Politecnico
di Torino

Politecnico di Torino

Master's Degree in Computer Engineering
Academic Year 2025-2026
Graduation Session April 2026

Running coach app

An application to integrate runners' data

Supervisors:
Prof. Maurizio Morisio
Prof. Diego Regruto Tomalino

Candidate:
Yann Villellas

Abstract

The widespread adoption of consumer-grade wearable devices has transformed endurance sports training from intuition-based to data-driven practices. Data has become a key element in training and performance analysis for competitive athletes first, but also for general users. Despite the availability of hardware, a silo problem persists in the software environment: each manufacturer such as Garmin, Coros, Polar, or Suunto, locks users into its ecosystem and confines data into proprietary structures that prohibit users from changing hardware or use multiple devices without losing a global view of their training history. This interoperability problem highlights the need for a neutral platform that can centralize and normalize this heterogeneous data.

This thesis presents a solution by designing and implementing a cross-platform architecture capable of collecting, normalizing, and analyzing data from various wearable devices. Driven by the research community's need for accessible, large-scale datasets, the system introduces a layer of abstraction that decouples athletic metrics from specific hardware limitations. While designed to support an extensible range of providers, the architectural validity is demonstrated through a complete integration with Garmin Connect and a partial integration with Polar, establishing the standardized patterns necessary for future vendor expansions.

The resulting application provides access to activity tracking with detailed time-series metrics (heart rate, pace, elevation, GPS routes), health monitoring (HRV, resting heart rate), performance indicators (VO2 max, lactate threshold), and race time predictions across standard distances, independent of the device manufacturer. Activities are displayed with interactive time-series charts synchronized with GPS route maps, while health and performance indicators are visualized through evolution charts that track trends over different time periods. Unlike the manufacturers' platforms, the application supports multi-device integration, allowing users to use different devices across various sports while preserving a unified training history and effectively preventing vendor lock-in. The key advantage of the application is that performance metrics are calculated from raw data instead of relying on the brand-specific computed values. This allows for consistent comparisons across devices and sports. The resulting dataset constitutes a valuable resource as a foundation for a data-driven coaching.

This work shows that a vendor-neutral mobile application can effectively unify fragmented wearable ecosystems into a coherent experience for athletes of all levels. Moreover, the platform serves sports science researchers by gathering training data across a wide user base. Unlike traditional research models that rely on fixed-period sampling, the platform enables continuous data collection, allowing researchers to adapt study parameters and gather insights in real-time without requiring manual data exports from users. This positions the application as a first step toward a long-term collaboration with the endurance community, where real-world data and user input will drive the development of more accurate and personalized performance models.

Acknowledgements

I would like to express my sincere gratitude to my supervisors, Prof. Maurizio Morisio and Prof. Diego Regruto Tomalino, for giving me the opportunity to carry out this thesis.

I am particularly grateful to Luca Bordino, who guided me throughout the entire project. His availability, expertise, and continuous feedback were invaluable to the success of this work.

Finally, I wish to thank my family, my friends, and my girlfriend for their constant support and encouragement throughout these months.

Table of Contents

List of Figures

Chapter 1

Introduction

Over the past decade, wearables marketed to consumers have changed from specialized equipment for professionals to mass market products that are worn by millions of non-professional athletes. Today, there are a variety of wearable devices (GPS Watches, Heart Rate Monitors, Cycling Computers) from manufacturers like Garmin, Coros, Polar, and Suunto that can capture physiological and performance data in a level of detail that could only be captured in a supervised lab environment prior to the development of these devices. These advances have taken endurance sports training from being primarily based on experience and instinct to a data rich field where athletes can track, measure and compare their training loads, recovery, and progression over time. The result is an enormous and continuously growing corpus of real-world athletic data. Yet despite the maturity of the hardware, the software landscape surrounding these devices remains fragmented and poorly suited to the needs of either independent athletes or sports science researchers: data is siloed within closed manufacturer ecosystems, and even shared file formats such as FIT do not achieve true semantic interoperability across brands. This thesis describes the design, implementation, and delivery of a platform that addresses this gap.

1.1 Background: Wearables and Data

Quantitative metrics are core to the planning of endurance sports training. For example, runners use heart rate zones for their workout intensities, cyclists may use pace and power levels for their workout structures, and athletes measure their resting heart rates and heart rate variability (HRV) on a daily basis to determine how well they have recovered from each day's workouts. Lactate thresholds and VO_2max values are generally considered to be the two most important physiological measures of an athlete's aerobic capacity and ability to progress with endurance training. In the past, these values were measured using clinical instruments and by trained technicians. However, the widespread availability of wearable sensors has changed this. A modern GPS watch records heart rate, pace, elevation, cadence, and GPS coordinates at one-second intervals throughout every activity, while simultaneously monitoring daily resting metrics including sleep quality, step count, and calorie expenditure.

A single activity generates hundreds to thousands of timestamped measurements, and an athlete can accumulate years of such records. The volume of data produced by the collective user base of any popular wearable device is considerable. The type and volume of data produced by athletes is exactly the sort of information that sports science

and researchers needs to develop accurate models of athlete adaptation, performance prediction and injury risk. The democratization of data collection has enabled athletes at all levels to monitor and optimize their training in ways that were previously available only to elite professionals.

1.2 Vendor Lock-in and Interoperability Issues

Despite the richness of data generated by these devices, a fundamental structural problem prevents athletes and researchers from exploiting it. Each wearable manufacturer operates a closed ecosystem in which hardware, application and cloud storage are proprietary. Garmin data resides in Garmin Connect, Polar data in Polar Flow, Coros data in the Coros platform, Suunto data in the Suunto app. Each platform uses its own internal schema for representing activities, health metrics, and performance indicators. There is no universal data standard, and no manufacturer has an incentive to make its data easily movable to a competitor's platform.

The implications for athletes are substantial. Individuals engaged in multisport training (such as triathletes who may employ a Polar device for running and a Garmin unit for cycling) encounter a disjointed data landscape, wherein training histories, workload metrics, recovery indices, and performance trajectories remain siloed across incompatible platforms. More critically, switching hardware manufacturers is effectively penalised. The accumulated training history on the outgoing platform cannot be migrated to the new one. A years-long record of fitness progression, health trends, and performance evolution is held hostage to the original hardware purchase. This is the vendor lock-in problem in its most consequential form.

Partial remedies exist but are insufficient. The FIT file format, developed by Garmin and published as an open standard, defines a binary container in which the schema itself is intentionally flexible: manufacturers are free to define their own message types and data fields within the same format. The result is that FIT is not a proprietary format, but it is also not a semantically interoperable one: each brand's FIT files encode data according to that brand's internal definition, and fields that carry the same nominal meaning often differ in resolution, reference frame, or computation method, not only across vendors but also between successive generations of devices from the same manufacturer. The older TCX and GPX formats suffer from analogous limitations. Taken together, these formats allow individual activities to be exported and re-imported between platforms, but they cover only raw activity data and do not encompass health metrics such as HRV or resting heart rate, nor the computed performance indicators such as VO_2max estimates or lactate threshold. Furthermore, they require manual per-activity export rather than automated programmatic synchronization. Some manufacturers expose web Application Programming Interfaces (APIs) (Garmin through the Garmin Connect API and Polar through the Polar Accesslink API) but these are primarily designed to facilitate third-party integrations within their respective ecosystems. Such APIs are often poorly documented, subject to restrictive rate limits, and lack the architectural intent to enable cross-platform data normalization.

Third-party platforms provide partial solutions but fail to address the core issue comprehensively. Strava, for example, aggregates activity data from diverse sources via Open Authorization (OAuth) integrations and supports a broad array of hardware. However, its primary functionality centers on the social and competitive dimensions of endurance sports, neglecting health monitoring, physiological analytics, and access

to granular time-series data through a client-facing API. Similarly, Apple Health and Google Fit offer device-agnostic health data aggregation at the operating system level but lack the specialized analytical frameworks required for performance metrics such as VO_2max, lactate threshold, or race time prediction. Professional coaching platforms, including TrainingPeaks, accommodate multiple data inputs but are expensive, tailored primarily to athletes under structured coaching plans, and are not designed for athletes seeking device-independent data ownership or for researchers seeking aggregate access to training data.

The limitations of the current landscape are equally acute on the research side. Sports science has historically addressed data collection through fixed-period studies: participants are recruited, instrumented, and monitored for a defined period, after which data is exported manually, cleaned, and analyzed. This model suffers from structural limitations that constrain its scientific utility. Sample sizes are bounded by the logistical cost of participant recruitment and manual data management. Study windows are fixed at design time, preventing researchers from adapting their questions to emerging patterns in the data. The observations reflect a snapshot in time rather than the long-term training trajectory that characterizes real athletic development. Once the study period ends, data collection stops, regardless of whether the research questions have been fully answered.

A platform that continuously collects normalized training data from a growing user base offers a fundamentally different model. Data accumulates passively as users pursue their training. Researchers can define and modify analysis windows without requiring any action from participants beyond initial consent. The dataset grows richer over time as more users connect more devices and accumulate longer histories. Crucially, because all data is normalized to a common schema independent of the originating device, researchers can formulate queries across the entire user base without having to take into account for manufacturer-specific data formats or computed values.

The persistent gap in this landscape is therefore specific and twofold: no existing platform provides a vendor-neutral, multi-device view of training history, health metrics, and performance indicators computed consistently from raw data across all connected devices, nor the research-grade infrastructure for continuous and large-scale data collection that would make such a platform scientifically valuable.

1.3 Motivation

This project originated in a collaboration between the thesis author and a PhD student whose doctoral research centres on data-driven sports performance analysis. That research requires a platform capable of collecting normalized, multi-vendor training data from a real user base at scale and continuously. The application developed in this thesis is designed to serve that need while simultaneously delivering genuine value to the athletes who use it.

The dual purpose is not incidental, it is architecturally necessary. A research data collection platform that offers no consumer value will not attract the user base needed to generate a scientifically useful dataset. Conversely, a consumer application that does not normalize data to a common schema cannot serve cross-device research queries. The design decisions described throughout this thesis were made with both goals held simultaneously, and the tension between consumer usability and research data quality was a recurring driver of architectural choices.

1.4 Proposed Solution

The technological conditions for this project are favorable. The Garmin Connect API and the Polar Accesslink API have reached a level of maturity that makes programmatic integration feasible, even where documentation is incomplete and API behavior must be discovered iteratively. Flutter, the open-source User Interface (UI) toolkit developed by Google and backed by the Dart programming language, enables a single codebase to compile to native Android and iOS applications, substantially reducing the development cost of cross-platform delivery. Firebase provides an integrated identity and authentication layer with mature mobile Software Development Kits (SDKs) and robust environment separation between development and production deployments. The combination of these technologies made it possible to design, build, and deliver a production-ready cross-platform application within the timeframe of a master's thesis.

The application presented in this thesis is a cross-platform mobile application, built with Flutter and targeting Android as its primary delivery platform, that allows athletes to connect wearable device accounts from multiple manufacturers and access a unified, normalized view of their training data independent of the originating device. Users authenticate through Firebase Authentication and connect their wearable accounts via OAuth 2.0 flows mediated by a RESTful backend. The backend is responsible for all provider-specific logic: it communicates with each manufacturer's API, retrieves activity and health data, normalizes it into a common schema, and computes performance metrics from raw values rather than from the manufacturer-provided computed fields. The Flutter client consumes only normalized data and contains no knowledge of individual provider formats or API behaviors.

This separation between provider-specific ingestion and normalized presentation is the central architectural contribution of the work. By locating all normalization and computation logic in the backend and exposing only standardized data to the client, the architecture ensures that performance metrics such as VO_2max, lactate threshold, and race time predictions are computed consistently regardless of which device produced the underlying data. A user who trains with both a Garmin and a Polar device sees their metrics evaluated by the same algorithm applied to the same normalized inputs, rather than comparing one manufacturer's proprietary fitness estimate against another's. This consistency is both a consumer benefit and a prerequisite for the cross-device research use case. The following objectives define what the thesis set out to achieve within that solution.

1.5 Research Objectives

This thesis pursues five research objectives. The first is to design and implement a cross-platform mobile architecture capable of collecting, normalizing, and presenting wearable training data from multiple heterogeneous providers within a single unified application. The second is to validate that architecture through complete integrations with both the Garmin Connect API and the Polar Accesslink API, establishing the normalization patterns and provider abstraction layer that future integrations for Coros and Suunto can follow. The Garmin integration was validated end-to-end against physical hardware. The Polar integration is fully implemented at the code and API contract level but could not be exercised against a physical device due to hardware unavailability during the testing phase. The third is to derive performance metrics

(VO_2max, lactate threshold, and race time predictions) from raw normalized data rather than from brand-provided values, enabling consistent cross-device comparisons that are not possible within any single manufacturer's ecosystem. The fourth is to build a continuous research data infrastructure into the platform from the outset, enabling passive, longitudinal collection of training data from a growing user base to support the research without the fixed-period sampling constraints of traditional study designs. The fifth is to demonstrate a complete production-grade continuous integration and continuous deployment (CI/CD) pipeline for a Flutter Android application, covering automated testing, build flavor management, signing key handling, and deployment to Google Play through GitHub Actions and Fastlane.

1.6 Scope

This thesis focuses on the design, implementation, and delivery of the mobile client for the Android platform, which is the primary delivery target. The iOS platform is architecturally supported by the Flutter codebase but is not part of the delivered scope of this work. The backend, comprising the RESTful API, database, normalization pipeline, and metric computation logic, is outside the scope of this thesis. It is described at the level of its API contract where necessary to explain client behavior, but its internal design is not examined here.

Provider integration is implemented fully for Garmin Connect and Polar. The extensibility patterns established by these two integrations are designed to support the addition of Coros and Suunto in future work without requiring structural changes to the core client architecture or data models. Real-time activity tracking, social features, AI-driven coaching recommendations, and structured training plan management are explicitly out of scope for this version. The analysis of data collected through the research infrastructure is the subject of future doctoral work and falls outside the scope of this thesis.

A measurement quality caveat applies throughout: the accuracy of the metrics presented in the application is bounded by the quality of the raw data provided by the wearable device. The platform normalizes and recomputes metrics from whatever data the device reports. It cannot correct for sensor noise, GPS drift, or the limitations of optical heart rate measurement. Users should interpret metrics in the context of the capabilities and limitations of their specific hardware.

Chapter 2

Requirements

The scope of this project, established in the introduction, is dual in nature: a consumer-facing mobile application for athletes and a research data infrastructure serving sports science. These two axes produce two families of requirements that must be held simultaneously throughout the design process. The functional requirements govern what the application does from a user perspective, while the research data requirements govern how data must be structured, collected, and exposed for scientific use. The non-functional requirements establish the constraints within which both must operate. This chapter formalizes all three families, explains how they were elicited, and describes the agile development methodology used to evolve and prioritize them throughout the project.

2.1 Requirements Elicitation

The core requirements of this project were established at the beginning. The goal was to create a mobile application for athletes to track and visualize their training data, and a backend data infrastructure supporting longitudinal sports science research. Some lower-level details such as which specific metrics could be reliably retrieved from each wearable provider and the exact shape of certain data screens were refined progressively as implementation progressed. This is expected in any project involving third-party integrations and evolving research needs, and it informed the choice of an agile methodology rather than a fixed-specification process.

Requirements were gathered through three principal mechanisms. The first was a series of bi-weekly sprint meetings with the PhD student collaborator, who acted as product owner throughout the project. At each sprint boundary, completed features were demonstrated, blockers were discussed, and the next sprint's scope was agreed upon collaboratively. This ensured that the research data requirements remained aligned with the PhD student's evolving understanding of what the dataset needed to look like. The second mechanism was comparative analysis of existing platforms: Garmin Connect and Strava were studied to identify the gaps that the proposed platform was designed to fill, as described in the introduction. The third was incremental discovery during implementation itself, which is discussed further in the context of the agile methodology in Section 2.5.

Prior to implementing any screen, Figma mockups were produced and reviewed with the PhD student collaborator at the bi-weekly meeting. This requirement validation step allowed intended user flows, screen layouts, and navigation structures to be

agreed upon before any code was written. Validating at the mockup level, rather than after implementation, substantially reduced the risk of rework caused by misaligned expectations, and it reinforced the principle that requirements must be validated by stakeholders before they are treated as finalized. Stakeholder personas were identified as part of the elicitation process and were used to check that each functional requirement served at least one concrete user need before it was admitted to the backlog.

2.2 Stakeholders

Two principal stakeholders were identified. The primary stakeholder group is the end user: athletes of all levels who wear consumer-grade devices and wish to access a unified, device-independent view of their training history, health metrics, and performance progression. This group spans casual recreational runners using a single device through to competitive multisport athletes who own hardware from more than one manufacturer and require a single platform that aggregates their data coherently.

The secondary stakeholders are the PhD student and his research group, who serve as the project's Product Owners and represent the interests of the sports science study. Because the longitudinal dataset accumulated by this application forms the empirical foundation of his doctoral work, he has a direct stake in both the research data infrastructure and the consumer-facing features that drive participant engagement. His research needs determine which metrics must be collected, how data must be normalized, and what analysis flexibility the backend must preserve. This dual role made his involvement natural: at every sprint boundary meeting he validated scope and priorities from both a research quality and a user utility perspective, ensuring that implementation decisions served both dimensions simultaneously. The developer occupies the additional role of sole engineer on the project. The single-person team context is significant, as it justifies a lightweight agile process adapted from Scrum rather than a full Scrum implementation with dedicated roles and ceremonies, as discussed in Section 2.5.

2.3 Functional Requirements

2.3.1 Authentication and User Management

The application must support account creation and login via both email/password and Google Sign-In. First-time users must be guided through an onboarding flow that collects the minimal profile information required to associate training data with a named identity. Users must be able to view and edit their profile at any time.

2.3.2 Provider Integration

Provider integration is the most architecturally significant functional requirement area. Athletes must be able to connect their wearable device accounts through the standard OAuth 2.0 authorization flow, after which the platform retrieves their data automatically without requiring manual exports or repeated authentication. The authorization process must leave the user in a native application context before and after granting access, with no manual steps required beyond approving the connection on the provider's website.

Garmin must be fully supported, covering the complete set of data types described in the subsequent sections. A single user must be able to connect accounts from multiple manufacturers simultaneously and must be able to connect, disconnect, and reconnect each provider independently. The integration design must accommodate Coros and Suunto as future providers without requiring changes to the core client data model.

2.3.3 Activity Management

The application must display all recorded activities across all connected providers, with summary fields including activity name, date, sport type, total distance, total duration, average pace, a user-defined category, and any attached notes. Each activity must be navigable to a details screen that exposes the full time-series data recorded during the session.

The activity details screen must present heart rate, pace, and elevation as interactive time-series charts, and the GPS route on an interactive map. Selecting a point on any chart must highlight the corresponding GPS position on the map, and selecting a position on the map must update the chart positions accordingly. Activities must support user-defined categories extending beyond the sport type provided by the device manufacturer, and must support free-text notes for qualitative annotations.

2.3.4 Health Metrics

The application must aggregate the following daily physiological indicators across a configurable time window: resting heart rate, heart rate variability in both its last-night and resting forms, minimum and maximum daily heart rate, active and total kilocalories, total daily step count, and total daily distance. Each metric must be filterable to at least a weekly, monthly, and all-time view. HRV display must be conditional on data availability, since not all providers expose both HRV variants.

2.3.5 Performance Metrics

The application must present the three principal indicators of aerobic fitness: VO_2max, lactate threshold heart rate, and threshold pace in minutes per kilometre. These values must be derived from raw data normalized to a device-independent representation, not from brand-specific computed values exposed by manufacturers. Each metric must be accompanied by an evolution chart showing how the value changes over time.

2.3.6 Race Predictions

The application must present estimated finish times for the 5 km, 10 km, half marathon, and marathon distances. These predictions must be computed from normalized performance data, not from any manufacturer's proprietary model. An evolution chart must track how predicted times change as the athlete's training progresses.

2.3.7 Research Data Infrastructure

The research data requirements are architecturally distinct from the consumer-facing requirements above in that they impose constraints on the backend infrastructure rather

than on the mobile client. They are recorded here because they shape the normalization strategy that the client and backend share.

Data collection must be continuous and passive: once a user has connected a provider, their data must be retrieved and normalized by the backend without any further action required from the user. The dataset must cover three categories: activity time-series data including heart rate, pace, elevation, and GPS coordinates; daily health metrics including HRV, resting heart rate, step count, and calorie expenditure; and derived performance indicators including VO_2max, lactate threshold, and race time predictions. All data must be normalized to a common schema independent of device manufacturer. Raw sensor values must be preserved without pre-aggregation. The platform must not impose a fixed study period: researchers must be able to define and adjust their analysis windows at any time after data collection has begun.

2.4 Non-Functional Requirements

Non-functional requirements were not tracked as discrete project management issues. They were instead enforced through architectural decisions and treated as constraints that any implementation must satisfy. They are stated explicitly here to make the design rationale traceable in subsequent chapters.

2.4.1 Performance

The application must render smoothly on mid-range mobile hardware, with no perceptible lag during navigation or chart interaction. Redundant network requests must be avoided. Interactive charts must respond immediately to user gestures, including the chart-to-map synchronization on the activity details screen.

2.4.2 Scalability

Adding a new wearable provider in the future must not require changes to the core client data model or the normalization schema. The platform must support multiple simultaneous users without data isolation issues, as it is intended to serve a growing research cohort over time.

2.4.3 Security

No credentials, API keys, or environment-specific configuration may be stored in the source repository. Production and development environments must be strictly isolated: a production build must not be able to communicate with a development backend. OAuth tokens must never be stored or handled by the mobile client.

2.4.4 Usability

A new user must be able to create an account and connect a wearable provider without requiring any external documentation. The navigation structure must be immediately legible, and charts must remain readable and interactive on small mobile screens.

2.4.5 Maintainability

The codebase must be structured so that a new contributor can locate and modify any feature without needing to understand the full system. This is a concrete requirement given that other students will continue development beyond this thesis.

2.4.6 Reliability and Quality

No code may reach a deployable build artifact without passing the automated test suite. Builds must be reproducible: the same source must always produce the same binary regardless of when or where the build is triggered.

2.4.7 Cross-Platform Compatibility

Android must be the primary delivery target. The codebase must not make Android-specific assumptions that would block a future iOS release, as iOS is explicitly identified as a near-term continuation target beyond this thesis.

2.4.8 Data Consistency

All metrics must be expressed in consistent, device-independent units across all providers, so that cross-device comparisons are valid and research queries over the aggregate dataset require no unit conversion. Performance metrics must be derived from this normalized representation rather than from manufacturer-specific computed values.

2.5 Agile Development Methodology

2.5.1 Rationale for an Agile Approach

The choice of an agile development methodology was not arbitrary. While the high-level scope of the project was clear from the outset, two structural characteristics made a fixed-specification process unsuitable. First, the project involved a product owner whose understanding of which metrics were most valuable for the research evolved as the application took shape. Requirements that seemed settled on paper sometimes needed to be revisited once a working screen made the implications concrete. Second, the project combined consumer-facing product development with academic research goals, two domains in which feedback loops are naturally iterative: a screen that looks reasonable in a Figma mockup may reveal usability issues only when tested on a real device, and a metric that seemed important from a research perspective may prove less tractable once real data is examined. A waterfall process would have required both the product and research requirements to be fully specified before any implementation began, which would have generated rework at every subsequent point of discovery.

Agile addressed both of these characteristics directly. The bi-weekly review cadence created a structured opportunity to demonstrate working software to the product owner at regular intervals, allowing research priorities to be re-aligned incrementally rather than all at once at the end. Figma mockups validated screen layouts and user flows before any code was written, reducing the risk of building the wrong thing. Incremental delivery meant that each sprint produced a testable increment, making it possible to catch misalignments early when the cost of changing direction was lowest.

2.5.2 Process: Scrum-Inspired with Adaptations

The development process was adapted from Scrum [1] to fit a single-developer context. The full Scrum framework specifies distinct roles (Product Owner, Scrum Master, Development Team), ceremonies (daily standups, sprint planning, sprint review, retrospective), and artifacts (product backlog, sprint backlog, increment). In a single-developer project, several of these structures collapse naturally: there are no daily standups, and the Product Owner and Scrum Master roles cannot be filled by separate individuals. The adapted process retained the elements that provided genuine coordination value and discarded those that would have imposed overhead without benefit in a one-person team.

The sprint length was two weeks throughout the main development phase. The PhD student collaborator acted as product owner, validating scope alignment with research goals at every sprint boundary. Each bi-weekly meeting combined the sprint review (a demonstration of completed features), the retrospective (a discussion of blockers and lessons learned), and the planning session for the next sprint. This consolidation of three Scrum ceremonies into a single meeting was appropriate for the scale of the project and ensured that none of the three functions was omitted while avoiding the overhead of holding them as separate events. Velocity was tracked informally by counting the number of YouTrack issues closed per sprint, which was sufficient for collaborative prioritization without requiring a formal estimation framework. Once the core backlog was largely complete, the final sprint focused on stabilization.

2.5.3 Sprint Structure

Each two-week sprint followed a consistent structure. At the start of the cycle, backlog issues were reviewed, estimated, and assigned to the sprint in YouTrack. The sprint had an explicit goal expressed as a statement of the highest-priority outcome to be achieved: for example, completing the Garmin activity detail integration, implementing the metrics screen with evolution charts, or establishing the CI/CD pipeline. During the sprint, development proceeded daily, with issues moved across the YouTrack board (Open, In Progress, To Verify, Done) as work progressed. Each feature was developed on a dedicated branch and merged to the `dev` branch via a pull request on GitHub. The `dev` branch served as the integration branch throughout development and was merged into `main` to produce release candidates.

Figma mockups played a specific role in the sprint structure. Before implementing any screen, mockups were produced and presented to the PhD student collaborator at the sprint meeting preceding the implementation sprint. This validation step converted the screen layout and user flow from a developer assumption into a collaboratively confirmed requirement before any code was written. The Figma-based validation workflow is a requirements engineering practice in the agile sense. It is how the right thing was confirmed before any code was written, and it is described in further detail in the UI chapter.

A feature was considered Done when it passed the automated CI test suite, built successfully for Android, and was demonstrated to the product owner at the sprint meeting without any blocking feedback. This definition of Done was applied consistently across sprints and ensured that issues were not closed on the basis of local developer testing alone.

2.5.4 YouTrack as Project Management Tool

JetBrains YouTrack [2] was used as the agile board and issue tracker during the main development sprints. Each issue represented a unit of work: a user story, a technical implementation task, a bug, or a research-oriented investigation. Issues were organized into epics corresponding to the major feature areas of the application. This epic structure made it possible to assess progress at a feature-area level as well as at the individual issue level, and to communicate priorities to the PhD student collaborator in terms of feature delivery rather than individual task counts.

The YouTrack board used the standard workflow columns (Open, In Progress, Done) and additionally included a "To Verify" column so reviewers could quickly identify pull requests and associated features requiring review. Sprint assignments made the scope of each two-week cycle explicit: at any moment, the set of issues assigned to the current sprint defined exactly what was in scope and what was not, providing a clear boundary against scope creep. The backlog was maintained with relative prioritization, with the highest-risk and highest-value items scheduled first. As the project progressed and the core backlog was exhausted, the process naturally moved away from formal issue tracking and towards direct iterative delivery, which is consistent with the transition described in Section 2.5.2.

2.5.5 Backlog and Prioritization

Backlog items were sequenced by a combination of risk and value. No formal prioritization framework such as MoSCoW or weighted shortest job first was applied; the single-developer context and comparatively small backlog made informal collaborative sequencing with the PhD student collaborator sufficient. The general sequencing principle was to schedule the highest-risk and most architecturally foundational items first, so that incorrect assumptions were discovered as early as possible when the cost of changing direction was lowest.

Sprint 1 was dedicated to infrastructure and exploration: initializing the Flutter project and Firebase environments, requesting API access for all target providers, and producing the first round of Figma mockups for stakeholder validation. Sprint 2 established the CI/CD pipeline and delivered the first mocked screens with the bottom navigation bar, validating the navigation skeleton before any real data integration was attempted. Sprint 3 completed the remaining mocked screens (activity list, activity details, dashboard, profile) and began authentication work with Firebase login. Front-loading a fully mocked UI skeleton in this way enabled end-to-end user flow validation before real data was integrated, deliberately separating API uncertainty from UI uncertainty so that neither source of risk amplified the other.

Sprint 4 introduced the REST API layer, real-data dashboard charts, backend authentication with Firebase JSON Web Token (JWT), and the Model–View–ViewModel (MVVM) architecture refactor. Sprint 5 completed the Garmin OAuth integration, the highest-risk provider task, which had been deliberately scheduled after the mocked UI was validated and the architecture was in place. Sprint 6 served as a feature completion phase, adding activity editing, the weekly mileage view, the GPS route map, and conditional HRV display. The Polar integration was implemented following the same architecture and subsequently deferred for end-to-end testing. Coros and Suunto were represented in the model layer but explicitly deferred pending API access confirmation that was not received within the project timeframe.

2.5.6 Adaptive Planning in Practice

A concrete instance of adaptive planning occurred during the integration of performance metrics. The performance metric charts were designed to show the last seven days of data, but in the beta cohort some testers trained only once or twice per week, in those cases the chart often contained only one or two points, making the trend line meaningless. This issue only surfaced once real users with varied training cadences started using the app, at which point the chart was updated to expose the full thirty-day series returned by the backend and the rendering was adjusted to remain readable with sparser data. This kind of discovery is precisely the scenario that the agile review cadence is designed to surface early.

The scope of the CI/CD pipeline proved more complex than initially estimated, expanding to cover Android signing key management, Firebase configuration injection for two environments, build flavor handling, and Fastlane-based Google Play deployment. This justified the decision to establish the pipeline early in sprint 2: every feature merged from sprint 3 onwards was automatically subject to the quality gate, rather than having to retrofit compliance onto already-merged code.

Chapter 3

Application Design

This chapter describes the architectural and design decisions made in developing the Endurance App mobile client. The chapter covers the overarching Model–View–ViewModel (MVVM) pattern, dependency injection (DI) strategy, error handling, data modeling, state management, navigation, environment separation, authentication, third-party provider integration, backend API contract, user interface architecture, and the utilities supporting them. Each section describes not only what was implemented but why the chosen approach was preferred over the available alternatives, grounding design decisions in the functional and non-functional requirements established in Chapter 2.

3.1 Architecture Overview

The application follows the Model–View–ViewModel (MVVM) architectural pattern. MVVM was chosen over alternatives such as Business Logic Component (BLoC) or Model–View–Controller (MVC) for three reasons. First, it aligns naturally with Flutter's reactive widget model: a ViewModel exposes observable state and the View rebuilds itself when that state changes, which mirrors how Flutter's widget tree already works. Second, it achieves a clean separation of concerns without the boilerplate overhead of the BLoC pattern, which is significant for a project developed by a single developer. Third, the official Flutter documentation endorses MVVM with the `provider` package [3] as the reference approach for state management in production applications [4].

The architecture is organized into four layers, each with a single responsibility. The **Service layer** provides stateless wrappers around external systems. Five of the six service classes (`ActivityService`, `MetricsService`, `UserService`, `IntegrationService`, and `CategoryService`) each hold an `http.Client` and translate raw HTTP responses into Data Transfer Objects (DTOs). The sixth, `AuthService`, holds no `http.Client`; it delegates entirely to the Firebase Authentication and Google Sign-In SDKs and returns their results directly. Services contain no business logic and no state. The **Repository layer** sits above the service layer and is defined by abstract interfaces with concrete implementations. Each abstract repository declares the operations available to higher layers; the concrete implementation performs the mapping from service-level DTOs to domain models and centralizes all error-handling translation for that domain. The **ViewModel layer** sits above the repository layer and is the exclusive owner of UI-visible state. Each ViewModel extends `ChangeNotifier`, calls repository methods, updates its internal state, and calls `notifyListeners()` to trigger rebuilds. ViewModels contain no widget code and no direct HTTP logic. The **View layer** comprises

Flutter widgets. Views read ViewModel state through `Provider.of` or `context.watch`, invoke ViewModel methods on user interaction, and contain no business logic beyond layout decisions.

The strict unidirectional dependency rule is: Views depend on ViewModels, ViewModels depend on Repositories, Repositories depend on Services, and Services depend on external systems. No layer references a layer above itself, and no layer skips a layer below itself. This discipline makes each layer independently testable and replaceable. Figure 3.1 illustrates the complete layer structure and the data flow between them.

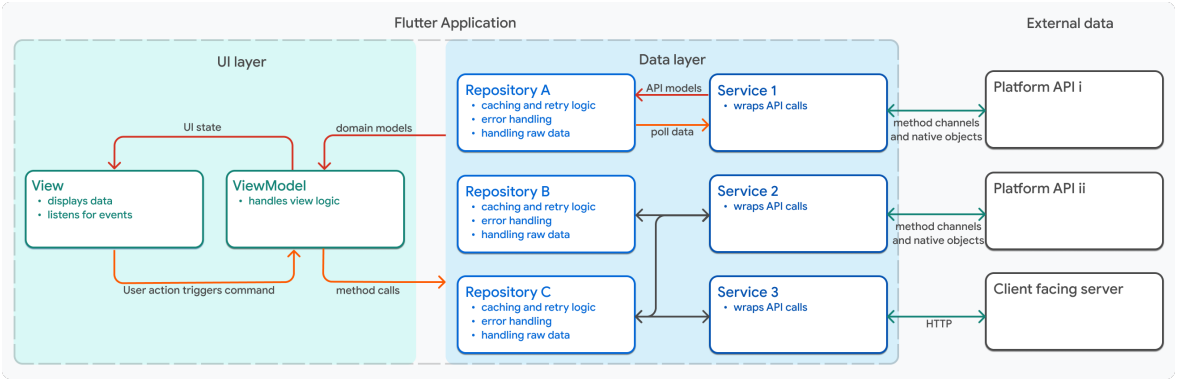


Figure 3.1: MVVM layer structure: the UI layer (View and ViewModel) communicates with the data layer (Repository and Service), which in turn depends on external systems. Source: [4].

3.2 Dependency Injection

Dependency injection (DI) is a technique for managing how the components of an application obtain the objects they depend on. Rather than a class creating its own dependencies internally, those dependencies are constructed externally and passed in, typically through constructor parameters. In this application, a ViewModel does not instantiate its own repository. The repository is built once at application startup and handed to the ViewModel. The practical consequence is that each component only knows the interface of what it receives, not the concrete implementation, which makes it straightforward to substitute one implementation for another without touching the component itself.

Three approaches were considered. With direct instantiation, each class constructs its own dependencies internally, which reduces initial setup but tightly couples layers together, making it impossible to replace a real network service with a mock without modifying the class itself. A service locator provides a global registry from which any class can retrieve its dependencies on demand, which reduces the amount of wiring code required but makes the dependency graph implicit and hidden, harming readability. Explicit DI with the `provider` package [5] was chosen because the full wiring is declared in one place (`AppProviders`), every dependency is traceable from a constructor signature alone, and swapping a real implementation for a mock requires changing a single line with no impact on any consuming class, a direct benefit for the automated test suite described in Chapter 4.

`AppProviders` wires the entire dependency graph into a `MultiProvider` widget placed above `MaterialApp.router` in the widget tree. The wiring follows a three-level

hierarchy that mirrors the layer structure described in Section 3.1: services are registered first, repositories above them receiving their service dependencies, and ViewModels at the top receiving their repository dependencies. Each HTTP service accepts an injectable `http.Client` that defaults to a real client when none is provided, which is the mechanism that allows tests to substitute a mock without modifying the service itself. Services that hold an HTTP connection register a dispose callback so the connection is released cleanly when the provider is removed from the tree.

This choice also has a known trade-off: dependency wiring is more verbose because all bindings are declared explicitly in one place. However, that verbosity is intentional and beneficial in this context: the dependency graph is visible at a glance in `AppProviders`, the creation and cleanup of every object is controlled centrally, and mocks can be injected without changing feature code. For a single-developer codebase expected to evolve incrementally, this improves maintainability and reduces the risk of unintended regressions.

3.3 Error Handling Strategy

3.3.1 Result Type

Error handling in application code can follow two approaches. The conventional approach is to throw exceptions on failure and catch them at a higher level. The problem with exceptions in a layered architecture is that they are invisible in the type signature: a caller has no indication that a function can fail, the compiler does not require the error branch to be handled, and an uncaught exception propagates silently up the call stack. The `Result<T, E>` type addresses this by encoding the outcome directly in the return type, making failure a visible and compiler-enforced part of the contract between layers.

All fallible operations in the service and repository layers return a `Result<T, E>` type rather than throwing exceptions. `Result` is a sealed class with two subclasses: `Success<T, E>` carrying the value and `Failure<T, E>` carrying the error. The sealed keyword renders the exhaustive `switch` expression possible: the compiler rejects any switch that does not handle both cases, which eliminates the possibility of silently ignoring failure paths. For example, a repository method returning `Result<List<Activity>, AppError>` forces every caller to handle the error branch; there is no mechanism by which the caller can access the activity list without first acknowledging the possibility of failure.

The `Result` type exposes convenience methods: `map` transforms the success value, `mapError` transforms the error value, `onSuccess` and `onFailure` provide side-effect callbacks, and `getOrDefault` provides a fallback value without requiring a full switch. These combinators allow ViewModels to chain result operations without nesting multiple switch expressions.

3.3.2 AppError Hierarchy

`AppError` is a sealed class hierarchy root. The four concrete subtypes are:

- `AuthError` carries an `AuthErrorType` enum value encoding the specific Firebase Authentication error (invalid credential, user not found, requires recent login,

email already in use, and others). Factory constructors on `AuthError` translate Firebase error codes into the typed enum.

- `NetworkError` carries a human-readable `message` and a boolean `canRetry` flag indicating whether the operation that failed is safe to retry. `canRetry` is a computed property that returns `true` only when the error represents a timeout or a lost network connection; all other `NetworkError` instances, including server-side errors, have `canRetry = false`.
- `ValidationError` carries a `message` describing a client-side validation failure, such as a malformed API response.
- `UnknownError` is a catch-all for unclassified exceptions, carrying the original stack trace.

Modeling errors as a sealed type rather than a string or integer code means that every error-handling site can dispatch on subtype using an exhaustive switch, with the compiler enforcing that all cases are handled.

3.3.3 ErrorHandler

`ErrorHandler` is a static utility class that translates `AppError` values into user-visible UI feedback. It dispatches on the error subtype and responds appropriately. A `NetworkError` with `canRetry = true` shows a dialog with a Retry button whose callback is passed to the handler. A `NetworkError` with `canRetry = false` shows an informational dialog without a retry option. An `AuthError` shows a `SnackBar` with the error message. A `ValidationError` shows a neutral `SnackBar`. An `UnknownError` shows a generic error dialog. A `_dialogOpen` flag prevents multiple dialogs from stacking when multiple errors arrive in rapid succession.

By centralizing this dispatch in one class, every ViewModel can pass errors to `ErrorHandler.handle(context, error, onRetry)` without duplicating the snackbar or dialog construction logic.

3.4 Data Modeling

3.4.1 Three-Tier Model Structure

The application separates its data representation into three tiers, each with a distinct responsibility. DTOs represent the JSON wire format of the backend API. Each DTO is an immutable class with `fromJson` and `toJson` methods. DTOs contain no business logic and are never exposed above the repository layer. Domain models represent the business entities used throughout the ViewModel and View layers. Each domain model is `@immutable` and is constructed from a DTO via a `fromDto` factory constructor that performs any necessary unit conversion (for example, converting distance from metres to kilometres, or converting a duration in seconds to a `Duration` object). Domain models expose only the fields that the application actually uses, discarding fields present in the DTO that serve no purpose in the current client. Presentation models are thin view-specific structures that bundle the display-ready strings and values needed to

render a single widget. They are constructed in the ViewModel from domain models and consumed directly by the widget tree without further transformation.

This separation ensures that changes to the backend JSON schema affect only the DTO and its `fromJson` factory, that changes to business logic affect only the domain model, and that changes to how a value is displayed affect only the presentation model. None of these changes propagate upward unnecessarily.

3.4.2 Activity and ActivityDetails Split

To keep the activity list responsive and memory-efficient, the model layer distinguishes between a lightweight summary and a full-detail representation. `Activity` contains the fields needed for the activity list view: identifier, name, date, distance in kilometres, duration, pace in minutes per kilometre, sport type, category, and notes. `ActivityDetails` adds time-series measurements (`List<Measurement>`), a GPS route (`List<RoutePoint>`), average and maximum heart rate, average cadence, perceived effort, subjective feeling, and the originating device and provider. The split exists for a performance reason: deserializing, storing in memory, and diffing a full time-series payload for every activity in a long list would impose unnecessary memory and CPU cost. The `ActivityViewModel` fetches `Activity` objects for the list and never triggers a full details fetch until the user navigates to a specific activity. The `ActivityDetailsViewModel` then fetches only the one `ActivityDetails` object needed for the open screen.

3.5 State Management

3.5.1 ViewModel Responsibilities

Each ViewModel has a clearly bounded responsibility corresponding to one feature domain. `AuthViewModel` manages the authentication lifecycle: sign-in, sign-out, registration, email verification polling, and password-sensitive profile operations. The paginated activity list, including pagination state, search filtering, and cache clearing on sign-out, is managed by `ActivityViewModel`. For a single open activity, `ActivityDetailsViewModel` manages the full details, including the editable form state for perceived effort, subjective feeling, category, and notes, as well as the chart series selection and X-axis type. `MetricsViewModel` manages health metrics, performance metrics, race predictions, and weekly mileage. `UserViewModel` and `ProfileViewModel` manage user profile loading and editing respectively. `IntegrationViewModel` manages the list of provider integration statuses and the connection and disconnection flows.

Every ViewModel exposes loading, refreshing, and error flags as public getters alongside the data payload. Views bind these flags to shimmer placeholders, progress indicators, and error states without containing any conditional logic beyond widget selection.

3.5.2 ViewModel Lifecycle

Most ViewModels are constructed by `ChangeNotifierProxyProvider` in the provider tree and are scoped to the lifetime of the widget subtree they are registered in, which in this application is the entire app. The exception is `ActivityDetailsViewModel`,

which is instantiated directly in `ActivityDetailsScreen.initState()` by reading its repository dependencies from the provider tree with `context.read`. Its lifetime is therefore scoped to the activity detail screen rather than the entire app. The `dispose` method of each ViewModel cancels any active stream subscriptions and closes any open resources. ViewModels that hold an auth-state listener cancel the subscription in `dispose` to prevent memory leaks after the ViewModel is removed from the tree.

3.5.3 Authentication State and Cache Clearing

In their constructors, both `ActivityViewModel` and `MetricsViewModel` subscribe to `AuthRepository.authStateChanges`, which proxies the underlying `FirebaseAuth` stream through the repository abstraction. When the stream emits `null` (signalling sign-out), these ViewModels clear their cached data and reset their pagination state. This prevents a scenario where a second user signs in after the first signs out and sees the previous user's data displayed while the first fresh fetch is in flight. Clearing the cache on sign-out rather than on sign-in ensures that the window of exposure is zero: the data is gone at the moment the session ends, not after the new session begins.

3.5.4 Loading and Refresh State Split

Two distinct in-flight flags are exposed by each caching ViewModel: `isLoading` and `isRefreshing`. `isLoading` is set by `loadActivities()` and `fetchAllMetrics()`, the initial fetch paths. `isRefreshing` is set by `refreshActivities()` and `refresh()`, the pull-to-refresh paths. The distinction drives a deliberate user experience (UX) difference: when `isLoading` is `true` the view replaces the entire content area with shimmer placeholder cards, because there is no previously fetched data to keep visible; when `isRefreshing` is `true` the `RefreshIndicator` spinner is shown at the top of the existing list, which remains visible and interactive. This prevents the jarring experience of the full list disappearing and re-appearing on every pull-to-refresh.

A third flag, `_hasLoadedOnce`, is set to `true` after the first successful fetch and reset on sign-out. It is not used for shimmer rendering; its sole purpose is to scope the Retry action when an error occurs. If an error is raised while `hasLoadedOnce` is `false`, the Retry callback invokes `globalFetch`, which reloads both `ActivityViewModel` and `MetricsViewModel` together. If `hasLoadedOnce` is `true`, the Retry callback reloads only the failing ViewModel. This ensures that a failed initial load of either ViewModel does not leave the application in a half-loaded state where, for example, metrics are visible but the activity list remains empty.

3.6 Navigation

3.6.1 Declarative Routing with go_router

Navigation in a mobile application refers to the mechanism that controls which screen is displayed and how transitions between screens are managed. `go_router` [6] is the routing package officially recommended by the Flutter team. Flutter's built-in imperative `Navigator` API, accessed through methods such as `Navigator.push` and `Navigator.pushNamed`, was not suitable because it does not natively handle incoming deep links from the operating system, requiring manual Uniform Resource Identifier

(URI) interception and parsing code. `go_router` was chosen because it provides declarative URL-based routing that handles deep links natively [7] and supports type-safe route parameter extraction, both of which are required for the OAuth callback flow described in Section 3.6.4.

Route path constants are centralized as static string constants in `AppRoutes`, with a helper method for parameterized paths:

```
class AppRoutes {
  static const String activityDetails = '/activity/:id';
  static String activityDetailsPath(String id) => '/activity/$id';
}
```

This prevents magic-string references to route paths from proliferating through the codebase: every navigation call references `AppRoutes.activityDetails` rather than a raw string literal.

3.6.2 Authentication Gate

Authentication state is managed by `_AuthDataGate`, a widget that subscribes to `FirebaseAuth.instance.userChanges()` via a `StreamBuilder`. This stream emits on every authentication state change: sign-in, sign-out, token refresh, and property changes including email verification status updates after `user.reload()`. Depending on the emitted state, `_AuthDataGate` renders a loading indicator while the stream is settling, `SignInScreen` when unauthenticated, `EmailVerificationScreen` when the user is authenticated but has not verified their email address (excluding Google Sign-In users who are pre-verified by Firebase), or `_DataGate` when the user is both authenticated and verified.

`_DataGate` is a separate `StatefulWidget` that, after its first frame is built via `WidgetsBinding.instance.addPostFrameCallback`, schedules a deferred callback. It then invokes `UserViewModel.loadUser()`. If the resulting user object does not exist in the backend (first login after registration), the gate navigates to `OnboardingScreen`. If an existing user is returned, it renders `HomePage`. This two-gate pattern separates the authentication concern from the data-loading concern, keeping each widget's responsibility narrow.

3.6.3 Bottom Navigation

`HomePage` implements the three-tab bottom navigation bar using `IndexedStack`. The three tabs Metrics, Activities, and Profile are kept alive in the `IndexedStack` so that their scroll position, loaded data, and widget state are not lost when the user switches tabs. This is preferable to `PageView` or `Navigator.push`-based tab switching because it avoids re-running `initState` and re-fetching data every time a tab regains focus. The navigation bar itself is implemented as a Material 3 `NavigationBar` with `NavigationDestination` entries.

3.6.4 Deep Link Handling for OAuth

`OAuthCallbackScreen` receives its `Uri` parameter directly from `go_router`'s `state.uri`, which `go_router` populates automatically when the app is opened via a deep link matching the `/callback` route. The screen parses the `status`, `provider`, and `error` query

parameters, delegates refresh logic to `IntegrationViewModel.refresh()`, displays a confirmation or error `SnackBar`, and navigates to the home screen on completion. No manual URI interception outside of the router is required.

3.7 Environment and Firebase Strategy

3.7.1 Compile-Time Environment Injection

The application supports two runtime environments: `dev` and `prod`. The active environment is selected at compile time via a `--dart-define=APP_ENV=prod` (or `dev`) flag injected into the Flutter build command. The `AppEnv` class reads this value using `String.fromEnvironment`:

```
class AppEnv {
  static const _env = String.fromEnvironment('APP_ENV');

  static Env get currentEnv {
    switch (_env) {
      case 'prod': return Env.prod;
      default:     return Env.dev;
    }
  }
}
```

Because `String.fromEnvironment` is evaluated at compile time by the Dart compiler's constant folding, the environment value is inlined into the binary. It cannot be altered after compilation by modifying a configuration file, which satisfies the security requirement that environment separation must be enforced at build time rather than at runtime.

3.7.2 Firebase Initialization

At application startup, the app entrypoint in `main.dart` selects the correct Firebase project by switching on `AppEnv.currentEnv` and calling `Firebase.initializeApp` with the matching `DefaultFirebaseOptions`. Two generated Dart source files containing the Firebase configuration are excluded from the source repository via `.gitignore` and injected during the CI/CD pipeline build from encrypted GitHub Secrets, as described in Chapter 4. This ensures that Firebase project identifiers and API keys are never committed to version control and that a production binary cannot accidentally communicate with the development Firebase project. The `debugShowCheckedModeBanner` flag is also set conditionally on the active environment, providing a visible banner in development builds.

3.7.3 API Base URL Selection

`ApiConfig` applies the same `AppEnv` switch to select the environment-specific backend base URL, which is injected at provider construction time in `AppProviders`. All five HTTP services read from `ApiConfig.baseUrl`, so changing either environment's URL requires a single-line edit in one file.

3.7.4 Android Build Flavors

Build flavors are a Flutter mechanism that allow a single codebase to produce multiple distinct application variants from the same source, mapped to product flavors on Android and to schemes on iOS. Each flavor can carry its own application identifier, configuration files, and signing credentials, with the active variant selected at build time by a single `-flavor` flag passed to the build command. For this project, two flavors are defined: `dev` for development and `prod` for production.

Two alternative approaches were considered for environment separation. The first is multiple entry points, where separate files such as `main_dev.dart` and `main_prod.dart` each import a different configuration. This requires no build system changes and is simple to set up, but it does not handle platform-specific asset files that must physically reside in the correct directory for the build system to select them. The second is using `-dart-define` flags alone (the approach already used for the `APP_ENV` variable described in Section 3.7.1), which injects compile-time constants but has the same limitation. Build flavors were chosen because they are the only mechanism that natively supports per-variant source sets on both platforms, meaning each environment's configuration files such as `google-services.json` on Android and `GoogleService-Info.plist` on iOS are isolated by the build system itself rather than by a manual copy step, and the correct variant is selected by a single flag that the CI/CD pipeline sets automatically.

In `build.gradle.kts`, the `dev` flavor appends `.dev`, producing a distinct package name that allows both environment variants to be installed side-by-side on the same physical device. Each flavor has a dedicated source set directory (`android/app/src/dev/` and `android/app/src/prod/`) containing the flavor-specific `google-services.json`. Android signing is configured only for the `prod` flavor, with key credentials loaded at build time from a `key.properties` file generated dynamically by the CI/CD pipeline (Section 4.2.1). The `--flavor prod` argument to the Flutter build command selects the correct variant, and the same selection determines both the package name compiled into the binary and the directory path where Gradle writes the output artifact. The production `google-services.json` is written into the flavor source set by the pipeline before the build command runs (Section 4.2.2).

3.8 Authentication Design

3.8.1 Firebase Authentication

Firebase Authentication [8] is a managed identity service provided by Google that handles user registration, credential verification, token issuance, and session management. The FlutterFire SDK provides a first-party Flutter integration that exposes an asynchronous Dart API, and the backend can verify Firebase-issued JWTs against Firebase's public keys without managing any credentials of its own.

Three alternatives were considered. A custom JWT implementation on the backend would give full control over the identity layer but requires building credential storage, token signing, refresh logic, and session invalidation from scratch, adding significant backend complexity for no functional gain at this project scope. Auth0 is a widely used managed identity provider but is primarily designed for web and enterprise contexts, and its Flutter SDK is less mature than the native FlutterFire integration. Supabase Authentication is an open-source alternative with a self-hosting option, which makes it

a candidate to revisit if the application grows to a user volume where Firebase's paid tiers apply. Firebase Authentication was chosen because its free tier covers the expected usage volume, the FlutterFire SDK includes native Google Sign-In support, and token refresh is fully automatic with no application code required.

The application supports two sign-in methods: email/password and Google Sign-In. Email/password users must verify their email address before the application grants them access to the main screen. This check is enforced by the `_AuthDataGate` described in Section 3.6.2. Google-authenticated users are pre-verified by Firebase at the point of sign-in and bypass the email verification gate entirely.

3.8.2 Bearer Token Strategy

The Firebase JWT identity token is the sole credential presented to the backend. Each HTTP service calls `AuthService.getIdToken()` before every request and attaches the result as a `Bearer` token in the `Authorization` header. Firebase automatically refreshes the identity token before it expires (tokens have a one-hour validity period), so the token in each request is always current. The token is obtained from the in-memory Firebase Auth session and is never serialized to disk. Application code has no visibility into any wearable-provider OAuth tokens, which remain on the backend.

3.8.3 Google Sign-In Implementation

The `AuthService.signInWithGoogle()` method uses the Google Sign-In 7.x API [9]. It initializes `GoogleSignIn.instance` once with the `serverClientId` read from the environment-specific `DefaultFirebaseOptions.webClientId`, ensuring the correct Google project is used without any additional configuration. The method sets up a `StreamSubscription` on `GoogleSignIn.instance.authenticationEvents` before triggering the sign-in flow and uses a `Completer` to convert the stream-based result into an awaitable `Future`. The resulting `GoogleSignInAuthentication` credentials are used to construct a Firebase `OAuthCredential`, which is then passed to `FirebaseAuth.instance.signInWithCredential`.

3.8.4 Password-Sensitive Operations

Account deletion and email address changes require re-authentication before proceeding, as mandated by Firebase Authentication. For this purpose, `AuthRepository` exposes two methods: `reauthenticateWithPassword` and `reauthenticateWithGoogle`, which `ProfileViewModel` calls before performing these irreversible operations. The UI presents a confirmation dialog requesting the user's current password before initiating re-authentication. Any `AuthError` of type `requiresRecentLogin` is surfaced as a specific user-facing message prompting the user to sign out and sign back in before retrying.

3.9 Third-Party Provider Integration

3.9.1 IntegrationProvider Enum

Supported providers are modelled as an enum. Each case carries three fields: the identifier string used in API calls (`id`), the human-readable display name (`displayName`), and the path to the provider's logo asset (`logoAsset`). The four defined cases are `garmin`, `coros`, `polar`, and `suunto`. A `fromId` factory method converts a string received from the API into the corresponding enum value, returning `null` for unrecognized identifiers rather than throwing an exception. Adding support for an additional provider in a future release requires only adding a new enum case and the corresponding logo asset; no changes are required to the repository, service, ViewModel, or any other client component.

3.9.2 IntegrationStatus Model

`IntegrationStatus` is a domain model pairing each `IntegrationProvider` with a boolean `isConnected` field. The wire format returned by `GET /auth/status` is a single JSON object whose `status` field is a map of uppercase provider identifiers to boolean connection flags (for example, `{"GARMIN": true, "POLAR": false}`). The service layer iterates over this map, lowercases each key, resolves it to an `IntegrationProvider` via `fromId`, and constructs one `IntegrationStatus` per recognized provider, discarding unrecognized identifiers. The class implements structural equality by overriding `operator ==` and `hashCode`, so that `IntegrationViewModel` can compare freshly fetched statuses against the cached list and avoid calling `notifyListeners()` when no meaningful state has changed.

3.9.3 OAuth Flow Design

The OAuth authorization flow is designed so that the mobile client never handles any OAuth tokens. The complete flow proceeds as follows. The user taps the connect button for a provider on the Profile screen. `IntegrationViewModel.connect(provider)` calls `IntegrationRepository.getAuthUrl(provider)`, which forwards the call to `IntegrationService`. To obtain the authorization URL, the service issues an authenticated `GET /auth/{provider}/connect` request to the backend, which responds with an HTTP 302 redirect to the provider's authorization page. The service disables automatic redirect following and extracts the target URL from the `Location` response header. The ViewModel opens this URL with `url_launcher` in external application mode, directing the user to the provider's website in the system browser. The user approves access on the provider's website, after which the provider redirects to the backend's callback endpoint. The backend exchanges the authorization code for long-lived access tokens and stores them server-side. It then redirects the user's browser back to the application by targeting the registered custom URL deep link `endurance://callback?status=success&provider={provider}`. The Android operating system routes this deep link to the application, `go_router` matches it to the `/callback` route, and `OAuthCallbackScreen` handles it by calling `IntegrationViewModel.refresh()` and navigating home. OAuth tokens are never transmitted to or stored by the mobile client.

3.10 Backend API Contract

3.10.1 RESTful Interaction Pattern

The Flutter client communicates with the backend exclusively via REST over HTTPS. All requests carry a Firebase JWT bearer token in the `Authorization` header. The client uses `GET` for all read operations, `PUT /activities/{id}` for the single content mutation it performs (updating perceived effort, subjective feeling, category, and notes from the activity details edit form), `POST /me` for user record creation during onboarding, `PUT /me` for profile updates, and `DELETE /auth/{provider}/disconnect` for provider disconnection. The small set of mutation verbs keeps the client's interaction surface small and predictable.

3.10.2 Endpoint Groups

The backend exposes the following endpoint groups consumed by the client:

Prefix	Service	Description
`/activities`	`ActivityService`	Paginated list, detail by ID, PUT metadata
`/stats/health`	`MetricsService`	Paginated daily health metrics
`/stats/performance`	`MetricsService`	Paginated daily performance metrics
`/stats/race`	`MetricsService`	Paginated race time predictions
`/stats/mileage`	`MetricsService`	Current week mileage summary
`/me`	`UserService`	User profile create, read, and update
`/auth/{provider}`	`IntegrationService`	Connect (redirect), status, disconnect
`/categories`	`CategoryService`	Available activity category labels

Table 3.1: Backend API endpoint groups consumed by the Flutter client.

3.10.3 Pagination

Paginated endpoints accept `page` and `page_size` query parameters along with an optional `start` and `end` ISO 8601 datetime range for filtering by date. `ActivityViewModel` implements page-number pagination: page 1 is fetched first, `loadMoreActivities()` increments the page counter and appends results to the existing list, and the `_hasMoreData` flag is set to `false` when a page returns fewer items than the page size. The activity list's scroll controller triggers `loadMoreActivities()` when the user scrolls within 500 logical pixels of the list bottom, providing a continuous infinite-scroll experience with no explicit page navigation controls.

3.10.4 HTTP Client Lifecycle

Each HTTP service holds an `http.Client` field injected through the constructor, defaulting to `http.Client()` when no mock is supplied. The client is reused across all requests on the same service instance, allowing connection pooling. In `AppProviders`, `ActivityService`, `MetricsService`, `UserService`, and `CategoryService` each register a `dispose` callback that calls `service.dispose()`, which closes the underlying

client; `IntegrationService` holds an `http.Client` but does not register a provider-level dispose callback. The constructor-injectable `http.Client` serves a secondary purpose: it enables service-level unit tests to inject a mock client, allowing HTTP-level test verification without a live network.

3.11 User Interface and Theming

3.11.1 Material 3 Theming

Material Design 3 (M3) is Google's design system for building user interfaces across platforms [10]. It defines a visual language covering color, typography, shape, and component behavior, and Flutter provides first-class support for it through its `ThemeData` API, enabling an application to adopt the full system by setting `useMaterial3: true`.

Three approaches were considered. The Cupertino design system, Apple's native iOS style, renders correctly on iOS but produces inconsistent results on Android, where the visual conventions differ significantly. Using a third-party cross-platform design library was considered but ruled out on two grounds: several such libraries do not cover the full set of standard Flutter widgets, requiring fallbacks that undermine visual consistency, and relying on a community-maintained library introduces the risk of abandonment should the project lose its maintainers. Material 3 was chosen because it is backed by Google and natively supported by Flutter, renders consistently on both Android and iOS, provides a complete component set with no gaps, and follows a simple and modern visual language suited to a fitness application. A practical benefit was that M3 is available as a Figma component library, which allowed the design phase to proceed directly in the prototyping tool without requiring manual asset creation.

The color scheme is derived from a single seed color (`Colors.lightGreenAccent`) via `ColorScheme.fromSeed`, which Material 3 uses to generate a full harmonious palette across all tonal roles (primary, secondary, tertiary, error, surface, and background). Separate `ThemeData` instances are built for light and dark brightness from the same seed color, ensuring visual coherence is maintained across both modes. The theme is defined in `AppTheme` and applied via `MaterialApp.router`'s `theme` and `darkTheme` properties with `ThemeMode.system`, so the operating system dark-mode preference is respected automatically without any application-level user toggle.

3.11.2 Widget Library

Reusable widgets are collected in a dedicated widgets package. Each data-displaying widget has a corresponding shimmer counterpart that mirrors its layout with animated placeholders, providing a skeleton loading experience without blocking the interface while data is being fetched. The organization of individual widget groups and their visual structure is covered in Chapter 6.

3.11.3 Activity Charts

`fl_chart` [11] is an open-source Flutter charting library that provides line, bar, pie, and scatter chart types with support for interactive touch gestures and customizable visual styling.

Two alternatives were considered. The `community_charts_flutter` package, a community-maintained fork of Google's original charting library, was ruled out due to low maintenance activity, which introduces a risk of the library becoming incompatible with future Flutter versions. Syncfusion Flutter Charts provides a comprehensive charting suite with full multi-axis support but is distributed under a proprietary license that requires either a community registration or a commercial plan, creating a dependency on a single vendor's licensing terms. `fl_chart` was chosen because it is open source under a permissive license, actively maintained, supports multiple simultaneous Y-axes natively, and exposes programmatic touch callbacks.

Concretely, the chart exposes an `onTooltipPositionChanged` callback that fires with the integer data index of the selected point, which is the architectural entry point for the chart-to-map synchronization mechanism described in Section 3.11.5. The visual configuration of each chart variant, series types, axis labels, and overlay rendering, is described in Chapter 6.

3.11.4 Route Map

GPS routes are rendered using `flutter_map` (version 8.2.2) [12] with the `latlong2` package for coordinate representation. `flutter_map` was chosen over Google Maps and Mapbox for several reasons. It renders OpenStreetMap (OSM) tiles, which are free regardless of usage volume and require no API key. Google Maps and Mapbox both require a registered API key that must be managed as a secret in the build pipeline, adding configuration overhead that `flutter_map` avoids entirely. `flutter_map` also exposes a composable layer system that allows arbitrary overlays to be rendered above the base tiles without restriction, which is required for the custom route polyline, the animated position marker synchronized with the activity chart, and potential future visualizations such as heatmap layers.

`ActivityRouteMap` exposes a `markerTimestamp` parameter: when a non-null timestamp is provided, the widget performs a linear scan over the route point list to find the closest GPS point by time and repositions a marker accordingly. This interface is what the chart-to-map synchronization mechanism (Section 3.11.5) targets. The visual appearance of the map and route overlay is described in Chapter 6.

3.11.5 Chart-to-Map Synchronization

The activity detail screen synchronizes the chart touch position with the map marker by maintaining a `_markerTimestamp` field as the shared state connecting the two widgets. When the user touches the `ActivityDetailsChart`, the chart fires a callback with an integer data index. This callback is exposed via `onTooltipPositionChanged`, which the screen uses to map the touch position back to a timestamp.. The screen translates this index into a `DateTime` by reading `activityDetails.measurements[dataIndex].time` and calls `setState` to update `_markerTimestamp`. The `ActivityRouteMap` widget rebuilds with the new timestamp and performs a linear scan over the route point list to find the closest match by time. A linear scan is sufficient because GPS route arrays are bounded to the number of seconds in the activity (typically 1 000 to 7 000 points), making the O(n) cost negligible. Matching by timestamp rather than by array index avoids alignment issues between the `measurements` and `route` arrays, which may differ in length because GPS and sensor data are sampled at different rates. Chart touches

update the map marker, but map touches do not update the chart.

3.12 Core Utilities

A set of static utility classes centralizes cross-cutting concerns that would otherwise be duplicated across ViewModels and widgets. `ActivityFormatter` gathers all display formatting for duration, pace, distance, and date into one place, using truncation rather than rounding for distance values to avoid overstating the result. `SportUtils` exposes `isGpsSport` and `iconForSport`: the former governs whether the activity detail screen mounts a route map widget; the latter maps sport type strings to `IconData` values for use in list items, with unrecognized types falling back to a generic icon. The Borg 6–20 Rating of Perceived Exertion (RPE) scale and a 1–5 subjective feeling scale are encoded in dedicated utility classes used to initialize the activity edit form and render slider labels. `globalFetch` is a free function that triggers a coordinated refresh of both `ActivityViewModel` and `MetricsViewModel`, ensuring that activities and metrics are always reloaded together when a full application refresh is required.

Input validation for authentication and onboarding forms is centralized in the dedicated `FormValidators`, following Flutter's standard `TextFormField` validator contract: each method returns `null` on success and a human-readable string on failure. Validation covers email format, password strength, required fields, date-of-birth format, and display-name length, and is performed client-side before any network call.

Activity categories are fetched once from `/categories`. The result is cached in `CategoryProvider`, a shared `ChangeNotifier` sitting alongside the ViewModels in the `MultiProvider` tree. Its load method is a no-op when the cache is already populated, preventing redundant API calls when multiple ViewModels request the category list in the same session.

Chapter 4

Actions and Pipeline

The Endurance App involves a mobile client communicating with a RESTful backend and third-party OAuth providers across two Firebase environments. Producing a working release requires Android signing keys, Firebase configuration for both development and production, correct build flavor selection, and compile-time environment variable injection. Performing these steps manually for every release is error-prone and difficult to reproduce consistently. A CI/CD pipeline addresses this by automating the full path from code change to deployable artifact, ensuring that every release goes through the same sequence of steps regardless of who triggers it. The pipeline also acts as a quality gate: code that does not pass tests cannot be built, and code that does not build cannot be deployed. For a project intended to grow and accept future contributions, this reduces the onboarding cost for new developers who do not need to understand the full build and signing process to produce a working artifact. The current pipeline covers Android only, with iOS and web as future scope. The following sections describe how the pipeline is structured, how secrets are managed, how builds and deployments are executed, and what limitations remain.

4.1 Overview

Continuous integration (CI) is the practice of automatically building and testing every code change as soon as it is pushed, catching regressions before they accumulate. Continuous deployment (CD) extends this by automatically delivering a verified build to one or more distribution targets once all quality gates have passed. Together, CI/CD replaces a fragile manual release process with a reproducible, auditable pipeline that runs identically on every trigger regardless of how it is initiated.

The continuous integration and continuous deployment pipeline is implemented using GitHub Actions [13]. It runs on a self-hosted Linux x64 machine. Because Flutter builds are resource-intensive and benefit from persistent local caches, self-hosted runners avoid the per-minute billing of GitHub-hosted machines. Moreover, the build environment can be configured once and reused across runs rather than being provisioned from scratch each time. The workflow is defined in a single file located at `build-upload-android.yml` and is triggered automatically on every push to the `dev` branch and on every pull request regardless of the source branch. Developers can also manually run the pipeline when needed without pushing a new commit through the `workflow_dispatch` event.

The workflow specifies explicit permissions at the highest level. The `contents:`

`write` permission allows the creation of Git tags and GitHub Releases. Declaring permissions explicitly rather than relying on the default token scope is a security practice that limits what the workflow can do if compromised.

The pipeline is organized around four jobs that execute in a strict linear sequence enforced through the `needs` keyword. Those jobs are `test`, `build_apk_prod`, `gh_release` and `play_deploy_internal`. Each job depends on the successful completion of the previous one, meaning a test failure prevents any build from being produced, a build failure prevents any release from being created, and a release failure prevents any deployment to Google Play. This strict dependency chain ensures that no artifact is published or deployed unless every preceding stage has passed.

An important distinction in the pipeline is that not all jobs run on all triggers. The `test` job executes on every trigger without exception, ensuring that every pull request and every push is validated against the test suite. However, the three subsequent jobs `build`, `release`, and `deployment` are additionally gated by the condition `if: github.event_name == 'push' && github.ref == 'refs/heads/dev'`, which restricts them to pushes on the `dev` branch only. This means the pipeline operates in two distinct modes depending on the context. For pull requests, it runs in a test-only mode where the code is checked out, dependencies are resolved, and tests are executed, but no artifact is built and no deployment occurs. For pushes to the development branch, the full pipeline runs from testing through to Google Play upload. This separation serves a clear purpose: pull requests are validated for correctness without producing unnecessary artifacts or consuming deployment resources, while merges to the development branch trigger the complete delivery chain. The result is a pipeline that balances feedback speed for contributors with deployment rigor for the main integration branch. Figure 4.1 illustrates the full structure.

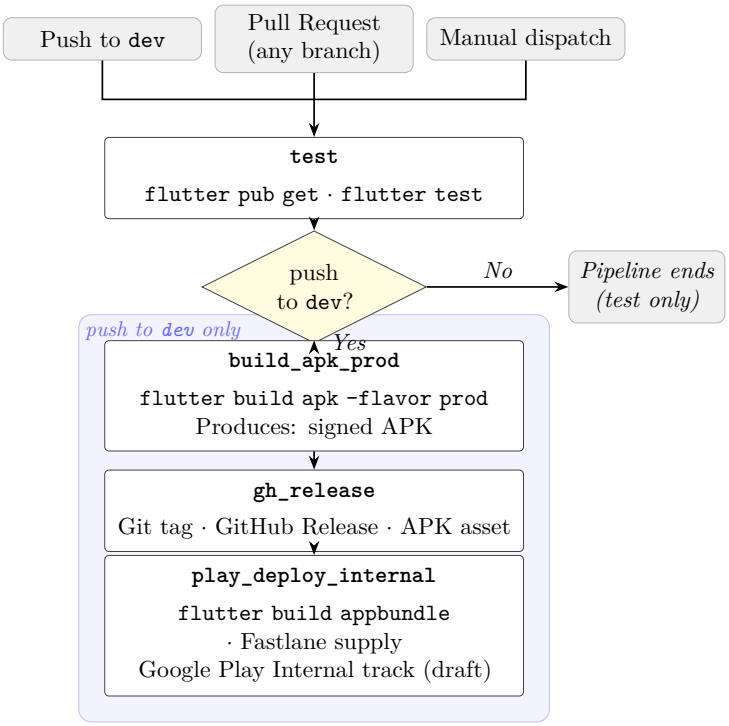


Figure 4.1: CI/CD pipeline: `test` runs on every trigger; the build, release, and deployment jobs run only on pushes to `dev`.

4.2 Secret Management

A fundamental security requirement for any CI/CD pipeline handling a mobile application is to never store credentials, keys, or environment-specific configuration files within the application source code. It is not possible to erase such material from the Git history without rewriting commits and force-pushing if it is ever committed. Thanks to the pipeline's use of GitHub Actions encrypted secrets, sensitive information is stored securely by GitHub, masked in all log output, and never exposed to the running workflow as plain text. Instead, secrets are made available as environment variables or step inputs at the moment of execution, ensuring that they are only accessible within the context of the job that needs them and are not persisted in any form on disk or in logs.

The pipeline relies on nine secrets grouped into four categories: Android signing credentials, Firebase configuration files, the Google Play service account key for store submission, and the automatically provisioned `GITHUB_TOKEN` for creating GitHub Releases. The following subsections describe how each category is injected at runtime.

A technical constraint appeared from the fact that GitHub Secrets are string values, while the Android keystore and the JSON Google services configuration are file credentials. Encoding these files in base64 transforms them into ASCII strings that can be safely stored as secrets. At runtime, the `timheuer/base64-to-file@v1` [14] action is used to decode these secrets back into actual files and write them to the runner's temporary directory, returning the file path as a step output that subsequent steps can use. This approach is used for the keystore in both the `build_apk_prod` and `play_deploy_internal` jobs, and for the Google Play service account key in the Fastlane deployment job `play_deploy_internal` also.

4.2.1 Android Signing Credentials

The Android signing configuration presents a specific challenge because the Flutter build system expects a `key.properties` file inside the `android/` directory containing the keystore path and credentials in a key-value format. Since this file must reference the path of the keystore as it exists on the runner's filesystem, a path that is only known at runtime after the `timheuer/base64-to-file@v1` action has decoded the keystore, it cannot be prepared in advance. The pipeline generates this file dynamically at build time using a shell heredoc, writing the four required values directly from the GitHub Secrets into the file. The file never exists in the repository and is only materialized during the job execution on the runner.

4.2.2 Firebase Configuration

Firebase configuration follows a different pattern. There are two generated Dart source files produced by the FlutterFire CLI (or obtained from the Firebase console). They contain Firebase project identifiers, API keys, and platform-specific configuration. Because these files are referenced by the application's Dart code at compile time, they must be present on disk before the build command runs. Both files are decoded from their respective base64 secrets and written into the application's Dart source tree at the start of each build job. Injecting both development and production configurations in every build ensures the codebase compiles correctly regardless of which flavor is being built, since the Dart code references both files. The production `google-services.json`

for Android is similarly decoded and written into the correct Android flavor source set directory, with `mkdir -p` used to create the directory if it does not already exist on the runner.

4.2.3 Deployment Tokens

For the Google Play deployment, the service account key is decoded to a temporary file in the runner environment and its path is passed to Fastlane through the `ANDROID_JSON_KEY_FILE` environment variable. Fastlane then uses this file to authenticate against the Google Play Developer API to query the current version code and upload the signed Android App Bundle. The key file is never written to a location inside the repository working directory and is not persisted between jobs.

The `GITHUB_TOKEN` secret differs from the other eight in that it is not manually created or stored by the developer. GitHub Actions automatically generates a short-lived token at the start of each workflow run, scoped to the repository in which the workflow executes. In this pipeline, the token is consumed in the `gh_release` job by `softprops/action-gh-release@v2` [15] to create a Git tag, publish a GitHub Release entry, and upload the signed Android Package (APK) as a release asset. As described earlier, the workflow's top-level `permissions` block restricts this token to the minimum scope required for this operation.

All files produced from secrets are not committed, uploaded as artifacts, or transferred between jobs. They exist only to serve the build steps that require them and are overwritten on each subsequent pipeline execution.

4.3 Build Pipeline

4.3.1 Test Job

The first job, `test`, executes unconditionally on every trigger: pushes to `dev`, pull requests from any branch, and manual dispatch. This is the only job with no branch restriction. Its sole purpose is to validate that the code compiles and that the test suite passes, providing the earliest possible signal to contributors without waiting for build infrastructure to be engaged.

The Flutter toolchain is installed via `subosito/flutter-action@v2` [16] with the version pinned explicitly to `3.35.5` on the `stable` channel. Pinning is a deliberate reproducibility measure: without it, the action installs whatever the latest stable release is at execution time, introducing the possibility of silent behavioral differences between runs triggered days or weeks apart. With a pinned version, every run of the test job uses an identical Dart compiler and standard library. The job then executes `flutter pub get` to resolve dependencies and `flutter test` to run the full test suite. The job succeeds only when all tests exit with code zero. The current suite contains only a placeholder assertion to keep the pipeline structure in place. Both unit and widget testing are absent due to time constraints and are identified as priorities for future work, discussed further in Section 4.5.1.

4.3.2 Build Job

The second job, `build_apk_prod`, is restricted to pushes on `dev` (Section 4.1): pull requests are validated but never produce an artifact, preventing unofficial or unsigned builds from entering any distribution channel.

The Flutter version is pinned again to `3.35.5` in this job. Because each job runs in an independent execution context, the toolchain must be reinstalled regardless of what the preceding job installed. Re-declaring the same pinned version ensures that the binary producing the artifact was compiled by an identical toolchain to the one that executed the tests.

With the runtime environment and secret-derived files in place the build is invoked as follows:

```
flutter build apk --release --flavor prod --dart-define=APP_ENV=prod
```

The `--flavor prod` argument selects the `prod` Android build variant; the flavor dimension, source set directories, and signing configuration are defined in the application design and described in Section 3.7.4.

The `--dart-define=APP_ENV=prod` argument injects the compile-time environment value consumed by `AppEnv.currentEnv`; the mechanism, constant folding guarantee, and security rationale are described in Section 3.7.1.

The resulting APK is uploaded via `actions/upload-artifact@v4` under the name `apk-prod`, at path `build/app/outputs/flutter-apk/*prod*-release.apk`, with the `if-no-files-found: error` option, which instructs the action to fail the job explicitly if no file matches the pattern, rather than silently succeeding with an empty artifact. This ensures that a misconfigured build that produces no output is surfaced immediately as a pipeline failure.

4.3.3 Artifact Formats

The pipeline ultimately produces two distinct signed artifacts: an APK and an Android App Bundle (AAB). The APK is built directly by the `build_apk_prod` job and is distributed via GitHub Releases as a directly installable artifact that can be sideloaded onto any Android device without Play Store involvement. The AAB is built later by Fastlane in the `play_deploy_internal` job using an equivalent command:

```
flutter build appbundle --release --build-number {versionCode}
--flavor prod --dart-define=APP_ENV=prod
```

Unlike an APK, an AAB is not directly installable. It is an optimized delivery format that Google Play uses to generate device-specific APK splits at distribution time, reducing download sizes for end users by including only the binaries, resources, and native libraries relevant to each device configuration [17]. Google Play requires the AAB format for new application submissions, making it the mandatory submission artifact for the Play Store channel. The two formats therefore serve entirely different distribution channels while encoding the same application binary, built from the same commit, with the same flavor and compile-time environment.

4.4 Deployment

Fastlane [18] is an open-source automation tool for mobile application delivery that provides a lane-based workflow for building, testing, and deploying Android and iOS applications.

One alternative was considered: a dedicated GitHub Action for Play Store uploads, which handles authentication and AAB upload within the workflow file. However, it provides no native version code management, requiring additional steps to query the Play API and increment the version separately. Fastlane was chosen because it handles authentication, version code querying, and upload in a single declarative lane through its `supply` plugin and the `fastlane-plugin-increment_version_code` [19] plugin.

Before each submission, the lane queries the Google Play Developer API [20] to retrieve the current maximum version code on the internal track and increments it by one, deriving the version number from the Play Console rather than a counter maintained in the repository. The AAB is then submitted with `release_status: "draft"`, making it visible in the Play Console but not distributed until a human explicitly promotes it. This introduces a mandatory human checkpoint between a successful CI run and any user-visible distribution. The internal track was chosen over alpha, beta, or production because it is the lowest-visibility option, appropriate for a project where not every passing CI run represents a build ready for external review.

4.5 Limitations and Future Work

4.5.1 Test Coverage

Although the architecture was intentionally designed to facilitate testing, current test coverage remains insufficient. The ViewModels utilize constructor injection for their dependencies, allowing them to be thoroughly validated through pure Dart unit tests using mocked providers. The user interface was validated by testers throughout the beta period, providing direct functional feedback on real devices. Unit tests for ViewModels and widget tests for stable screens are identified as a concrete priority for the next development phase.

4.5.2 Platform Coverage

The pipeline currently targets Android only. Extending it to iOS would require a macOS runner and Apple Developer credentials managed as pipeline secrets, with deployment handled through Fastlane's existing `pilot` lane or a dedicated App Store Connect action. This is a natural extension of the current pipeline rather than a fundamental rework.

Chapter 5

Beta Execution

Delivering a working application to real users requires more than a passing CI run and a signed artifact in the Play Console. It requires testers who generate meaningful data, an onboarding path that gets them connected without friction, a backend that does not break builds they have already installed, and a feedback loop tight enough that issues surface and are resolved before they erode confidence in the platform. This chapter covers the Closed Beta used to distribute the application, the tester profile and recruitment approach, the onboarding experience, the strategy for maintaining backward compatibility across concurrently installed versions, how feedback was collected and acted upon, the concrete issues that surfaced during testing, and the runtime monitoring strategy and its planned extensions.

5.1 Google Play Closed Beta

The application was distributed via the Google Play Store under the Closed Beta program. Closed Beta is a store designation that restricts distribution to a set of invited testers, keeping the application invisible to the general public while still allowing installation directly from the Play Store. The listing was not publicly discoverable: distribution was limited to users who held a direct link to the store page for the application package. This restricted visibility was intentional, keeping the beta population small and known while still avoiding the friction of manual APK sideloading.

The promotion path from a qualifying commit to a tester's device followed the pipeline described in Chapter 4. An automated build deployed a draft release to the Internal Testing track (Section 4.4). The developer then manually promoted this draft to the Closed Beta listing through the Play Console. This manual promotion step was a deliberate checkpoint ensuring that only builds judged ready for external visibility were made available, regardless of how many automated builds had passed on the internal track.

Testers installed the application directly from the Play Store link. The Play Store's automatic update mechanism then delivered subsequent builds to installed devices passively, without requiring any action from the tester. Because no forced update was enforced, different testers could be running slightly different build versions simultaneously during any given testing window.

5.2 Tester Profile

The beta cohort consisted of a small group of runners who came to the application through the Closed Beta link rather than through a structured recruitment campaign. The group was not homogeneous in training volume: some testers were high-frequency runners completing six to eight sessions per week, while others trained less regularly. This variation turned out to be directly relevant to one of the issues described in Section 5.7.

At the time of the beta, only running was actively represented in the tester cohort; cycling and triathlon integrations were planned as future scope. All testers used Garmin devices, making Garmin the only wearable provider exercised against real hardware during the testing period. The Polar backend integration was implemented and ready, but no tester owned a Polar device. Coros and Suunto integrations were identified as future work. As a consequence, only the Garmin provider integration was validated end-to-end during this phase.

5.3 Onboarding

The user journey from installation to a connected device followed a fixed sequence: install the application from the Closed Beta link, create an account via Firebase Authentication using either email/password or Google Sign-In (Section 3.8), complete the onboarding flow to create a user profile, and then connect a wearable provider. The `_DataGate` widget (Section 3.6.2) handled the routing distinction between a first-time user who must complete onboarding and a returning user who proceeds directly to the home screen.

On first connection to a provider, the backend triggered an initial data backfill, ingesting the user's existing historical activity records from the provider's API. This first-sync operation surfaces in the client as a loading and shimmer state (Section 3.5.4), acknowledging to the tester that data is being fetched without blocking the interface.

5.4 Testing Period

The beta ran for multiple weeks of continuous operation. There were no structured test sessions: testers were free to use the application however they chose, integrating it naturally into their ongoing training routine. No tasks were assigned, no scripts were followed, and no reporting was expected on a schedule.

New builds were delivered continuously throughout the testing period and promoted to the Closed Beta listing as described in Section 5.1.

On selected screens, a Figma-based design review loop preceded implementation. Mockups were reviewed with the PhD student collaborator, revisions were made within the design tool, and only then was the screen implemented in code. This front-loaded the design feedback cycle for the screens involved. The UI design process is described in detail in Chapter 6.

5.5 Backward Compatibility During the Beta

A fundamental constraint of running a live beta without forced updates is that any installed app version must continue to function correctly against the current backend, even as the backend evolves. A schema change that broke older clients would require every tester to update immediately which is a friction that is inconsistent with the passive update model provided by the Play Store.

Two design decisions described in Chapter 3 directly addressed this constraint. First, the sport category list is fetched from `GET /categories` at runtime rather than embedded in the binary, so the backend can add or rename categories without requiring a client update. Second, unrecognized provider identifiers returned by the API are silently ignored rather than crashing the client, meaning a new provider can be introduced on the backend without breaking installed versions.

All backend changes made during the testing period were additive: new response fields were introduced where needed, existing fields were never removed, and no endpoint paths changed. In combination with the two runtime-fetch strategies above, this policy ensured that no tester ever experienced a broken build as a result of a backend schema change.

The current implementation does not enforce a minimum client version. If a future backend release requires a breaking change, there is no mechanism to prompt users to update before the change takes effect. Android provides a native solution through the Play Core In-App Updates API [21], exposed on Flutter via the `in_app_update` package [22], which supports an immediate flow that blocks the application until the update is installed. This is identified as a continuation task.

5.6 Feedback Collection and Response Cycle

Feedback arrived through two channels: direct messaging for immediate and informal reports, and the in-app **Feedback** button in the Profile screen for structured submissions without interrupting a training session.

Bug reports typically arrived as a screenshot paired with a written description. Usability observations were surfaced through conversation rather than formal documentation. Feature requests were logged and triaged as either in-scope for the current beta or deferred to future development.

The response cycle was short. A tester would report an issue; the developer would reproduce and fix it; the fix would be committed to `dev`, triggering the CI/CD pipeline; and the tester would receive the corrected build via the Play Store auto-update mechanism, requiring no manual action on their part.

5.7 Issues Found During Testing

Two concrete issues were identified and resolved during the testing period.

5.7.1 Activity Card Overflow on Large System Font Settings

A tester using an Android device with a non-default display size setting reported that the metrics row on activity cards was clipping content. The row displays three values:

distance, duration, and pace, separated by bullet points. The original implementation used a fixed horizontal layout with no overflow handling. On standard display sizes this rendered correctly, but at larger display scales the combined width of the three values exceeded the available card width, causing the rightmost content to be cut off.

The fix replaced the fixed layout with a reflowing layout: when the available horizontal space is insufficient, the values wrap onto a second line rather than overflowing. At default display settings the three values continue to appear on a single line, so the change introduced no visual regression for the majority of users.

5.7.2 Performance Metrics Window Too Narrow for Lower-Frequency Testers

The performance metrics charts display VO_2max, threshold heart rate, and threshold pace as line charts over a trailing window of days. The backend was already returning thirty days of data, but the chart was only rendering the last seven. For testers running six to eight times per week, seven days still produced a reasonably dense line. For testers who trained less frequently, the window often contained only one or two data points, rendering the trend line nearly meaningless.

The fix exposed the full thirty-day series to the chart. The dot and line rendering was adjusted in parallel: with thirty data points on a small chart, showing a dot at every point was visually noisy, so dots were reduced to a single marker at the last known value. The line style was also changed from straight segments to a smooth curve to improve readability when data is dense.

A static thirty-day window nonetheless remains a compromise. A user who trains twice a week and a user who trains every day have fundamentally different data densities over the same calendar period. A more appropriate solution would adapt the window to the user's own training cadence, for example, targeting a fixed number of recent sessions rather than a fixed number of days. This is identified as a future improvement.

5.8 Runtime Monitoring

The beta cohort's small size and direct communication channels made backend server logs a sufficient monitoring strategy. When a tester reported unexpected behavior, the developer inspected the relevant server-side log entries to establish whether the origin was in the client or the backend. For a cohort generating low and predictable traffic where every tester could be reached directly, this lightweight approach matched the scale of the deployment. As the user base grows, two Firebase services represent natural additions that would extend this visibility: Firebase Crashlytics [23] would capture client-side crashes automatically without requiring a tester to report them, and Firebase Performance Monitoring [24] would provide continuous insight into network request latency and screen rendering times across the full installed base.

Chapter 6

Main UI

The primary audience of this application is athletes. The interface must show training history, health metrics, and performance indicators without requiring any technical knowledge of how the data was collected or normalized. Because the application also serves as a data-collection instrument for a PhD research project, it is essential that participants actually use it consistently, which means the user experience must be good enough that athletes choose to keep it in their routine. This is precisely why the PhD-student collaborator was involved in the screen design. He had a direct interest in maximizing adoption and ensuring that every screen encouraged participants to log their sessions rather than abandon the app.

The design strategy was therefore to model every screen in Figma and validate it with the PhD-student collaborator before any Flutter code was written. This enforced a separation between the cost of discovering misaligned expectations (which is low at the design stage) and the cost of rewriting implemented screens, which is high.

6.1 Design Tool: Figma

6.1.1 What Figma Is

Figma is a browser-based, collaborative interface-design tool [25]. Unlike traditional desktop tools such as Sketch or Adobe XD, Figma stores every file in the cloud and renders the canvas entirely in the browser, allowing multiple collaborators to edit the same file simultaneously, view one another's cursors in real time, and leave comments anchored to specific design objects. There is no export-and-send step: sharing a Figma link grants the recipient instant, live access to the file in its current state.

Screens are represented as **Frames**, which are fixed-size containers corresponding to a `Scaffold` in Flutter. Reusable elements are defined as **Components**, which can be instantiated multiple times across the file. Editing the master propagates the change to all instances, mirroring the way a Flutter widget class propagates changes to all its usages. Layout within Frames is governed by **Auto Layout**, a stacking mechanism with configurable direction, gap, and padding that maps almost directly to `Row` and `Column` in Flutter. Shared visual constants (colors, typography, and spacing values) are captured as **Design Tokens**, the Figma equivalent of `ThemeData` entries. Navigation between Frames is simulated in the **Prototype** view by wiring hotspot links, producing a clickable flow that can be opened on a physical mobile device. The **Inspect Panel** exposes exact measurements, color values, and font details to the developer

implementing the design.

6.1.2 Why Figma Was Chosen

Three properties made Figma the natural choice for this project. First, the browser-based sharing model meant the PhD-student collaborator could review mockups with no software installation, leaving comments directly on the design canvas that were resolved at the bi-weekly sprint meetings. Second, the Prototype view allowed usability problems such as navigation dead-ends, ambiguous button placements, and confusing information hierarchy to be discovered on a physical mobile device before any implementation cost had been incurred. Third, the close structural correspondence between Figma concepts and Flutter widgets reduced translation ambiguity: Inspect Panel values could be copied directly into `Padding` and `SizedBox` calls, and Auto Layout rules mapped almost directly to `Column` and `Row` parameters, keeping the gap between the approved mockup and the shipped screen small.

6.1.3 Figma Workflow Applied in This Project

Figma was used primarily at the start of the project to establish the overall visual direction and validate the main screens before any Flutter code was written. The core screens (authentication, onboarding, home, metrics, and activities) were drafted as Frames using the official Material 3 Design Kit published by Google on the Figma Community [26], wired into a clickable Prototype, and reviewed with the PhD-student collaborator. His feedback was incorporated before implementation began. For screens added in later sprints, the Figma mockups were used as a personal design reference rather than a formal review artifact, with the Inspect Panel consulted for spacing and color constants during implementation.

6.1.4 Figma and the Material 3 Design System

The `AppTheme` module configures Flutter's Material 3 design system with a single seed color:

```
const mainSeedColor = Colors.lightGreenAccent;

ThemeData buildAppTheme(Brightness brightness) {
  return ThemeData(
    useMaterial3: true,
    colorScheme: ColorScheme.fromSeed(
      seedColor: mainSeedColor,
      brightness: brightness,
    ),
  );
}
```

`ColorScheme.fromSeed` generates the full Material 3 tonal palette from a single seed, producing a consistent set of named color roles (`primary`, `surfaceContainerLow`, `primaryContainer`, `onSurfaceVariant`, and so on) for both light and dark brightness [10]. The Figma mockups used the same Material 3 Design Kit seeded with

`Colors.lightGreenAccent`, so the color roles in the Figma Frames correspond exactly to the roles used in the Flutter implementation. This alignment ensured that the mockup was an accurate preview of the shipped application, not an idealized approximation that would diverge during implementation.

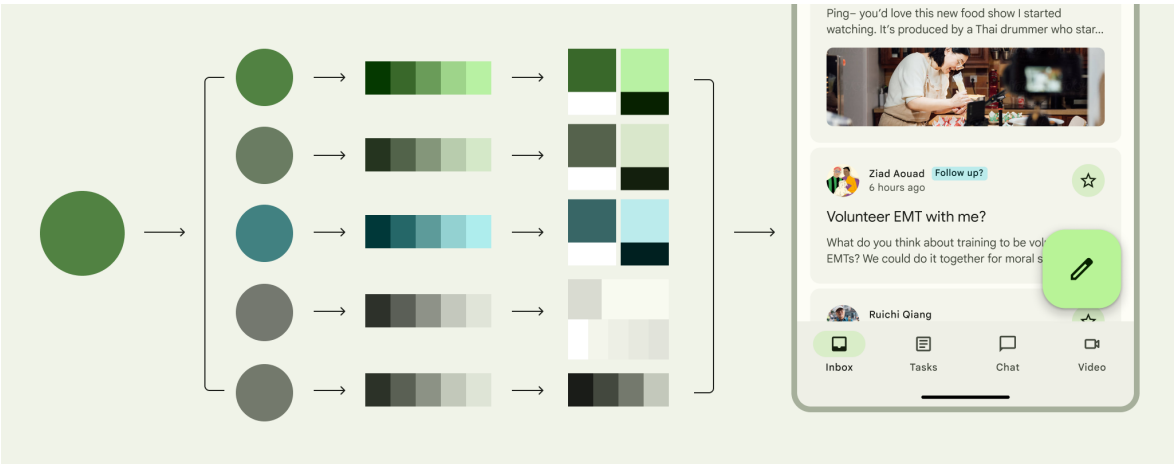


Figure 6.1: Material 3 color system: a single source color is expanded into five tonal palettes, which are then mapped to named color roles that drive every surface and component in the UI. Source: [27].

6.2 Application Theme and Visual Identity

The theme governs every visual property that is not overridden at the widget level. Four decisions are worth stating explicitly because they recur throughout the screen descriptions below.

6.2.1 Color System

Both light and dark color schemes are generated from the same seed, and the active scheme is selected by `ThemeMode.system` in `main.dart`, so the application respects the device's system-wide dark-mode setting without any in-app toggle. Because `ColorScheme.fromSeed` derives tonal contrast ratios algorithmically, all text-on-surface color pairs in both schemes meet the Material 3 accessibility contrast requirements by construction.

6.2.2 Typography

No custom typeface was introduced. The application uses the default Material 3 type scale (`textTheme`), which renders using the system font on Android. This keeps the APK size minimal and ensures that text renders legibly across all Android OEM font stacks.

6.2.3 Elevation and Shape

Cards throughout the application use `elevation: 0` with a `surfaceContainerLow` background color and a 16 px corner radius (`BorderRadius.circular(16)`). This is

the Material 3 tonal-surface pattern: visual depth is expressed through color tint rather than drop shadow, which retains legibility in both light and dark modes and avoids the halo artifacts that cast shadows produce on dark backgrounds.

6.2.4 Navigation Bar

The persistent shell uses Flutter's Material 3 `NavigationBar` widget with three tab entries of type `NavigationDestination`. These represent Metrics (`Icons.bar_chart`), Activities (`Icons.directions_run`), and Profile (`Icons.person`).

6.3 Application-Wide Navigation

The routing architecture, including the choice of `go_router`, the flat route configuration, the `_AuthDataGate` and `_DataGate` widgets, and the `IndexedStack` tab shell, is covered in Section 3.6. The remainder of this chapter describes each screen in the order a new user encounters them: authentication, onboarding, the home page and bottom navigation bar, then the three main tabs and their detail screens.

6.4 Authentication Screens

The authentication flow comprises sign-in, sign-up, email verification, and password reset screens. Sign-in and sign-up share the same field components and offer both email/password and Google Sign-In paths. On a successful authentication, the application calls `TextInput.finishAutofillContext()` to signal the platform credential manager to save the entered credentials, enabling system autofill on subsequent launches. The email verification screen is purely reactive: the polling logic (`user.reload()` on a timer) lives entirely in `AuthViewModel`, so the screen rebuilds when the ViewModel emits a verified state without containing any stateful behavior of its own. The password reset screen delegates to Firebase's built-in reset email flow [8] and confirms dispatch with a SnackBar.

6.5 Onboarding Screen

`OnboardingScreen` is shown exactly once, immediately after the first sign-in, when `_DataGate` calls `UserViewModel.loadUser()` and discovers that no server-side user record exists for the authenticated Firebase user.

The screen collects three fields. The first is a free-text display name, validated as non-empty. The second is a date of birth entered through a date-picker dialog, validated as a date in the past. The third is a boolean toggle labeled "Allow retrieval of past data", which defaults to `true`. When enabled, it grants the backend permission to retroactively fetch historical wearable data accumulated before the user connected a provider, populating the researcher's dataset with the user's full prior training history rather than only data recorded after the connection was established. This toggle is a direct implementation of the research data infrastructure requirements described in Section 2.3.7.

The email address is read from `FirebaseAuth.instance.currentUser.email` and is not presented in the form, since it is already known from the authentication step.

On submission, `UserViewModel.createUser()` is called with the collected fields. On success, `go_router` navigates to `/`, which re-evaluates the gate and routes to `HomePage`.

6.6 Metrics Screen

`MetricsScreen` displays four groups of metric cards: performance indicators (VO_2max, lactate threshold heart rate, threshold pace, and weekly mileage progress toward a goal), race time predictions (5 km, 10 km, half-marathon, marathon evolving over time), daily heart rate (maximum, resting, minimum), and daily health data (steps, active and total calories, distance). Performance and race series are rendered as `MetricEvolutionChart` sparklines or a `MultiSeriesMetricEvolutionChart`, both built on `fl_chart` [11]. Weekly mileage progress uses a circular arc chart. All values are derived from raw normalized data by the backend, not from manufacturer-provided computed fields, in accordance with the performance metrics requirements (Section 2.3.5). Pull-to-refresh calls `MetricsViewModel.refresh()` without clearing the currently displayed data.

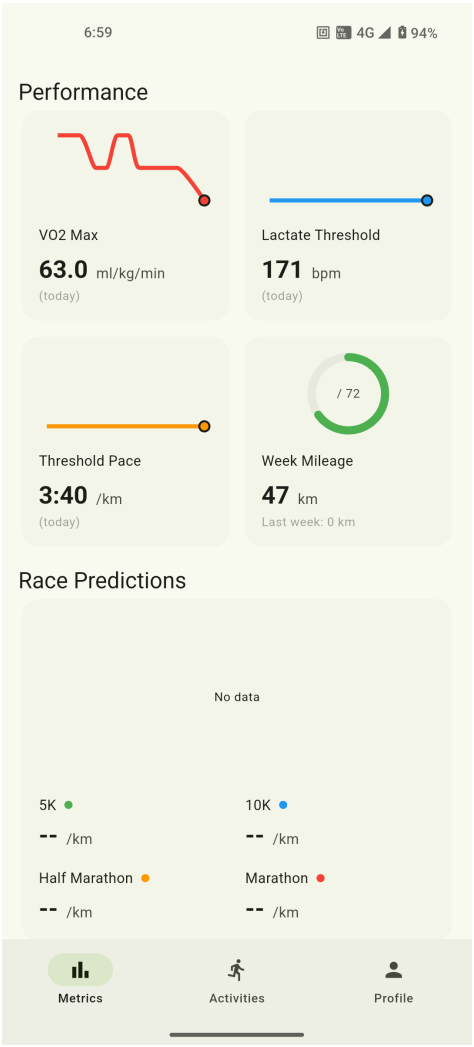


Figure 6.2: Metrics screen: performance indicators (VO_2max, lactate threshold, threshold pace, and weekly mileage progress) followed by the race predictions section.

6.6.1 Conditional HRV Section

HRV measurement requires a wearable capable of detecting beat-to-beat intervals; this is only available on recent generations of sports watches, making missing HRV data the common case. The entire HRV section, including its header, is therefore omitted when `MetricsViewModel.hasHrvData` is `false`, which is preferred over rendering cards with empty or zero values that could be misread as a valid (and alarmingly low) baseline. When present, the section displays Maximum HRV, Resting HRV, and Last Night HRV.

6.6.2 Loading State and Shimmer Placeholders

While metrics data is loading for the first time (`isLoading && !hasData`), every section renders one or more `MetricCardShimmer` widgets in place of the real cards. Shimmer placeholders are sized identically to the real cards they stand in for, so the layout does not shift when data arrives. This prevents the jarring content-pop-in effect that occurs when a spinner is shown over an otherwise empty screen. The distinction between first-load shimmer and pull-to-refresh spinner is enforced by the `isLoading`/`isRefreshing` flag split described in the design chapter (Section 3.5.4).

6.7 Activities Screen

`ActivitiesScreen` displays the complete activity history as an infinite-scrolling list with live client-side search. A search field above the list filters the in-memory list on every keystroke via `ActivityViewModel.updateSearchQuery()`, without issuing additional API requests. Each item is an `ActivityCard` showing a sport-specific icon (selected by `SportUtils.iconForSport()`), the activity name and date, and key statistics (distance, duration, average pace).

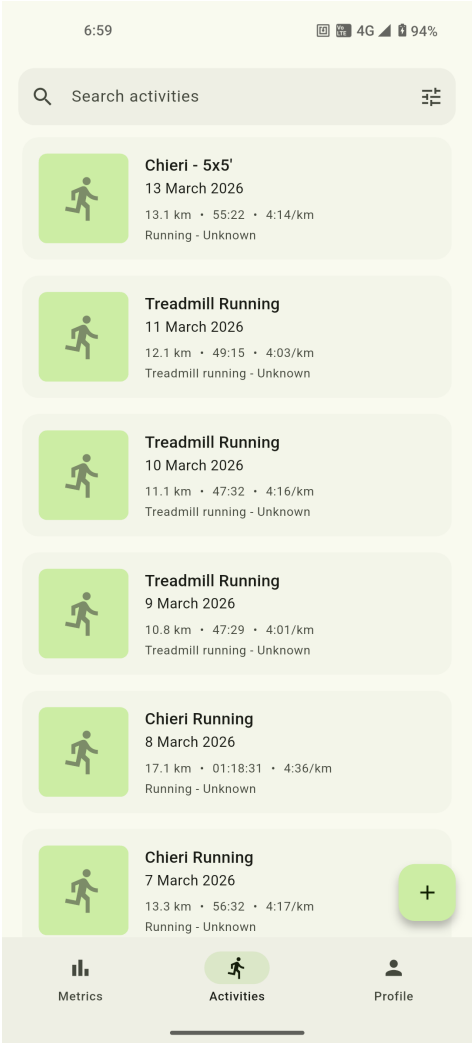


Figure 6.3: Activities screen: scrollable list with live search bar, sport-specific icons, and per-card summary statistics.

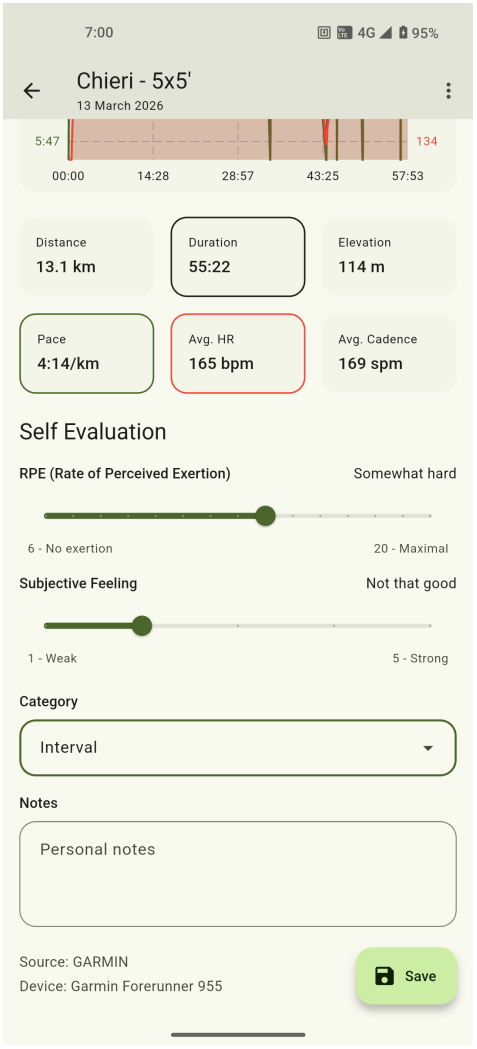
6.7.1 Shimmer Loading and Infinite Pagination

On initial load, the `ListView.builder` renders `ActivityCardShimmer` widgets for a full page. As the user scrolls within 500 px of the list bottom, the scroll listener triggers a paginated fetch by calling `ActivityViewModel.loadMoreActivities()`. The ViewModel fetches the next page of 30 activities and appends them to the cached list. While the next page is in flight, `isLoadingMore` is true and the `itemCount` is temporarily expanded so that `ActivityCardShimmer` widgets appear at the bottom of the real content, giving the user immediate feedback that more data is on the way.

6.8 Activity Details Screen

`ActivityDetailsScreen` is reached by tapping an `ActivityCard`. The route receives the activity ID as a path parameter. The screen scopes its `ActivityDetailsViewModel` instance, as described in Section 3.5.1. A save button appears in the app bar when `hasChanges` is true, and a `PopScope` wrapper intercepts back navigation when there are unsaved changes, showing a confirmation dialog to prevent accidental data loss.

(a) Route map with GPS polyline and dual-series chart with summary cards.



(b) Self-evaluation panel with RPE, subjective feeling, category, and notes fields.

Figure 6.4: Activity Details screen: map and chart panel (left) and editable metadata panel (right).

6.8.1 Chart Panel

The chart panel is implemented as a dedicated reusable widget, `ActivityDetailsChart`, built on `fl_chart` [11]. The active Y-series is one of three `ChartSeriesConfig` options (Pace (min/km), Heart Rate (bpm), and Cadence (rpm)). The selection is controlled by the `ActivityDetailsViewModel`. An optional elevation overlay (`elevationSeries`) is rendered as a background fill behind the selected Y-series when elevation data is available. Pace uses an inverted Y-axis (`isReversed: true`) because a lower value is better.

Below the chart, a two-row grid of selectable summary cards controls the visualization. The first row toggles the X-axis between elapsed time (`ChartXAxis.duration`) and cumulative distance (`ChartXAxis.distance`) and can also enable an elevation overlay. The second row toggles the displayed Y-series between pace, heart rate, and cadence, with up to two Y-series visible at the same time for comparison. Press-

ing and holding a finger on the chart surface reveals a tooltip at that data point and synchronizes the map marker. The tooltip callback `_onTooltipPositionChanged` receives the data point index, retrieves the corresponding `Measurement.time` from `activityDetails.measurements`, and stores it in `_markerTimestamp`, which is passed to the map widget on the next frame, keeping the chart and map synchronized as the user drags their finger across the chart.

6.8.2 Route Map

The route map uses `ActivityRouteMap`, which wraps a `FlutterMap` widget with raster tiles from OpenStreetMap [28] via the `flutter_map` package [12]. The GPS polyline from `activityDetails.route` is drawn as a `Polyline` layer on the map. A circular marker tracks `_markerTimestamp`: when the chart tooltip moves, the marker jumps to the GPS coordinate whose timestamp is closest to the tooltip's current data point.

The map is rendered only when the activity is classified as a GPS sport and the loaded `ActivityDetails` contains GPS data. During loading, the `showMapWhileLoading` flag is passed from the activity list so that a map shimmer is shown only for activities expected to contain a route. For indoor activities such as treadmill runs or stationary cycling, the map section is omitted entirely in the loaded state.

6.8.3 Editable Metadata Panel

Below the chart, grid, and optional map, the screen exposes four editable fields. Perceived Effort uses a slider based on the Borg RPE scale through the `RPEScale` utility. Subjective Feeling uses a second slider backed by `FeelingScale`. The current label is shown next to the field title. Category is chosen through a searchable `DropdownMenu` populated from the user-defined category list managed by `CategoryProvider`. Notes uses a multi-line `TextField` bound to `_notesController`. A clear button appears when the field is not empty. All four fields write to `ActivityDetailsViewModel`. The ViewModel tracks `hasChanges` and enables the save button. Saving invokes `ActivityDetailsViewModel.saveChanges()` and sends a PUT request through the repository layer. When the user leaves the screen, the updated activity is returned to `ActivitiesScreen` through the navigation result passed back by explicitly calling `context.pop(_viewModel.updatedActivity)`. This lets the list update without a second fetch.

6.9 Profile Screen

`ProfileScreen` occupies the third tab (index 2). It renders a `ListView` with two labeled sections.

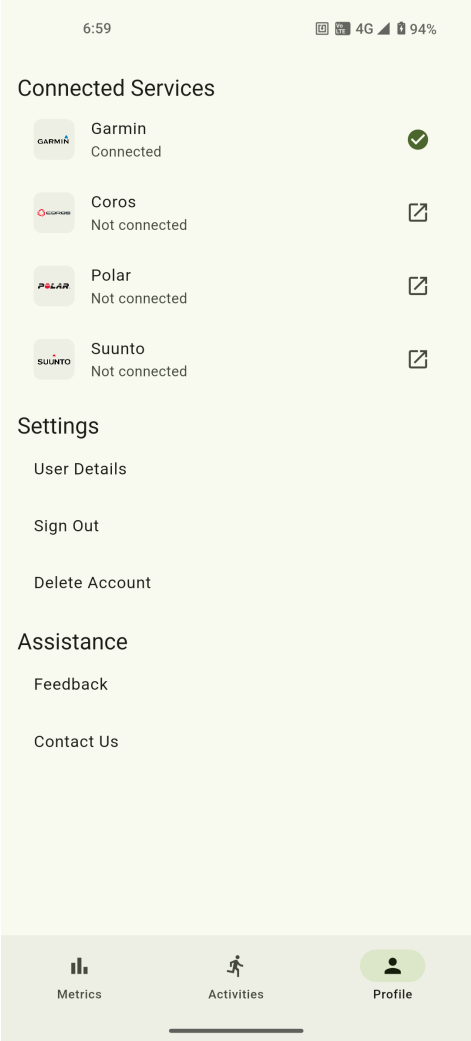


Figure 6.5: Profile screen: Connected Services section with Garmin connected and Coros, Polar, and Suunto not connected, followed by the Settings section.

6.9.1 Connected Services

The first section lists all four wearable providers (Garmin, Polar, Coros, and Suunto) as `ListTile` rows. Each tile's leading widget is the provider's logo image (loaded from `assets/icons/` via `provider.logoAsset`) inside a rounded container whose background color adapts to the theme brightness, ensuring the logo remains legible in both light and dark modes. The trailing widget is `Icons.check_circle` in the primary color when the provider is connected, or `Icons.open_in_new` when it is not. The status text below the provider name reads "Connected" or "Not connected" accordingly. Using both an icon and a text label ensures that the connection state is not conveyed by color alone, satisfying the accessibility requirement discussed in Section 6.12.

Tapping a connected provider first shows a confirmation dialog and only then calls `IntegrationViewModel.disconnect(provider)`. Tapping a disconnected provider calls `IntegrationViewModel.connect(provider)`, which launches the OAuth 2.0 authorization flow in an external browser and instructs the backend to initiate the connection. On return, `OAuthCallbackScreen` processes the callback URI and navigates to the home screen.

6.9.2 Settings

The second section contains three `ListTile` items. "User Details" first pushes to `/user-details`, a read-only profile summary (email, display name, date of birth). An edit button pushes `/user-details/edit`, which exposes the two editable fields (display name and date of birth) behind a `PopScope` unsaved-changes guard. "Sign Out" presents a confirmation dialog and then calls `ProfileViewModel.signOut()`, which triggers the `AuthRepository` sign-out, clears the `ActivityViewModel` and `MetricsViewModel` caches via the `authStateChanges` subscription described in Section 3.5.3, and causes the `_AuthDataGate` to route back to `SignInScreen`. "Delete Account" is styled with the error color to indicate its destructive nature: Google-authenticated users trigger a re-authentication flow before deletion, and email/password users must confirm their current password before the account is permanently removed.

6.10 Responsive Behavior

The application targets portrait-mode phones as its primary form factor. All screens scroll gracefully in landscape orientation. `ActivityDetailsScreen` adjusts layout when `MediaQuery.of(context).orientation == Orientation.landscape`, increasing horizontal padding from 16 px to 128 px:

```
final isLandscape =
    MediaQuery.of(context).orientation == Orientation.landscape;
final pagePadding = EdgeInsets.symmetric(
    horizontal: isLandscape ? 128.0 : 16.0, vertical: 16.0);
```

This centers the chart and map within the middle third of the screen, preventing the chart from spanning the full width of the display on landscape phone screens or small tablets, where very wide charts become difficult to read.

Responsiveness is also handled through `Wrap` in places where content may not fit cleanly on a single row. In `MetricsScreen`, the internal two-column layout uses `Wrap` so cards reflow according to the available width instead of overflowing. Smaller inline data groups use the same approach. For example, the distance, duration, and pace labels in `ActivityCard` are placed in a `Wrap`, which lets them break onto a second line on narrow screens.

6.11 Loading, Error, and Empty States

Every networked screen must handle three non-data states correctly. The strategy applied uniformly across the application is as follows.

The loading state distinguishes first-load from refresh. On first load, the application prefers layout-preserving placeholders over an otherwise empty screen with a spinner. The concrete implementations differ by screen. For example, Section 6.6.2 describes the metrics shimmer cards, while Section 6.7.1 describes the activity-list placeholders. On pull-to-refresh, the `RefreshIndicator` spinner is shown at the top of the existing populated content, which remains visible and interactive.

The error state is designed around visible recovery paths. Retryable failures show a dialog with an explicit Retry action. Non-retryable failures show informational feedback

without implying that the same action should simply be attempted again. Less severe issues, such as validation or authentication feedback, are surfaced through transient SnackBars instead of blocking the screen. This keeps the interface responsive while still making failure states clear to the user.

The empty state is handled explicitly in `ActivitiesScreen`: when the current `ActivityViewModel.filteredActivities` list is empty and `isLoading` is false, a dedicated `_buildEmptyState()` widget is shown (an icon with explanatory text) rather than a blank scroll area. This ensures the user can distinguish "no activities recorded yet" from "still loading".

6.12 Accessibility

Accessibility was addressed through four concrete practices. First, tonal color contrast between text and surface is derived from Material 3 color roles in both light and dark schemes, as noted in Section 6.2. Second, some icon-only actions expose explicit `Tooltip` labels, including the filter button in the Activities search bar and the edit action in User Details. Third, non-text visual elements can expose semantic labels: the reusable `Logo` widget is wrapped in a `Semantics` widget with the label "Endurance Lab logo". Fourth, the interface does not rely on color alone to convey state. In the Profile screen, provider status is shown through both a colored icon and a text label ("Connected" or "Not connected"). Text throughout the app uses the Material 3 type scale rather than hard-coded pixel sizes, so it follows the system font settings.

6.13 Figma Mockups vs. Final Implementation

Figure 6.6 shows the Activity Details screen as designed in Figma alongside the shipped implementation.

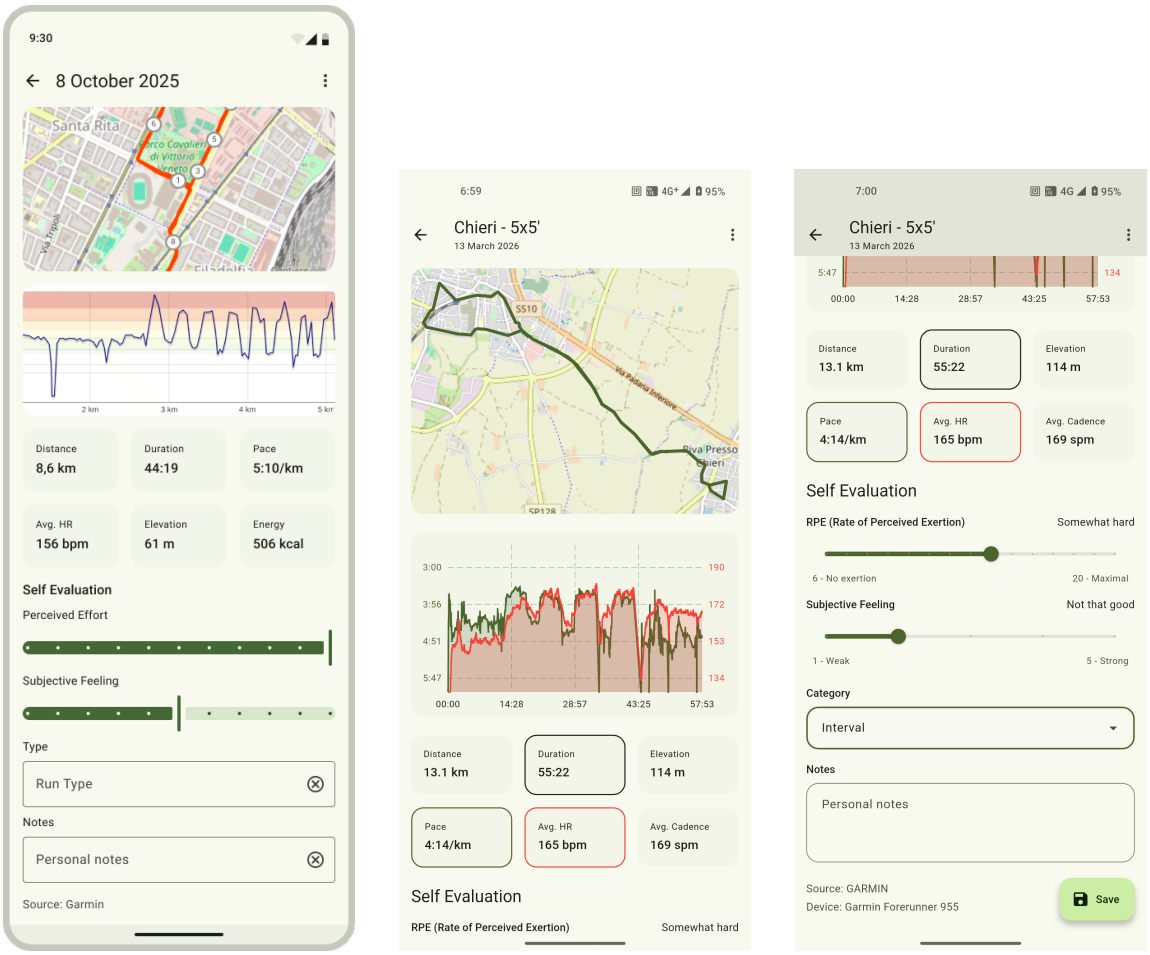


Figure 6.6: Activity Details screen: Figma mockup (left) vs. shipped implementation split across the map/chart panel (centre) and the self-evaluation panel (right).

Two deliberate deviations between the Figma mockups and the shipped implementation are also worth documenting. The first concerns the activity card thumbnail: the original Figma design showed a small GPS route map rendered inline in each card. During implementation, loading a map for every visible card proved too slow, so the thumbnail was replaced by a sport-specific icon. A better future solution would be to generate lightweight static thumbnails on the backend, because the activity-list payload does not contain the detailed route data needed to render the preview directly in the client. The second deviation concerns the activity filter panel: the Figma prototype included a bottom sheet for filtering activities by sport type. This functionality was deferred because the current search already covers multiple fields in the activities that have been fetched locally, including name, category, notes, date, distance, duration, pace, and ID. The more useful future improvement is therefore a backend-powered search and filter layer that works on the full activity history rather than only on the pages already loaded in memory.

Chapter 7

Conclusion

This thesis set out to address a structural gap in the wearable sports data landscape: the absence of a vendor-neutral mobile platform capable of unifying training data from heterogeneous device manufacturers into a single normalized view, while simultaneously serving as a continuous research data infrastructure for longitudinal sports science. This chapter summarizes the contributions made toward that goal, reflects on the limitations that remain, and proposes concrete directions for future development.

7.1 Summary of Contributions

Five research objectives were stated in Section 1.5. Each is addressed in turn.

7.1.1 Objective 1: Cross-Platform Architecture

The first objective was to design and implement a cross-platform mobile architecture capable of collecting, normalizing, and presenting wearable training data from multiple heterogeneous providers within a single unified application. The architecture described in Chapter 3 achieves this through a four-layer structure whose central property is a strict unidirectional dependency rule: the View layer depends on ViewModels, ViewModels depend on Repositories, Repositories depend on Services, and Services depend on external systems. No layer references a layer above it. This discipline makes each layer independently testable and replaceable, and it was maintained consistently across all seven ViewModels and six Repositories that constitute the application. The use of the sealed `Result<T, E>` type rather than exceptions throughout the service and repository layers is a concrete consequence of this architecture: it forces every failure path to be handled explicitly, with the Dart compiler rejecting any call site that does not address both the `Success` and `Failure` cases. The application was built with Flutter, targeting Android as its primary delivery platform, and was successfully deployed to Google Play via the CI/CD pipeline described in Chapter 4. The iOS platform is architecturally supported by the same codebase but falls outside the delivered scope of this work.

7.1.2 Objective 2: Provider Integration and Extensibility

The second objective was to validate the architecture through complete integrations with both Garmin and Polar, establishing the abstraction patterns that Coros and Suunto integrations can follow. The Garmin integration was fully validated end-to-end

against physical hardware: the complete pipeline from OAuth authorization through data ingestion, normalization, and client-side presentation was exercised with real device data during the beta testing period described in Chapter 5. The Polar integration is fully implemented at the code and API contract level, covering the same data types as the Garmin integration via the same provider abstraction. However it could not be exercised against a physical device due to hardware unavailability during the testing phase, and end-to-end validation is deferred to future work. The extensibility claim is supported by the structure of `IntegrationProvider`: the enum-based design means that adding a new provider requires only a new enum case. No changes are required to any repository, service, ViewModel, or screen. This constraint was verified for both the Garmin and Polar implementations, each of which was introduced without modifying any pre-existing client component.

7.1.3 Objective 3: Raw-Data Metric Derivation

The third objective was to derive performance metrics such as VO_2max, lactate threshold, and race time predictions directly from raw data rather than relying on the proprietary algorithms of each watch manufacturer. Because the heavy mathematical computation and cross-provider data normalization happen entirely on the backend, the mobile application's responsibility is purely presentational. It consumes a single, unified API where all brand-specific quirks have already been stripped away. To handle this cleanly, the mobile client implements a strict three-tier data pipeline: Data Transfer Objects (DTOs) mirror the backend's generic JSON, Domain models apply standard unit conversions (such as meters per second to minutes per kilometer), and Presentation models prepare the exact strings and colors for the UI. As a result, the app displays fitness metrics computed from the exact same baseline regardless of whether the athlete recorded their run with a Garmin or a Polar device, enabling consistent cross-device comparisons that are impossible within any single manufacturer's closed ecosystem.

7.1.4 Objective 4: Continuous Research Data Infrastructure

The fourth objective was to build a continuous research data infrastructure enabling passive, longitudinal collection of training data without the fixed-period sampling constraints that characterize traditional sports science studies. The backend database schema comprises nine tables organized into four functional groups: user identity and consent, OAuth session state, activity data at three levels of granularity (summary, GPS track, and physiological time series), and computed health and performance metrics. Data accumulates passively as users pursue their training, with no action required beyond the initial provider connection. The `allow_past_data` consent field in the `users` table controls whether the backend backfills historical data on first connection, giving participants explicit control over the scope of their contribution to the research dataset.

At the close of the beta period, the platform had accumulated data from 8 athletes over a period of three and a half months. The dataset comprised 1,001 activity records with a combined 2,940,498 GPS track points and 1,723,761 physiological time-series samples, 621 daily health metric rows, 589 performance metric snapshots, and 589 race prediction snapshots. All athletes connected exclusively through Garmin during the testing period. The dataset is narrow in provider coverage but longitudinal in depth for

participants who remained active throughout the beta, constituting a starting point for the PhD student's research program rather than a complete research corpus.

7.1.5 Objective 5: Production-Grade CI/CD Pipeline

The fifth objective was to demonstrate a complete production-grade CI/CD pipeline for a Flutter Android application, covering automated testing, build flavor management, signing key handling, and deployment to Google Play. The pipeline described in Chapter 4 achieves this through a four-job GitHub Actions workflow organized in a strict linear dependency: the `test` job runs unconditionally on every trigger and blocks all subsequent work on failure. The `build_apk_prod` job produces a signed APK restricted to pushes on `dev`. The `gh_release` job publishes a GitHub Release with the APK attached. Finally, the `play_deploy_internal` job submits a signed Android App Bundle to the Google Play internal track as a draft, requiring explicit manual promotion before any user reaches the build. Environment isolation is enforced at compile time by injecting `APP_ENV` via `--dart-define` and relying on Dart's constant folding to inline the value into the binary, preventing a production artifact from communicating with the development backend regardless of any runtime configuration change. All credentials (the Android keystore, Firebase configuration files, and Google Play service account key) are stored as encrypted GitHub Actions secrets and are materialized on the runner's filesystem only during the specific job step that requires them, never committed to the repository and never transferred between jobs.

7.1.6 Emergent Outcome

Beyond the five stated objectives, this work demonstrated that adapting an agile, Scrum-inspired methodology (described in Chapter 2) is highly effective for a solo-developer research project. By consolidating formal sprint ceremonies into a single bi-weekly alignment meeting with the PhD researcher, the project maintained strict alignment between the software architecture and the evolving scientific requirements without the administrative overhead typical of enterprise Scrum. This iterative approach allowed the software to be tested continuously by a small, engaged cohort during the beta testing execution described in Chapter 5. This real-world usage produced concrete usability improvements (such as the reflowing activity card layout for large system font scales and the expanded 30-day metrics window for lower-frequency athletes), demonstrating that continuous feedback loops surface edge-cases that neither upfront requirements analysis nor isolated developer testing can reveal.

7.2 Limitations

Several limitations constrain the scope of what has been demonstrated.

The most significant functional limitation is the state of the test suite. The CI/CD pipeline enforces a test-passing gate before any build can be produced, but the test suite itself consists of a single placeholder assertion. The infrastructure for testable business logic is entirely in place: every ViewModel receives its dependencies by constructor injection and extends `ChangeNotifier`, making it straightforward to instantiate and exercise in pure Dart unit tests with mock repositories substituted for real implementations. The missing piece is an high enterprize-grade coverage, enlarging the suite

is the most direct path to closing the gap between the quality gate that the pipeline nominally enforces and the quality guarantee that the pipeline actually provides.

The absence of runtime observability is the second structural gap. No crash reporting integration was established: errors occurring on production devices are not collected, aggregated, or surfaced back to the development workflow. During the beta period this gap was tolerable because the cohort was small and tester-reported issues were sufficient, but at any meaningful scale, unreported failures go entirely undetected. Firebase Crashlytics [23] is identified as the most direct remedy, providing automatic exception collection without requiring changes to the client's architecture.

On the provider side, the Polar integration's inability to complete end-to-end hardware testing leaves one aspect of the extensibility claim unverified. The abstraction pattern has been demonstrated to work for two providers at the API contract level, but until real Polar device data flows through the complete pipeline (OAuth authorization, backend ingestion, normalization, and client-side presentation), the possibility of integration-level issues not captured by testing cannot be fully excluded.

The dataset accumulated during the beta reflects a single-provider cohort. Because all participants used Garmin devices, the research dataset does not yet exercise the cross-provider normalization and comparison capabilities that are among the platform's central claims. Provider-specific nullability patterns and normalization decisions have not been validated under the conditions they were designed for: a user who trains with both a Garmin and a Polar device, viewing consistent metrics derived from two independently normalized sources.

7.3 Future Work

The most immediate continuation tasks are those that address the structural gaps identified above. The first is the completion of the test suite. Widget tests impose maintenance overhead proportional to the rate of UI change and are therefore most appropriately introduced once the interface stabilizes, but unit tests for the ViewModel layer can be written now without any such qualification: the mock repository infrastructure is in place, and the business logic in `ActivityViewModel`, `MetricsViewModel`, `AuthViewModel`, and `IntegrationViewModel` is specific enough to generate meaningful test cases. The second is the integration of Firebase Crashlytics, which requires only dependency addition and SDK initialization and would immediately close the runtime observability gap.

In parallel, a dedicated GDPR assessment should be done before a public release. This includes publishing a complete privacy notice and retention policy, exposing a rights center in the application for access, correction, deletion, and export requests, and mapping each processing operation to a documented lawful basis. Because the platform processes health-related training data, a formal legal review should also determine whether additional controls such as a Data Protection Impact Assessment are required.

Provider expansion is the priority for both the consumer product and the research dataset. The Polar integration requires end-to-end hardware validation once a device becomes available. This is not an architectural task but a testing and debugging one, and the existing implementation provides the starting point. Coros and Suunto integrations require implementing the provider-specific OAuth and data retrieval logic in the backend, after which the client-side addition is bounded to a new enum case. The research dataset's representativeness improves directly with each additional provider

that contributes data, because it is only with multi-provider data that the cross-device normalization claims can be verified against real measurements.

The iOS delivery path is architecturally unobstructed. The Flutter codebase makes no Android-specific assumptions, and the `AppEnv` environment injection and Firebase initialization strategy are platform-agnostic. The work required to complete iOS delivery is operational rather than architectural: establishing Apple Developer program membership, configuring provisioning profiles and certificates, extending the CI/CD pipeline with a macOS runner and Fastlane integration with App Store Connect, and verifying the build on physical iOS hardware.

Beyond closing these gaps, two concrete implementation improvements would increase the platform's robustness. The first is the integration of the Play Core In-App Updates API through the `in_app_update` package, which would provide a mechanism to enforce a minimum client version when a future backend change requires a breaking API update. The current implementation relies entirely on the Play Store auto-update mechanism and has no fallback if a tester does not update before a breaking change is deployed. The second is backend-powered activity search and filtering. The current client-side search operates only on activities already loaded in memory, meaning it cannot find an activity that has not yet been fetched from the server. A backend search endpoint accepting query and filter parameters would address this within the existing `ActivityViewModel.updateSearchQuery()` entry point, without requiring changes to the ViewModel interface or any layer above it.

7.4 Final Remarks

The landscape that motivated this work is a consequence of commercial incentives rather than technical necessity. Wearable manufacturers have no structural reason to make their data portable, and the result is that the richest corpus of real-world athletic training data ever generated is, for most practical purposes, inaccessible: siloed within proprietary platforms, owned by manufacturers rather than athletes, and unavailable in the normalized form required for cross-device research. This work demonstrates that the technical barriers to changing this situation are not prohibitive. A single codebase, built by one developer over the course of a master's thesis, can collect, normalize, and present data from multiple wearable manufacturers in a form that is both genuinely useful to athletes and structurally suitable for longitudinal research.

The dual-purpose design demonstrated in this work illustrates how consumer utility and research data collection can be mutually reinforcing. By offering athletes a vendor-neutral platform that normalizes their data and computes cross-device performance metrics, the application organically attracts the longitudinal user base that sports science researchers require. In turn, the rigorous data normalization mandated by the research infrastructure provides the foundation for the application's consumer-facing features. The initial dataset, athlete cohort, and operational platform established during the beta period provide the immediate foundation for the PhD student's upcoming research program. Moving forward, the scalability of this model will depend on continuous provider expansion and user base growth, but the architectural infrastructure required to support that growth is now fully operational.

Bibliography

[1] Scrum.org. *What is Scrum?* https://www.scrum.org/learning-series/what-is-scrum/. 2026.

[2] JetBrains s.r.o. *YouTrack — Project Management Tool*. https://www.jetbrains.com/youtrack/. 2026.

[3] Flutter Team. *Simple App State Management — Flutter Documentation*. https://docs.flutter.dev/data-and-backend/state-mgmt/simple. 2026.

[4] Flutter Team. *Flutter App Architecture Guide*. https://docs.flutter.dev/app-architecture/guide. 2026.

[5] rrousselGit. *provider — Dart Package*. https://pub.dev/packages/provider. 2026.

[6] Flutter Team. *go_router — Dart Package*. https://pub.dev/packages/go_router. 2026.

[7] Flutter Team. *Deep Linking — Flutter Documentation*. https://docs.flutter.dev/ui/navigation/deep-linking. 2026.

[8] Google LLC. *Get Started with Firebase Authentication on Flutter*. https://firebase.google.com/docs/auth/flutter/start. 2026.

[9] Flutter Team. *google_sign_in — Dart Package*. https://pub.dev/packages/google_sign_in. 2026.

[10] Flutter Team. *Material Design for Flutter*. https://docs.flutter.dev/ui/design/material. 2026.

[11] imaNNeo. *fl_chart — Dart Package*. https://pub.dev/packages/fl_chart. 2026.

[12] fleaflet. *flutter_map — Dart Package*. https://pub.dev/packages/flutter_map. 2026.

[13] GitHub, Inc. *GitHub Actions Documentation*. https://docs.github.com/en/actions. 2026.

[14] timheuer. *base64-to-file: GitHub Action to decode a base64 string to a file*. https://github.com/timheuer/base64-to-file. 2026.

[15] softprops. *action-gh-release: GitHub Action for creating GitHub Releases*. https://github.com/softprops/action-gh-release. 2026.

[16] subosito. *flutter-action: Flutter environment for use in GitHub Actions*. https://github.com/subosito/flutter-action. 2026.

[17] Google LLC. *Android App Bundle — Android Developers*. https://developer.android.com/guide/app-bundle. 2026.

[18] Fastlane Contributors. *Fastlane Documentation*. https://docs.fastlane.tools. 2026.

[19] Jems22. *fastlane-plugin-increment_version_code*. https://github.com/Jems22/fastlane-plugin-increment_version_code. 2026.

[20] Google LLC. *Google Play Developer API*. https://developers.google.com/android-publisher. 2026.

[21] Google LLC. *In-app updates — Android Developers*. https://developer.android.com/guide/playcore/in-app-updates. 2026.

[22] Pavel Sulimau. *in_app_update — Dart Package*. https://pub.dev/packages/in_app_update. 2026.

[23] Google LLC. *Firebase Crashlytics Documentation*. https://firebase.google.com/docs/crashlytics. 2026.

[24] Google LLC. *Firebase Performance Monitoring Documentation*. https://firebase.google.com/docs/perf-mon. 2026.

[25] Figma, Inc. *Figma — The Collaborative Interface Design Tool*. https://www.figma.com. 2026.

[26] Google LLC. *Material 3 Design Kit — Figma Community*. https://www.figma.com/community/file/1035203688168086460. 2026.

[27] Google LLC. *How the System Works — Material Design 3*. https://m3.material.io/styles/color/system/how-the-system-works. 2026.

[28] OpenStreetMap Contributors. *OpenStreetMap*. https://www.openstreetmap.org. 2026.