis supervisor, a deterministic background execution policy was selected as the institutional default for production releases.

The application utilizes an automated background fetch model when launched by an end-user. When booting from a cold start, the client dispatches an asynchronous, non-blocking request to the manifest endpoint of the custom server requesting evaluation. If the custom server finds a validated and compatible update package, it will begin a background download in which it downloads the new JavaScript payload and graphical assets without disrupting the user's current usage session. Once the update has been downloaded, it is staged locally on the client for seamless loading upon the next startup of the application. This lifecycle paradigm guarantees zero performance overhead or blocking load screens, preserving a transparent and fluid user experience.

4.5.3 Dynamic Environment Routing and Channel Surfing

To facilitate rapid testing and feature Quality Assurance (QA) without continuous native binary distributions, a dedicated runtime overriding mechanism was designed for internal testers. This feature utilizes Expo's *channel surfing* paradigm [19], which allows an application binary to dynamically switch its targeted distribution track on the fly without altering the immutable native compilation layout. The capability is exposed via an administrative selector view hidden within the standard application settings panel (Figure 4.8).

User Authorization	**Backend Endpoint Return**	**UI Behavior**
Standard Student	Zero available testing tracks.	Selector hidden from view.
Authorized QA Tester / Developer	Array of valid release channels (e.g., *alpha*, *beta*).	Interactive selector rendered in settings.

Figure 4.7: Conditional authorization matrix for runtime distribution channel selection.

The available channels are retrieved dynamically through the internal PoliTO API client. While deployment on the university backend is scheduled for a future

development phase, the complete client-side business logic is fully integrated. When a user navigates to the configuration panel, the planned backend will act as an authoritative proxy: it evaluates the authenticated user's ID, securely queries the self-hosted OTA server by embedding an application-scoped `EOO_TOKEN`, and streams the authorized release tracks back to the client. As shown in Figure 4.7, standard students receive an empty dataset (hiding the selector), whereas authorized testing personnel can view and select accessible development tracks.

Programmatic channel surfing is executed by invoking the `setUpdateRequest HeadersOverride` primitive from the `expo-updates` SDK to modify the outgoing `expo-channel-name` header. Crucially, this primitive operates at the underlying native layer rather than just an ephemeral JavaScript state; it writes the overridden headers directly to the application's persistent native internal storage. Consequently, the client library continuously utilizes these headers across cold starts and application restarts until the app is completely uninstalled or the override is explicitly cleared.

To manage this native persistence predictably, the workflow is paired with the monorepo's shared application preferences framework described in Chapter 3. When an authorized tester updates the target track, the selection is saved to disk, and the client immediately fetches and downloads the corresponding OTA update package in the background. Once the download completes successfully, a prompt appears informing the user that an application restart is required to apply the changes, swap out the active runtime bundle, and safely initialize the new code context.

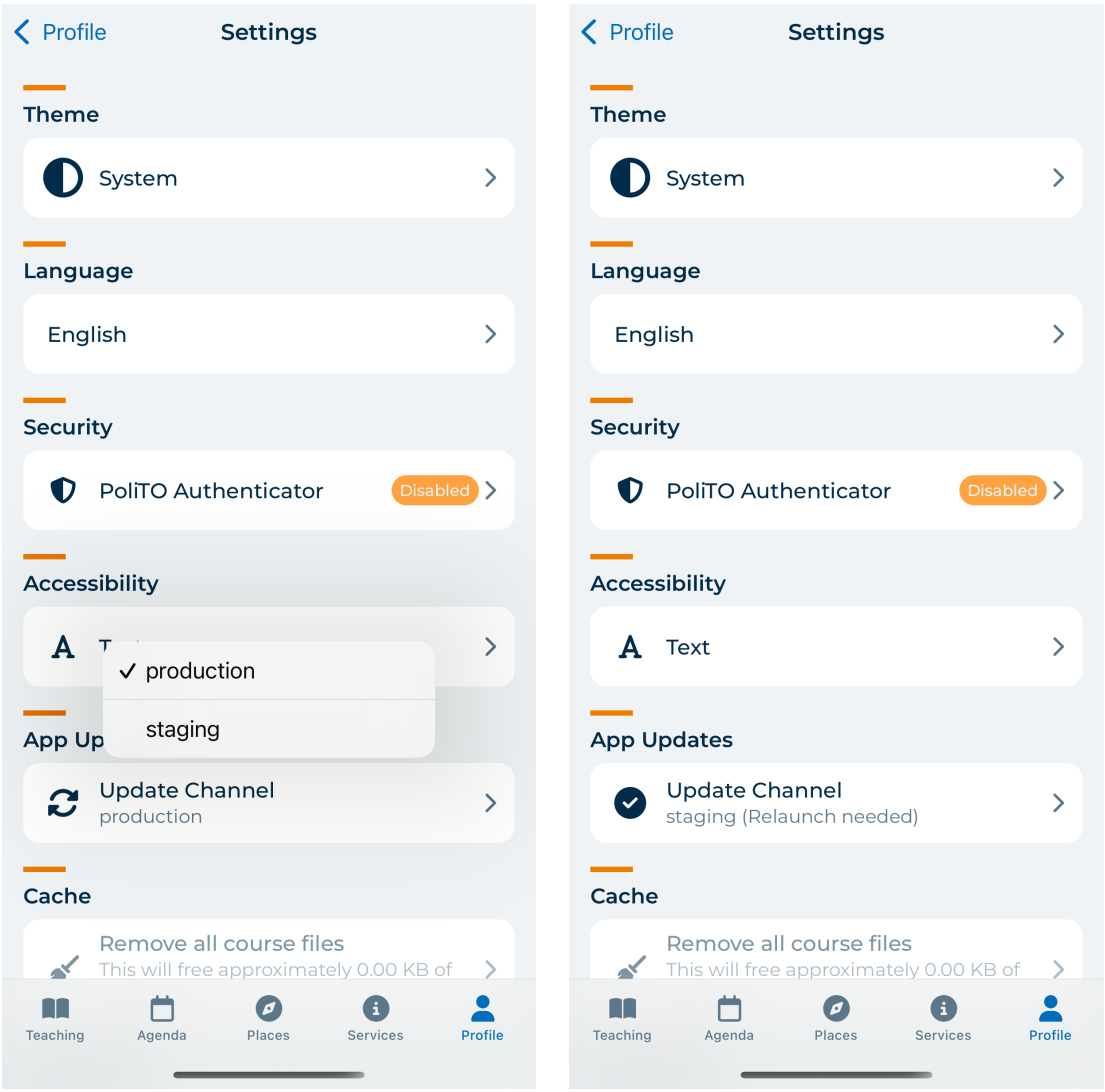


Figure 4.8: PoliTO Students: OTA update channel switching mechanism.

Chapter 5

Context-Aware Pipeline Redesign and Mobile DevOps Automation

The operational scaling of a multi-app monorepo framework demands a strict restructuring of continuous integration and continuous deployment (CI/CD) architectures. In a fragmented polyrepo model, deployment structures are inherently localized, leading to high infrastructure redundancy, unpredictable release intervals, and extensive execution delays. This chapter details the comprehensive engineering redesign of the automation toolchains at the Politecnico di Torino. By consolidating deployment engines and building a custom, context-aware orchestration layer, the new infrastructure transforms continuous integration into a matrix-driven, deterministic workflow.

5.1 Legacy Infrastructure Bottlenecks and Fastlane Foundations

Prior to this architectural migration, the Students application relied on a dedicated, standalone integration pipeline. Native compilation and store distribution tasks were driven by Fastlane, an open-source automation engine engineered to wrap complex deployment workflows for mobile applications. Within the legacy structure, Fastlane executed critical operations, including:

- **Automated Provisioning (Fastlane Match):** Centralizing code-signing configurations by storing encrypted iOS provisioning profiles and certificates within a secure, dedicated remote Git repository, establishing a single source

of truth across development workstations and ephemeral CI runner nodes.

- **Binary Compilation Management:** Mapping build hooks to platform-specific compilers (xcodebuild for iOS and Gradle for Android) to automate version tracking, code signing, and artifact generation.
- **Targeted Distribution Tracks:** Directing compiled binaries to Apple TestFlight and the internal or beta release tracks of the Google Play Console.

The previous pipeline model used Fastlane through two different execution patterns. The tool was run via the Pull Request verification cycle on the runner, where it ran the appropriate compiler for each target platform to determine if new source changes would fail native builds before they were merged to the `main` branch. Alternatively, it was run by a manual `workflow_dispatch` event or by a strict semantic Git Tag Push event; both events would trigger a full distribution lifecycle. This distribution lifecycle process accepted many input parameters related to how applications are distributed in stores. Input parameters included:

- A boolean flag that controlled whether deployments were sent to Apple's TestFlight.
- Targets for Google Play Store tracks (`internal`, `beta`, `production`).
- Runtime values for version number overrides written to the applications `package.json` configuration. These versions defined the final build numbers for all marketplaces being targeted.
- Integer values representing individual build-pieces. For example, a default value of `90` meant an official production build, while lower integer values represented intermediate testing artifacts created internally.
- Boolean flags that would automatically create a formal GitHub Release and generate changelogs along with creating separate *APK* assets for hardware platforms that do not have Google Play Services.

Despite its operational breadth, this monolithic setup exhibited severe efficiency constraints. Because every code modification—even a minor fix or styling change in a TypeScript file—triggered native compilation workflows in parallel, the median execution threshold reached approximately 30 minutes on Android runners and 20 minutes on macOS iOS nodes. As the development ecosystem expanded to include the Faculty application, maintaining duplicate, decoupled continuous integration matrices across distinct repositories became unfeasible. The integration of Over-The-Air (OTA) capabilities presented a clear opportunity to redesign this lifecycle. By identifying structural deltas prior to build allocation, the orchestration layer

could decouple lightweight JavaScript bundle exports from resource-intensive native binary compilations, compressing integration feedback loops from tens of minutes down to a few minutes.

5.2 Centralization of the Global Fastlane Engine

The primary objective during the infrastructure redesign was the total removal of duplicated deployment configurations across the monorepo workspaces. In a conventional multi-target React Native repository, Fastlane expects a local configuration directory (`/fastlane`) containing localized `Fastfile` and `Appfile` manifests embedded directly within each platform's native folder layout (`/android` and `/ios`). For an ecosystem encompassing two production-grade mobile applications, this layout would require maintaining four distinct, independent Fastlane execution nodes, a pattern highly prone to configuration drift and structural errors during routine updates.

To resolve this limitation, the entire automated toolchain was centralized into a single, highly generalized global engine situated within the root-level `tools/fastlane` directory. This shared architecture abstracts platform-specific variances by introducing parameterized execution paths. The internal automation logic is structured around three primary, dynamic lanes:

1. **`build`:** Orchestrates pure native compilation cycles for both iOS and Android configurations, handling local provisioning, resource compilation, and raw artifact signing.

2. **`release`:** Composes the foundational functionalities of the `build` lane, adding automated delivery hooks to securely transmit the compiled binary packages to the respective digital marketplaces.

3. **`apk`:** A specialized execution path restricted to the Android platform, designed to compile and output stand-alone, production-signed *APK* files. This ensures the application remains fully accessible to students operating devices without Google Play Services, providing a direct installation package independent of any specific distribution network.

To drive these centralized lanes dynamically across different applications and target operating systems, the global engine processes runtime arguments passed via the Fastlane (*CLI*). Commands are dispatched from the runner utilizing the following parameterized structure:

```
bundle exec fastlane [lane_name] app:[app_name] platform:[os_name]
```

When a lane is initialized using this configuration, the engine will read the `app` and `platform` context arguments to perform two initialization actions. In the first step, the engine references one environment matrix to safely extract application-specific credentials. It then extracts the correct keystore and password for either android or maps the correct provisioning profile via Fastlane Match for iOS. The second action that the engine performs is changing the location of its current working directory so that it points at the specific application subdirectory within the monorepo workspace. This centralized model creates an immutable and very scalable model for deploying software. New mobile applications are able to be added to the university's digital portfolio without having to write new code for continuous delivery (CD), they just need to add structural data about their own applications to the base configurations files.

5.3 The Context-Aware Orchestrator Plane

With the deployment actions centralized, the subsequent engineering phase focused on designing a highly intelligent, context-aware orchestrator plane within the GitHub Actions framework. The objective was to eliminate redundant execution overhead by answering two core questions during any Git event: which application workspaces have been affected by the incoming commit, and do those changes require a native compilation or can they be fulfilled via an OTA update bundle? The comprehensive context-aware execution paths and their architectural integration across the separate deployment tracks are illustrated in Figure 5.1.

The orchestration architecture is built upon a deterministic matrix-processing engine driven by a centralized coordinator script written in JavaScript, located within the `tools/ci` directory. Choosing a high-level scripting language over conventional bash files allowed for greater structural flexibility when parsing complex Git differentials and manipulating JSON-based execution matrices. The continuous integration lifecycle is initialized via three distinct entry routes: automated Git triggers (such as Pull Request updates or target branch merges), semantic Git tag pushes for official releases, and manual `workflow_dispatch` inputs. The manual dashboard configuration has been explicitly updated to allow administrators to isolate specific target applications and pass explicit override flags that force native compilations, bypassing automated checking when manual verification is required.

For automated triggers, the orchestrator begins by parsing the file change vector of the current commit hash. To map workspace boundaries accurately, target evaluations are governed by precise path dependency rules. An application workspace inside the monorepo matrix (e.g., `students` or `faculty`) is flagged as active if the underlying file changes intersect with:

- The explicit application subfolder directory.

- The centralized shared library workspace (`lib/`), ensuring that modifications to the core Design System or shared "Places" module instantly trigger downstream validation for all consuming clients.

- The root-level `package-lock.json` manifest, acknowledging that dependency modifications within an optimized, hoisted workspace environment can alter execution contexts across all application nodes.

Once the target application matrix is resolved, the orchestrator determines the exact compilation pathway by executing native structural analysis. This process utilizes the native cryptographic tracking engine of the `@expo/fingerprint` utility. The coordinator script generates a deterministic runtime hash representing the current native configuration of the active branch. It then compares this value against an authoritative baseline hash. For Pull Request workflows, the baseline is generated by evaluating the target destination branch; for tagging events or manual release dispatches, the orchestrator targets the last verified release tag currently active in production.

If the calculated native fingerprint matches the baseline hash identically, the orchestrator concludes that the incoming modification is strictly confined to the non-native application layers (e.g., pure TypeScript logic, styling definitions, or presentational assets). Consequently, the target application identifier is pushed into an execution array denoted as `bundleApps`. Conversely, if a native discrepancy is discovered—indicating that a native package has been added, an iOS Podfile modified, or an Android manifest altered—the target application is routed into an execution array denoted as `nativeApps`.

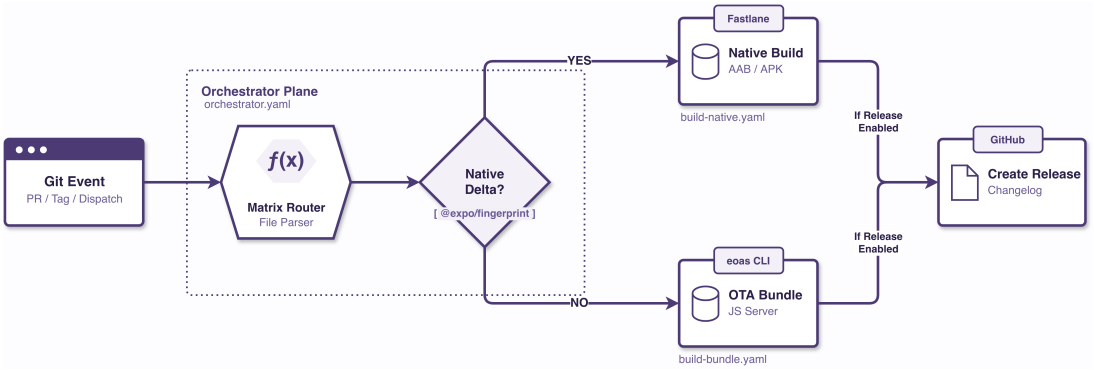


Figure 5.1: Context-Aware CI/CD Execution and Orchestration Topology.

This differential division allows the orchestrator to balance pipeline loads, dispatching secondary triggers to two distinct, highly specialized sub-workflows: `build-native.yaml` and `build-bundle.yaml`. The native sub-workflow boots

runners to execute the parameterized global Fastlane lanes, managing full compilation and marketplace delivery. The bundle sub-workflow bypasses compilation layers entirely. If the event is an internal check, it triggers a lightweight `expo export` cycle to validate code syntax; if the event represents an official release on a production track, it compiles the JavaScript payload in seconds and utilizes the custom `eoas publish` CLI utility detailed in Chapter 4 to transmit the signed update directly to the self-hosted distribution infrastructure.

5.3.1 Ensuring CocoaPods Lockfile Consistency

A recurring challenge in cross-platform mobile monorepos is ensuring native dependency configuration consistency among developers operating on different hardware platforms. In a team where certain engineers utilize workstation machines running Microsoft Windows or Linux distributions, those local environments lack the native macOS capabilities required to execute the CocoaPods package manager during dependency updates. When a native React Native module is added or updated within the monorepo, the installation scripts accurately modify the root-level `package-lock.json` and the JavaScript dependency layers. However, the accompanying platform-specific native file—the iOS `Podfile.lock`, which serves as the precise lockfile for native iOS frameworks—cannot be updated locally by non-macOS systems.

If a developer commits these changes without updating the native lockfile, it creates a critical synchronization failure within the continuous integration infrastructure. The incomplete configuration changes are pushed to remote branches, distorting the input paths analyzed by the `@expo/fingerprint` engine. The orchestrator may fail to recognize an underlying native modification due to the missing lockfile data, incorrectly routing a native change into the lightweight `bundleApps` pathway. This configuration error would deploy an incompatible JavaScript bundle to active devices, causing runtime application crashes due to missing native modules.

To systematically prevent this failure mode, the orchestrator includes a mandatory validation gate for all incoming Pull Requests. Upon execution, the orchestrator scans the workspace file tree to isolate any modifications intersecting native iOS paths. If a discrepancy is detected between the JavaScript dependency manifests and the corresponding state of the `Podfile.lock`, the orchestrator halts downstream execution blocks. It marks the integration checks as failed and dispatches an automated comment via a dedicated institutional ChatOps agent, preventing the corrupted branch state from being merged into stable lines while offering immediate automated mitigation options.

5.4 ChatOps Governance: The Minerva ChatOps Bot

To streamline the interaction between the complex monorepo infrastructure and the development team, the continuous integration architecture incorporates a specialized ChatOps engine driven by a proprietary bot profile named **@polito-minerva**. ChatOps bridges the gap between automation processes and manual governance, allowing developers to execute precise operational workflows directly from the comments interface of a Pull Request without context-switching to external console systems [20].

When a developer opens a new Pull Request against the unified repository, the *Minerva* engine interceptor hooks into the event, instantly evaluating the branch integrity and publishing a formal, markdown-rendered interactive greeting comment directly inside the conversation feed (Figure 5.2).

The ChatOps agent processes three core slash commands, executing them on isolated remote environments:

- **`/rebase`:** Instructs the automation server to pull the latest commit history from the `main` production line and execute a Git rebase operation on the current working branch. This validates conflict alignment and keeps the commit log linear without requiring local terminal interventions from the developer.

- **`/fix-locks`:** Specifically engineered to resolve the platform inconsistency issues described previously. When a developer operating on a non-macOS machine pushes a dependency shift that corrupts the iOS lockfile, typing this command instructs Minerva to boot an isolated, cloud-hosted macOS runner. The remote node initializes a complete installation loop, synchronizes the native `Podfile.lock` structures accurately, and automatically commits the resulting modifications back to the developer's active PR branch. This automated fix clears the validation blocks and allows the pipeline checks to resume automatically.

- **`/deploy`:** Designed to accelerate feature testing and quality assurance cycles. When invoked, Minerva instructs the orchestrator to provision an ephemeral, isolated testing branch and a corresponding routing channel on the self-hosted distribution server. The pipeline then compiles and deploys an isolated OTA update preview to this temporary track. Authorized testing personnel can instantly access and evaluate the new feature on physical devices via the settings panel described in Chapter 4, without requiring an app store release. To preserve ecosystem stability, the command executes a strict native pre-flight check: if the branch contains native modifications that alter the baseline

fingerprint, Minerva aborts execution and posts an error message, ensuring that runtime-breaking changes never reach the testing devices.

Figure 5.2: Automated comment initialized by @polito-minerva within Pull Request contexts.

Chapter 6

Evaluation and Impact Analysis

The structural transition of the mobile ecosystem at the Politecnico di Torino from a fragmented multi-repo pattern to a unified monorepo architecture, backed by a self-hosted OTA distribution infrastructure, requires a systematic quantitative and qualitative evaluation. This chapter presents an empirical analysis of the engineering impact of these interventions, measuring improvements across codebase maintainability, continuous integration workflows, and distribution deployment velocities.

The following sections detail the quantitative and qualitative metrics compiled across the three core areas of the intervention: code composition deduplication, Continuous Integration compilation timelines, and the overall distribution velocity of the self-hosted infrastructure.

6.1 Codebase Composition and Deduplication Metrics

To evaluate the structural efficiency of the architectural consolidation, a differential quantitative analysis was performed on the software metrics of the mobile ecosystem before and after the migration. Codebase composition—specifically Lines of Code (LoC), file counts, and language distributions—was extracted using the open-source software analysis utility *Tokei* under default configurations.

6.1.1 Pre-Migration Polyrepo Baseline

Prior to the intervention, the university's mobile portfolio operated as two completely isolated codebases under a classic polyrepo model. The software metrics of these independent repositories were distributed as follows:

- **Students Application:** Comprised 522 files, totaling 76,745 lines (70,754 lines of raw source code), heavily dominated by TSX (37,237 LoC) and TypeScript (10,236 LoC).
- **Faculty Application:** Comprised 326 files, totaling 57,075 lines (53,488 lines of raw source code), with 30,568 lines of TSX and 2,819 lines of TypeScript.

Summing the individual independent environments yields a total pre-migration baseline of 848 files and 133,820 overall lines (124,242 lines of pure source code).

6.1.2 Post-Migration Consolidated Metrics

Following the migration into the unified `rn-apps` monorepo architecture with an integrated shared library layer (`@polito/lib`), the consolidated ecosystem exhibited a significant scale-down in redundant assets. The post-intervention matrix reveals a total of 744 files and 109,902 overall lines (101,096 lines of active source code).

Table 6.1: Codebase Metrics Comparison Pre- and Post-Migration

Architectural State	Total Files	Total Lines	Source Code (LoC)
Students App (Polyrepo Baseline)	522	76,745	70,754
Faculty App (Polyrepo Baseline)	326	57,075	53,488
Combined Polyrepo Baseline	**848**	**133,820**	**124,242**
Shared Library (`./lib`)	186	13,962	12,593
Unified Monorepo Architecture	**744**	**109,902**	**101,096**
Absolute Net Reduction	**-104**	**-23,918**	**-23,146**

As summarized in Table 6.1, the transition to the monorepo yielded an absolute net reduction of 104 physical files and 23,146 lines of active source code, representing an 18.63% optimization in the active codebase footprint.

This structural optimization is visible within the modern JavaScript and TypeScript application layers (TSX and TS files). In the polyrepo state, the combined

presentational and logical UI layers accounted for 67,805 LoC of TSX and 13,055 LoC of TypeScript. In the unified monorepo, these layers were consolidated into 59,296 LoC of TSX and 10,707 LoC of TypeScript. This net reclamation of 8,509 LoC in TSX and 2,348 LoC in TypeScript provides empirical proof of the successful extraction of the shared Design System, common state contexts, and the complex cross-application "Places" map module into the centralized workspace.

6.2 Distribution Infrastructure Efficiency and Operational Cost Mitigation

The migration from public application stores to a self-hosted, protocol-compliant OTA infrastructure is evaluated along two primary lines: monthly operational costs compared to commercial SaaS platforms, and release propagation latency across production tracks.

6.2.1 Financial Scalability and SaaS Constraint Mitigation

As discussed in Chapter 4, relying on third-party cloud hosting like EAS introduces fixed costs and unpredictable expenses when scaled to an enterprise level. Telemetry data shows that the *Students* application averages 40 000 MAU. At this scale, commercial platforms require a transition to a premium tier (e.g., the EAS *Production* plan), which carries a fixed cost of $199/month [14].

However, the main financial bottleneck lies in the metered infrastructure surcharges for data bandwidth and concurrent update requests. Pushing OTA bundles to tens of thousands of active devices during peak academic windows—such as exam sessions or course registration weeks—generates severe, sudden spikes in network traffic. Because these surges are billed out-of-band and excluded from the flat subscription, they make monthly infrastructure costs highly volatile.

By refactoring the Go-based `expo-open-ota` repository and setting up a local, database-backed Control Plane, this cost model was replaced. The migration eliminated the $199/month subscription fee—yielding a structural savings of at least $2,388 per year-and protected the university from variable bandwidth surcharges.

6.2.2 Release Propagation Velocity and Rollback Recovery

To evaluate deployment agility, propagation velocities were tracked across historical release cycles. In the legacy environment, shipping a critical bug fix or patch required triggering a full Fastlane-driven native compilation block, followed by manual asset ingestion inside the developer consoles of the Apple App Store and Google Play Store.

This traditional framework introduced a propagation velocity bottleneck governed by external store review latencies:

- **Google Play Store Ingestion:** Review loops ranged between 4 to 24 hours for internal testing tracks, extending up to 72 hours for public production releases.

- **Apple App Store Ingestion:** Validation cycles required an average of 12 to 36 hours before native binaries were cleared for customer distribution.

By bypassing store validation mechanisms via the self-hosted OTA platform, code deployment intervals were completely decoupled from external review workflows. When modifications are routed to the asset endpoint of the local server, the propagation velocity scales down from a multi-day window to an immediate runtime handshake.

Upon cold startup, mobile clients evaluate server manifest states within milliseconds via non-blocking background threads. If a validated JavaScript bundle update is discovered, it is retrieved and staged silently. This approach completely compresses hot-fixing intervals from an average of 36 hours down to under 5 minutes.

Furthermore, operational resilience during production failures was evaluated by tracking the execution metrics of the custom `eoas rollback` and `republish` commands. If a breaking regression is introduced into production, the response timeline to reverse the environment state using traditional native stores mimics the initial ingestion review latency (12–72 hours).

Utilizing the database-backed Control Plane, issuing an `eoas rollback` command transmits a stateless directive that triggers an immediate client-side native retreat to the embedded bundle on the next app launch. Alternatively, executing an interactive `republish` routine re-links the historical stable bundle ID at the head of the deployment branch in under 5 minutes. This mechanism provides institutional administrators with near-instant disaster recovery capabilities that protect service availability for both mobile applications.

6.3 Continuous Integration Optimization and Pipeline Efficiency

The engineering impact of shifting from duplicate, siloed continuous integration configurations to the centralized, context-aware orchestrator plane described in Chapter 5 was evaluated by analyzing execution profiles across both architectural states. The primary evaluation metrics targeted pipeline execution duration, resource utilization overhead, and configuration maintainability.

6.3.1 Execution Timeline and Velocity Analysis

Under the legacy polyrepo architecture, the lack of structural delta tracking forced a monolithic execution model. Every commit—regardless of whether it modified native configuration properties or superficial presentational layout strings—triggered parallel, resource-intensive compilation tasks for both mobile platforms. This design introduced a severe performance bottleneck during routine pull request verification cycles.

As established by historical baseline logs, the median execution threshold reached approximately 30 minutes on Android runners and 20 minutes on macOS iOS nodes. Because developers were required to wait for these compilation loops to successfully complete prior to merging code, code integration became highly serialized, significantly dragging down daily team development velocity.

The deployment of the context-aware orchestrator script and the dynamic integration of the `@expo/fingerprint` utility fundamentally altered these performance dynamics. By checking file path intersections and native structural state hashes before launching runner matrices, the pipeline successfully decouples non-native changes from compiler-level constraints. The performance impact of this structural decoupling is categorized into two distinct runtime execution profiles:

- **Native Execution Profile (Fingerprint Mismatch):** When underlying native source structures, package dependency trees, or configuration manifests exhibit a delta, the orchestrator triggers a full native compilation via the centralized Fastlane engine. Under this condition, the pipeline mechanics remain structurally identical to the legacy baseline, as compiling the native binaries cannot bypass hardware and compiler constraints. Because these jobs run in parallel, the total execution wall-time is governed by the slower of the two tracks, evaluating as $\max(30 \text{ min Android}, 20 \text{ min iOS}) \approx 30$ minutes.
- **Bundle Execution Profile (Fingerprint Match):** When modifications are mathematically proven to be restricted to non-native application layers (e.g., pure TypeScript source code, presentational components, or localized static assets), the orchestrator bypasses native compilation entirely. The pipeline routes execution to the lightweight `build-bundle.yaml` sub-workflow. Within this pathway, the operational velocity depends almost exclusively on the execution efficiency of the `expo export` bundling engine. This optimized workflow finishes within an average of 5 minutes.

Empirical tracking of active development cycles indicates that, on average, routine application updates do not introduce infrastructure-level or native architectural alterations. Consequently, for the vast majority of daily pull requests, feature testing iterations, and production deployments, the operational overhead is entirely absorbed by the accelerated bundle pathway.

Pipeline Configuration State	Compilation Pathway	Average Execution Time
Legacy Polyrepo Workflow	Monolithic Native (Android)	$\approx$ 30 min
Legacy Polyrepo Workflow	Monolithic Native (iOS)	$\approx$ 20 min
Context-Aware Orchestrator	Bypassed Bundle (Expo Export)	$\approx$ 5 min
Context-Aware Orchestrator	Enforced Native (max(Android, iOS))	$\approx$ 30 min

Table 6.2: Continuous Integration Pipeline Execution Velocities

As summarized in Table 6.2, execution times for these typical iterations dropped from a parallel wait block bounded at 30 minutes down to a single 5-minute loop. This represents a massive compression in validation and feedback timelines, significantly accelerating the velocity of shipping features through verification gates into both testing and production environments. This reduction in internal continuous integration latency directly complements the external deployment efficiencies explored in Section 6.2.2, where bypassing lengthy app store review cycles via OTA bundle updates similarly compresses total production propagation times from days to minutes.

6.3.2 Configuration Drift and Maintenance Mitigation

Beyond accelerating raw execution speed, the centralization of the automation toolchain into the unified `tools/fastlane` directory eliminated critical infrastructure reliability challenges.

In the legacy polyrepo state, keeping code-signing properties, deployment targets, and marketplace metadata perfectly aligned across separate independent repositories created operational friction. Minor changes to store policies or institutional certificate renewals required manual, simultaneous updates to multiple localized `Fastfile` and `Appfile` structures. This layout made the deployment layer highly vulnerable to configuration drift, where hidden differences between the applications' deployment definitions would cause unexpected build failures on remote runners.

Centralizing this entire framework into an abstraction layer driven by parameterized environment variables completely resolved these structural bugs. Because the two mobile client apps now execute exactly the same shared lanes out of the same directory, the risk of configuration drift is entirely removed. Updates to underlying deployment tools, third-party GitHub Actions packages, code-signing credentials, or Fastlane plugins are written exactly once in the shared configuration, instantly

hardening both mobile applications against infrastructure failures while heavily reducing long-term DevOps maintenance overhead.

Chapter 7

Conclusion and Future Developments

7.1 Conclusions

This thesis work addressed the operational and structural inefficiencies of a fragmented mobile application ecosystem at the Politecnico di Torino by executing a complete architectural migration. The legacy polyrepo model, characterized by high codebase duplication, infrastructure drift, and long continuous integration cycles, was replaced with a unified monorepo architecture integrated with a self-hosted OTA distribution infrastructure and a fast, context-aware continuous integration pipeline.

The quantitative evaluation confirms that the consolidation of the development environments achieved an 18.63% absolute net reduction in the active codebase footprint, successfully reclaiming redundant presentation layers and shared logic into a centralized workspace. Furthermore, the deployment of the context-aware orchestration plane, backed by automated native configuration tracking via `@expo/fingerprint`, fundamentally optimized the integration pipeline. For the vast majority of daily development iterations where changes are strictly confined to the non-native layer, validation feedback loops and deployment intervals dropped from a parallel wait block of 30 minutes down to a 5-minute bundle export cycle. Finally, by transitioning to a self-hosted OTA server, the architecture effectively eliminated commercial platform fees, securing a baseline structural savings of at least $2,388 per year for the university.

The monorepo architecture is now fully operational at the production level. The development team has officially migrated its daily workflows to the unified monorepo framework, which was successfully utilized to orchestrate and deploy the latest official production release of the *Students* application to the public market

stores.

7.2 Future Developments

While the foundational data plane for the OTA server, workspace centralization, and automation pipelines have been established, several engineering steps and iterative features are planned to expand the capabilities and resilience of the ecosystem.

7.2.1 Deployment Roadmap and Infrastructure Integration

The new continuous integration pipeline and the unified application codebase are fully verified and ready for production deployment. The final step toward deployment requires the project maintainer to review and merge the active Pull Request into the primary branch.

The Go-based distribution server will first be packaged and published as an independent, public alpha release, making the new database-backed architecture accessible to the wider open-source community for external testing. Concurrently, an instance of this alpha build will undergo preliminary local deployment inside a dedicated staging environment within the official Politecnico di Torino infrastructure.

Once this local server instance is assigned its authoritative institutional URL, the active client applications will be compiled and published with their updated native versioning, embedding the custom server gateway URIs inside the native `Expo.plist` and `AndroidManifest.xml` configurations. This will initialize the production deployment phase, establishing continuous runtime manifest signing, public certificate validation, and background update downloads across the live institutional network.

7.2.2 Control Plane Evolution and Technical Roadmap

The structural modifications introduced to build the local database-backed Control Plane are currently under technical evaluation by the primary upstream maintainer of the open-source `expo-open-ota` project to validate the architecture for wider community ingestion. Looking ahead, the backend infrastructure will be extended to support advanced enterprise use cases through the following technical developments:

- **Granular Access Tokens:** The initial version of the application-scoped token system operates on a broad, combined read and write privilege model. Future iterations will introduce fine-grained access control lists (ACLs) to decouple these permissions, allowing administrators to restrict automated

analytics services or third-party observability tools to read-only capabilities while isolating write privileges strictly to automated deployment nodes.

- **Lightweight Database Dialects:** To increase deployment flexibility for smaller organizational setups or local development instances, the persistence layer will expand its SQL dialect support. Leveraging the structural isolation within the storage layer, compatibility for embedded engines—specifically SQLite—will be introduced alongside the existing PostgreSQL implementation to allow zero-configuration micro-instances.
- **Automated Canary Deployments:** The infrastructure can leverage its local channel-surfing primitives to automate phased rollouts. Future iterations of the orchestrator plane could integrate with the database control plane to automatically assign a small random percentage of active client requests to an intermediate testing track, monitoring crash analytics and system stability before propagating the OTA update to the entire production user base.
- **Dynamic A/B Testing Infrastructure:** The runtime channel-switching mechanism can be extended to support split-testing methodologies. By interacting with the database-backed control plane, the server will be able to segment the incoming client traffic and simultaneously distribute different JavaScript application bundles to specific target groups, allowing the university to evaluate layout optimizations, feature performance variants, and user adoption rates under real production conditions.

Bibliography

[1] Slinger Jansen, Michael A. Cusumano, and Sjaak Brinkkemper. *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry*. Cheltenham, UK: Edward Elgar Publishing, 2013 (cit. on p. 7).

[2] Robert Potvin and Josh Levenberg. «Why Google Stores Billions of Lines of Code in a Single Repository». In: *Commun. ACM* 59 (July 2016), pp. 78–87 (cit. on p. 7).

[3] Gleison Brito, Ricardo Terra, and Marco Tulio Valente. «Monorepos: A Multivocal Literature Review». In: *Proc. 6th Brazilian Workshop on Software Visualization, Evolution and Maintenance (VEM'18)*. São Carlos, Brazil, Sept. 2018, pp. 1–8 (cit. on p. 7).

[4] Sam Newman. *Building Microservices*. Sebastopol, CA: O'Reilly Media, 2015 (cit. on p. 7).

[5] Ruiyin Li, Peng Liang, and Paris Avgeriou. «Warnings: Violation Symptoms Indicating Architecture Erosion». In: *Information and Software Technology* 164 (Dec. 2023), p. 107319 (cit. on p. 8).

[6] Inc. npm. *npm Workspaces Documentation*. `https://docs.npmjs.com/cli/v9/using-npm/workspaces`. 2023 (cit. on p. 8).

[7] Google. *Bazel Build System*. `https://bazel.build`. 2016 (cit. on p. 8).

[8] Vercel. *Turborepo Documentation*. `https://turbo.build/repo/docs`. 2022 (cit. on p. 8).

[9] Matteo Mazzarolo and Contributors. *react-native-monorepo-tools*. `https://github.com/mmazzarolo/react-native-monorepo-tools`. GitHub repository, accessed 2026-05-27. May 2022 (cit. on p. 9).

[10] Expo Contributors. *Work with Monorepos*. `https://docs.expo.dev/guides/monorepos/`. Expo Documentation, accessed 2026-05-27. May 2026 (cit. on p. 9).

[11] D. L. Parnas. «On the Criteria to Be Used in Decomposing Systems into Modules». In: *Commun. ACM* 15 (Dec. 1972), pp. 1053–1058 (cit. on p. 14).

[12] John Ousterhout. *A Philosophy of Software Design*. Palo Alto, CA: Yaknyam Press, 2021 (cit. on p. 15).

[13] Conventional Commits Contributors. *Conventional Commits*. `https://www.conventionalcommits.org/en/v1.0.0/` (cit. on p. 24).

[14] Expo. *Expo Application Services Pricing*. `https://expo.dev/pricing`. Accessed: 2026-06-10 (cit. on pp. 28, 53).

[15] Expo. *Custom Expo Updates Server Reference Implementation*. `https://github.com/expo/custom-expo-updates-server`. 2021 (cit. on p. 28).

[16] Axel Marciano. *Expo Open OTA: Self-hosted OTA updates for Expo — multi-cloud, production-ready.* `https://github.com/axelmarciano/expo-open-ota`. 2025 (cit. on p. 29).

[17] Brendan Burns. *Designing Distributed Systems*. Sebastopol, CA: O'Reilly Media, 2018 (cit. on p. 30).

[18] Martin Fowler. *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Professiona, 2003 (cit. on p. 30).

[19] J. Clausen. *Channel surfing for Expo Updates: How to switch update channels at runtime*. Expo Blog. Available: `https://expo.dev/blog/channel-surfing-for-expo-updates-how-to-switch-update-channels-at-runtime`. Accessed: 2026-06-25. Jan. 2026 (cit. on p. 40).

[20] G. V. Hulme. *ChatOps: Communicating at the Speed of DevOps*. DevOps.com. Available: `https://devops.com/chatops-communicating-speed-devops/`. Accessed: 2026-07-05. July 2014 (cit. on p. 49).