

Bachelor's Thesis

Department of Computer Science
OST - University of Applied Sciences
Campus Rapperswil-Jona

Spring Term 2026

Version: 2026-06-12

Companion App for Portable Medication Cooler

Developing a Monitoring and Controlling Mobile Application

Authors:	Nina Grässli Jannis Tschan
Advisors:	Prof. Laurent Metzger Olivier Lischer
Project Partner:	Lukas Reitemeier Nicolas Wirz Adam Ohnesorge
External Co-Examiner:	Simon Göldi
Internal Co-Examiner:	Prof. Dr. Andreas Peter

Abstract

Temperature-sensitive medication, such as biologics, must be stored within a specific temperature range to maintain effectiveness. This can be challenging during travel, as current portable cooling solutions often lack the reliability and battery life required for longer trips. The industry partner has developed a portable medication cooler capable for maintaining a set temperature range for multiple days. To provide transparency regarding storage conditions and support future product development and data analysis, the industry partner requested a companion application and telemetry backend.

The objective of this Bachelor's Thesis was to design and implement a mobile application for monitoring and controlling the medication cooler, as well as a backend for storing and aggregating cooler data. The app enables users to verify the current storage conditions of their medication, view temperature data over time and adjust selected cooler settings. The backend provides the technical foundation for future analysis of cooler and usage data, for example for predictive maintenance and optimisation of cooling behaviour.

The mobile application was developed using Flutter and Dart and communicates with the cooler via Bluetooth Low Energy. It displays the cooler's current status, including temperature, battery percentage and remaining battery time. Furthermore, it visualises historical temperature data in charts and allows users to modify settings such as the operational temperature range. The app follows the MVVM architecture, separating user interface, data handling and communication logic to support maintainability and extensibility.

The backend was implemented with Rust and Axum and uses a PostgreSQL database to persist cooler data. Together, the app and backend form a functional prototype for continued development by the industry partner. Software quality is supported by unit tests, strict linting, CI/CD pipelines and code documentation. The pipelines automatically verify the application and generate Android debug and release builds. In addition, two usability tests were conducted to evaluate the app's usability and identify areas for improvement.

The central requirements of the project were fulfilled to varying degrees. Core features such as Bluetooth connection, live status display, historical data synchronisation, temperature history visualisation and telemetry transfer to the backend were successfully implemented. Some features, including a "Trips" page, push notifications and detailed alarm configuration, were removed from the final scope due to time constraints, changing industry partner priorities and findings from usability testing.

The result of this thesis is a working companion application and backend prototype. The system demonstrates the core interaction between the cooler, app and backend. It provides users with transparency regarding medication storage conditions and offers the industry partner a solid technical basis for further development.

Table of Contents

Abstract	i
Table of Contents	ii
1 Introduction	1
1.1 Situation	1
1.2 Objectives	1
1.2.1 Mobile Application	1
1.2.2 Server Backend	1
1.3 Conditions	1
2 Research	2
2.1 Market Research	2
2.1.1 Analysis of the OHealth App	2
2.2 Requirements	4
2.2.1 Functional Requirements	4
2.2.2 Non-Functional Requirements	9
3 Design and Architecture	13
3.1 System Context	13
3.1.1 Cooler	13
3.1.2 App	14
3.1.3 Backend	15
3.1.4 Development Environment	15
3.1.5 Development and Domain Constraints	15
3.2 Domain Model	16
3.3 C4 Model	17
3.3.1 System Context	17
3.3.2 Container Context	18
3.3.3 Component Context	18
3.4 Backend Database	19
3.4.1 Technology	19
3.4.2 Database Schema	19
3.4.3 Requirement Fulfilment	20
3.4.4 Considered Alternatives	22
3.4.5 Limitations of the Current Schema	22
3.4.6 Scalability	22
3.5 Local Database	23
3.5.1 Technology	23
3.5.2 Database Schema	23
3.5.3 Requirement Fulfilment	24
3.5.4 Limitations of the Current Schema	24
3.6 Design Specifications	25
3.6.1 Wireframes	25
4 Implementation	27
4.1 App	27
4.1.1 A Brief Introduction to Flutter MVVM	27
4.1.2 Implementation Overview	28
4.1.3 Bluetooth Functionality	30
4.1.4 Live Data Processing	33
4.1.5 History Synchronisation	37

4.1.6 Settings	41
4.1.7 Send Telemetry to Backend	44
4.2 Backend	46
4.2.1 Introduction to Gerust	46
4.2.2 Database implementation	47
4.2.3 API	48
4.2.4 Limitations and Conclusion	49
5 Quality Assurance	50
5.1 Guidelines	50
5.1.1 Version Control	50
5.1.2 Git Strategy	50
5.1.3 Issues	51
5.1.4 Definition of Ready (DoR)	51
5.1.5 Definition of Done (DoD)	51
5.1.6 Time Tracking	52
5.1.7 Code Guidelines	52
5.1.8 Documentation Guidelines	56
5.2 Testing	56
5.2.1 Unit Tests	56
5.2.2 Widget Tests	58
5.2.3 Integration Tests	59
5.2.4 Manual Testing	62
5.2.5 Usability Testing	63
5.3 CI/CD	64
5.3.1 Documentation Pipeline	64
5.3.2 App Pipeline	65
5.3.3 Backend Pipeline	66
5.3.4 GitLab Pages Deployment	69
5.4 External Code Reviews	70
6 Results	72
6.1 Fulfilment of Functional Requirements	72
6.1.1 User Story 1 – Establish Connection to Cooler	72
6.1.2 User Story 2 – System Status	73
6.1.3 User Story 3 – Continuous Data Retrieval from Cooler	73
6.1.4 User Story 4 – Send Collected Cooling Data to the App	74
6.1.5 User Story 5 – Status Notification	75
6.1.6 User Story 6 – Temperature and Battery History	75
6.1.7 User Story 7 – Trips	77
6.1.8 User Story 8 – Set Temperature Range	77
6.1.9 User Story 9 – Rapid Cooling	78
6.1.10 User Story 10 – Configure Alarms	79
6.1.11 User Story 11 – Device-specific Data Retrieval	79
6.1.12 User Story 12 – Aggregated Sensor Data	80
6.1.13 User Stories About Possible Extensions	80
6.1.14 Summary	81
6.2 Fulfilment of Non-Functional Requirements	82
6.2.1 NFR1: Maintainability – Testability	82
6.2.2 NFR 2: Interaction Capability – Learnability	84
6.2.3 NFR3: Functional Suitability – Functional Correctness	85
6.2.4 NFR4: Reliability	86

6.2.5 NFR5: Safety – Hazard Warning	86
6.2.6 NFR6: Maintainability – Analysability	87
6.2.7 Summary	88
6.3 Conclusion	88
6.4 Outlook	89
A Project Plan	90
A.1 Project Management	90
A.1.1 Project Plan	90
A.1.2 Milestones	91
A.1.3 Meetings	92
A.2 Risk Management	92
A.3 Tooling	94
A.3.1 Code	95
A.3.2 Documentation	95
A.3.3 Pipeline	95
A.3.4 AI Tools	96
B Usability Testing	97
B.1 Questionnaire	97
B.1.1 Warm-up Questions	97
B.1.2 Task Walkthrough	97
B.1.3 Follow-up Questions	98
B.2 Protocols	99
B.2.1 First Usability Test	99
B.2.2 Second Usability Test: Expert Evaluation	102
C External Code Reviews	104
C.1 Code Review by Clemens Meier	104
C.1.1 Initial Response, 20.04.2026	104
C.1.2 Code Review, 22.04.2026	104
C.1.3 Clarification Response, 23.04.2026	106
D Glossary and List of Abbreviations	108
D.1 Glossary	108
D.2 List of Abbreviations	109
E List of Tables	110
F List of Figures	111
G List of Listings	111
H References	113

1 Introduction

1.1 Situation

The industry partner has been developing a portable medication cooler for use with temperature-sensitive medication such as biologics. Its aim is to provide users with a reliable and long-lasting solution to keep their medication at a temperature between 2 and 8°C during multi-day travel trips. Existing solutions for this scenario have been identified as insufficient in their cooling reliability and battery life by both the industry partner and users of those solutions. The main issue is that the medication may lose its effectiveness if it is not properly cooled for a period of time. This is crucial because injecting poorly-cooled medicine or stopping the treatment altogether while on holiday can severely affect the patient's health.

To help achieve the goals stated above, the medication cooler should be bundled with a mobile app that allows users to verify the reliability of the storage conditions by checking temperature and humidity data over time. Various features like adjustment of the temperature range and viewing status information of the cooler are also part of the application. By creating an app, the industry partner aims to increase customer confidence in the product and distinguish themselves from their competition that do not offer this functionality.

1.2 Objectives

The aim of this Bachelor's thesis is to develop the mobile application as described in [Section 1.1](#). The project will be split into two distinct parts. After the conclusion of the Bachelor's thesis, the development will be handed over to the industry partner, who will continue to develop the application.

1.2.1 Mobile Application

The mobile application should display the current and past temperature data from the medication cooler in easy-to-understand diagrams. The app should also handle interaction with the cooler's settings and notify the user of its status such as the remaining battery life. Communication with the cooler should use the Bluetooth Low Energy (BLE) protocol. Most data processing will be handled by the app, not by the cooler itself. Given the requirement for the mobile application to be available for both iOS and Android operating systems, the industry partner has requested that the application be written using the cross-platform Flutter framework in the Dart programming language [1].

1.2.2 Server Backend

The server backend should allow the industry partner to aggregate usage data from all medication coolers currently in use. The industry partner should be able to use this data for predictive maintenance and optimal control modelling. The industry partner has also submitted technology requests for the backend. The choice was between Rust with the Axum framework [2] or Go with the Gin framework [3]. The team was satisfied with their experience of using Rust during their Term Paper and consequently decided to continue to use it.

1.3 Conditions

The project is carried out as part of a Bachelor's thesis as specified by the module "Bachelor-Arbeit Informatik" at OST [4]. The focus is on the evaluation, design and development of the mobile application and the server backend specified in [Section 1.2](#).

- **Type of project:** Bachelor's thesis
- **Work hours:** 360 hours per person, 720 hours total
- **Rewarded ECTS:** 12 ECTS per person

2 Research

2.1 Market Research

Before defining the product requirements, this chapter will analyse existing similar solutions to gather information about customer expectations and design the app's UX (user experience) based on these findings.

The industry partner's competitors do not offer a companion app for their coolers. Consequently, there are no equivalent products that can be examined. The team has instead opted to analyse the UX of apps that offer similar functionality to the app created in this project: Connection to Bluetooth devices and collection of health data.

The largest group of such apps consists of companion apps for wearables. Given that one team member owns a OnePlus Watch 2, its companion app "OHealth" was examined in closer detail. However, the analysis was conducted in a minimal way with no user testing, as this would exceed the scope of the project. The primary objective of the analysis was to evaluate the setup of the Bluetooth connection and the overall structure of the app.

2.1.1 Analysis of the OHealth App

OHealth is the companion app for OPPO and OnePlus smart wearables [5]. The provider is HeyTap, the provider of internet services for OPPO, OnePlus and realme, based in Singapore [6]. Once the app has been downloaded and paired with the wearable via Bluetooth, the wearable will be able to access the phone's data, including notifications, contacts, call history and messages. Similarly, the app will be able to record and visualise health statistics generated by the wearable [5].

2.1.1.1 First Run Wizard

The setup process of the OHealth app is well-guided and easy to follow. It uses a sequence of screens to guide the user through the required steps to connect with the wearable device and ask for the required permissions, as illustrated in [Figure 1](#). This is relevant for the companion app, as its first run wizard will require the "Nearby Devices" permission to connect to the cooler via Bluetooth Low Energy. However, the process could be improved by showing how many steps remain in the process.

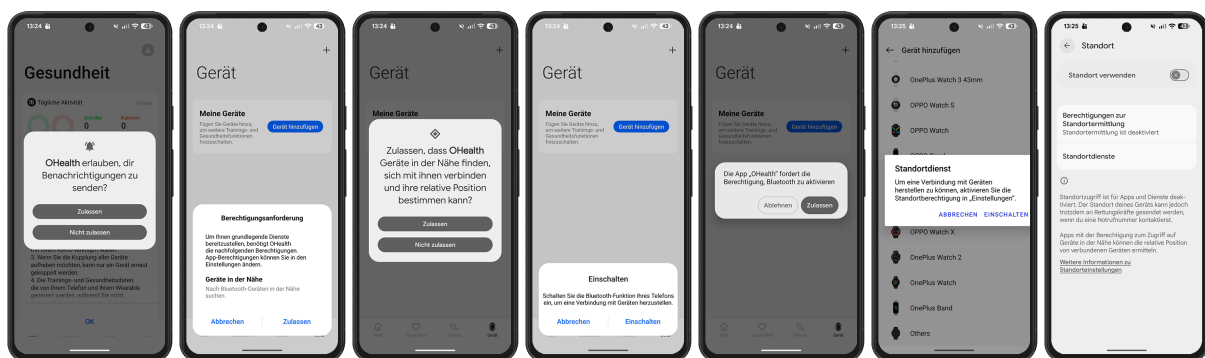


Figure 1: First run wizard of the OHealth app, guiding the user through connecting with their wearable

2.1.1.2 Main App Structure

Once the wearable has been connected, the main structure of the app is presented. The app is divided into four main screens, as seen in [Figure 2](#): Home, Health, Fitness and Device. The home screen displays various statistics collected by the device in different boxes, such as activity level, heart rate and sleep quality. Tapping on one of these boxes opens a detail page. This card-based structure is useful because it allows users to quickly scan important information while offering access to more detailed data when needed.



Figure 2: The four main screens in the OHealth app

The health screen provides an overview of the activities for the current day, such as walks and progress towards user-defined goals. The fitness screen keeps track of individual activity sessions in different sports and offers a summary of all sessions in the current month. On the devices page, the wearable can be configured.

Although the card-based design is well thought out, the app shows some visual inconsistencies across screens, as can be seen in [Figure 2](#). For example, title styling, profile icon visibility, and the use of cards vary between screens. Such inconsistencies should be avoided when creating the companion app for the medication cooler to create a more predictable user experience.

2.1.1.3 Data Visualisation

[Figure 3](#) shows a detail page with statistics in the OHealth app. A segmented button is used to select different time periods. The level of detail in the chart changes if a longer time segment is displayed. The page also shows information below the chart, such as the average or highest value.

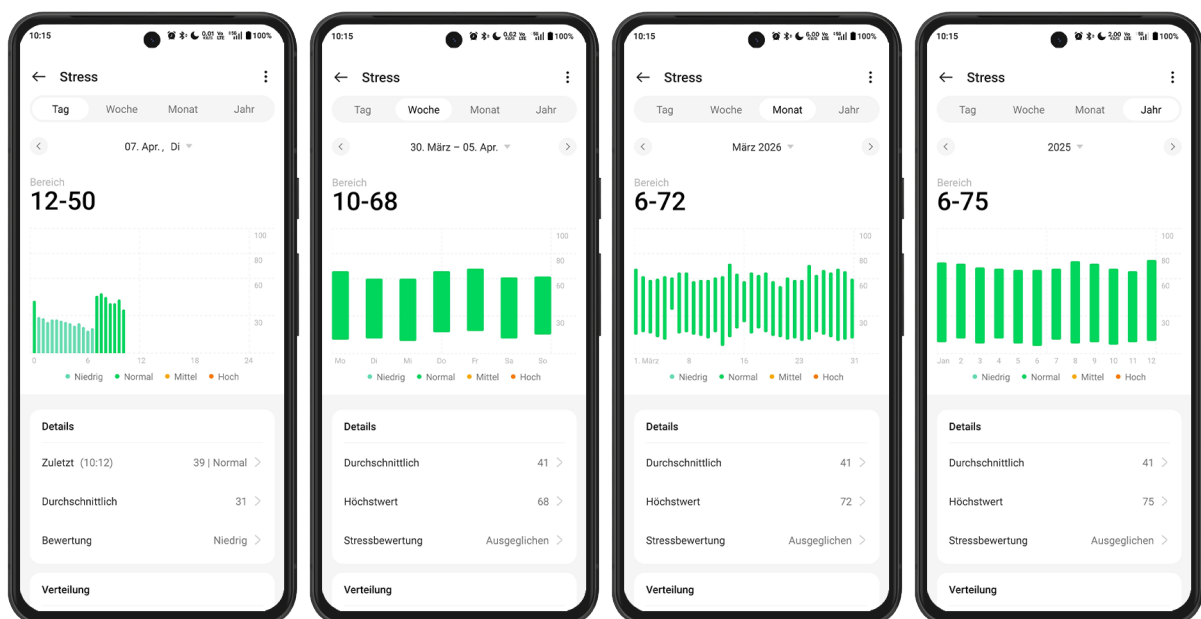


Figure 3: Detail page with statistics in the OHealth app

Most charts within the app can be viewed in greater detail by opening them in landscape mode. Tapping on a specific data point opens a small info pop-up which contains the time and date at which the data point was captured as well as its value. This can be seen in [Figure 4](#).



Figure 4: Chart in OHealth in landscape mode

These patterns are relevant for the companion app because users may need to inspect temperature data at varying levels of detail. For instance, a daily overview may be beneficial for quick monitoring, whereas a detailed chart may be required when investigating a temperature deviation.

2.1.1.4 Key Findings and Design Implications

This chapter provides a summary of the key findings and their implications for the companion app.

Findings from OHealth	Implication for medication cooler companion app
Setup is clear and well-guided	Use a step-by-step first-run wizard
Remaining setup steps are not visible	Add a progress indicator
UI consistency varies across screens	Define consistent UI over all pages
Important data is shown in cards	Use cards for key information
Charts can be viewed at various levels of detail	Let users switch between different levels of details in the charts
Activities let users track a particular time period	Allow users to create custom time periods during which they may wish to monitor the temperature more closely
Detail pages provide deeper insights	Allow users to tap on cards to view more details
Data points reveal additional information	Let users tap chart points to view exact temperature and timestamp

Table 1: Key findings from the market analysis

2.2 Requirements

This chapter outlines the requirements for the developed product. The functional requirements are illustrated with user stories that model the product's functions. Non-functional requirements, on the other hand, are used to describe the quality characteristics of the product.

2.2.1 Functional Requirements

The functional requirements (FRs) are defined as user stories (US) according to the structure "as <user>, I want to <action> so that <effect>" [7]. In total, the team has formulated 25 user stories in cooperation with the industry partner. The user stories 1 to 12 represent the features of the minimum viable product (MVP) and are formulated in more detail with acceptance criteria.

2.2.1.1 Target Audience

The project has two different kinds of users.

- **App users:** The primary target audience consists of customers of the medication cooler. They want to use the app on their mobile device to access information about the cooler and control it (e.g. turn the cooler on/off, set the cooling temperature range). This user group is highly diverse, spanning age demographics and technological expertise levels.
- **Admin users:** Members of the industry partner, who need to be able to export data about the medication coolers from the backend.

2.2.1.2 User Story 1 – Establish Connection to Cooler

As an app user, I want to connect my cooler to my mobile device in a simple manner, so that I can use the app with my cooler.

Acceptance Criteria

- A first run wizard guides the user through the connection process when the app is started for the first time
- It is possible to unlink the current cooler and connect with a new one (e.g. the old cooler is damaged or lost)

2.2.1.3 User Story 2 – System Status

As an app user, I want to see the current status of the cooler at a glance (temperature, remaining battery charge and time) so that I can ensure that the cooler is functioning properly and how long I can expect it to continue working.

Acceptance Criteria

- The current temperature inside the cooler can be seen on the app's home screen
- The current ambient temperature can be seen on the app's home screen
- The remaining battery charge can be seen on the app's home screen
- The remaining battery time can be seen on the app's home screen
- If a problem occurs, it is displayed on the app's home screen

2.2.1.4 User Story 3 – Continuous Data Retrieval from Cooler

As an app user, I want the app to automatically receive live data from the cooler via Bluetooth when the app is opened so that my information is always up to date.

Acceptance Criteria

- When connected, the app displays the current live data from the cooler, as defined in "User Story 2 – System Status"

2.2.1.5 User Story 4 – Send Collected Cooling Data to the App

As an app user, I want to see the data the cooler has collected while the app was not running in the foreground, so that I can view all cooler data in the app.

Acceptance Criteria

- When the app is reopened, data collected while it was not running in the foreground must be synchronised and displayed
- If the connection between the app and the cooler is lost and then re-established, data collected during the interruption must be synchronised and displayed in the app

2.2.1.6 User Story 5 – Status Notification

As an app user, I want the app to continue synchronisation with the cooler when it is running in the background, so that I can get push notifications about the current status of the cooler.

Acceptance Criteria

- The app will continue synchronising data with the cooler while it is connected and running in the background

- The app will send a push notification if the temperature exceeds a defined limit while the app is running in the background
- The app sends a push notification if the battery level falls below a defined limit while the app is running in the background

2.2.1.7 User Story 6 – Temperature and Battery History

As an app user, I want to have the ability to view the temperature and battery history over different periods of time (including average, min. / max.) so that I can assess whether my medication has been stored correctly throughout and how my battery is being used.

Acceptance Criteria

- The app shows the inside temperature history over a specified time period
- The app shows the ambient temperature history over a specified time period
- The app shows the battery history over a specified time period
- The app shows the average temperature over a specified time period
- The app shows the minimum and maximum temperature over a specified time period

2.2.1.8 User Story 7 – Trips

As an app user, I want to manually record a certain time frame, so that I can define important durations in which I use the cooler heavily (e.g. during a week-long vacation) to keep track of the cooling status (see “[User Story 2 – System Status](#)”) more easily.

Acceptance Criteria

- The app lets the user start and stop a trip
- The trip can be given a custom name
- Data from past trips can be viewed in detail
- Data from the active trip can be viewed in detail

2.2.1.9 User Story 8 – Set Temperature Range

As an app user, I want to have the option to adjust the target temperature (e.g. 2-6°C instead of 2-8°C), so that I can comply with the storage requirements for my medication.

Acceptance Criteria

- The cooler’s temperature range can be set using the app
- The app displays the set temperature range from the cooler after connecting

2.2.1.10 User Story 9 – Rapid Cooling

As an app user, I want to have the option to initiate rapid cooling prior to embarking on a trip so that the cooler reaches the target temperature as quickly as possible while it is still connected to the power supply. I also want the app to provide clear information on the increased energy consumption of this mode in comparison to the standard, optimised slow cooling mode, so that I can make informed decisions as to when to use this mode.

Acceptance Criteria

- Rapid cooling can be initiated from within the app
- Information about this mode is provided when it is enabled

2.2.1.11 User Story 10 – Configure Alarms

As an app user, I want to set custom alarms that trigger push notifications and notices in the app based on the battery percentage or cooling time remaining, so that I have enough time to react accordingly.

Acceptance Criteria

- The user is able to set a custom value for the alarm in addition to the pre-set alarms

2.2.1.12 User Story 11 – Device-specific Data Retrieval

As an admin user, I want to be able to retrieve the status data from a specific cooler via its serial number and export it for a specific period of time in order to provide support and perform diagnostics.

Acceptance Criteria

- Data from a specific cooler can be exported from the backend
- Data can be exported for a specific time period

2.2.1.13 User Story 12 – Aggregated Sensor Data

As an admin user, I want to retrieve values like average, minimum and maximum across all coolers per sensor, so that I can identify trends and develop models for predictive maintenance.

Acceptance Criteria

- Data from all coolers can be exported from the backend in a single request
- The user can specify what properties they would like to export

2.2.1.14 User Stories About Possible Extensions

This section defines the user stories for potential extension features. There is a possibility that these will be implemented within the Bachelor's thesis, should either all MVP user stories be implemented or the team consider a user story to be low effort. However, their primary function is to provide insights into the features that the industry partner plans to implement following the conclusion of the thesis. As these user stories fall outside the scope of the MVP, they are listed without acceptance criteria.

User Story 13 – Language Switcher

As an app user, I want to be able to switch the app language between German and English, so that the language aligns with my preferences.

User Story 14 – Temperature Unit Switcher

As an app user, I want to be able to switch the temperature unit between °C and °F, so that the temperature unit aligns with my preferences.

User Story 15 – Information on Cooler Display

As an app user, I want to be able to show personal contact information on the cooler's E-Ink display, so that the device can be identified if lost.

User Story 16 – Opt Out of Usage Data Tracking

As an app user, I want to be able to opt out of anonymous data tracking by the startup, so that my data remains private.

User Story 17 – Silent Mode

As an app user, I want to be able to set the cooler to silent mode (fan off or no alarm) so that I will not be disturbed.

User Story 18 – User Feedback

As an app user, I want to be able to provide feedback via the app, so that I can share my experience of using the product with the manufacturer.

User Story 19 – Account Management

As an app user, I want to create, sign in to and manage my own account, so that I can use it to store my cooler data and access it from another device.

User Story 20 – User Data Export

As an app user, I want to export my temperature history in a human- and/or machine-readable format (e.g. PDF, CSV), so that I can share it with my health care provider or keep it for my own records.

User Story 21 – Dark Mode

As an app user, I want to switch the app between light and dark mode, so that I can use the app based on my preferences and conditions (e.g. to check a temperature alert at 3 AM without being blinded by a bright, white screen).

User Story 22 – Notification Permission Handling

As an app user, I want to be clearly informed when push notifications are disabled on my device and which notifications are affected, so that I understand that I will not receive temperature or battery alerts and can re-enable them as required.

User Story 23 – GDPR/nFADP Compliance

As an app user, I want to view the privacy policy, request an export of all my data and permanently delete all data associated with me, so that my rights under the GDPR and nFADP are respected.

User Story 24 – Connect App with Multiple Coolers

As an app user, I want to be able to connect multiple coolers to my app, so that I can use multiple coolers if I do not have enough space for my medication in one cooler.

User Story 25 – Connect Cooler with Multiple Apps

As an app user, I want to be able to connect multiple mobile devices to one cooler, so that I can lend the cooler to someone else, or enable someone to check the status of my cooler on their mobile device as well.

2.2.1.15 Group User Stories into Epics

The user stories listed above have been prioritized and grouped into Epics, as seen in [Table 2](#). The earlier Epics are of greater significance to the project than the later ones.

Epic	User Stories
App FRs	
Epic A.1: Cooler status	– User Story 1 – Establish Connection to Cooler – User Story 2 – System Status – User Story 3 – Continuous Data Retrieval from Cooler – User Story 4 – Send Collected Cooling Data to the App – User Story 5 – Status Notification
Epic A.2: Display historical data	– User Story 6 – Temperature and Battery History – User Story 7 – Trips
Epic A.3: Control cooler via app	– User Story 8 – Set Temperature Range – User Story 9 – Rapid Cooling – User Story 10 – Configure Alarms
Epic A.4: Data synchronisation	The user stories are first implemented without direct communication with the cooler due to the protocol not being ready after M2 (see Section A.1.1.1). Instead, dummy data is used. The communication itself will be addressed in this Epic.
Epic A.5: Personalize app (Extension features/Optional user stories)	– User Story 13 – Language Switcher – User Story 14 – Temperature Unit Switcher – User Story 15 – Information on Cooler Display – User Story 16 – Opt Out of Usage Data Tracking – User Story 17 – Silent Mode – User Story 18 – User Feedback – User Story 19 – Account Management – User Story 20 – User Data Export – User Story 21 – Dark Mode – User Story 22 – Notification Permission Handling – User Story 23 – GDPR/nFADP Compliance – User Story 24 – Connect App with Multiple Coolers – User Story 25 – Connect Cooler with Multiple Apps
Backend FRs	
Epic B.1: Telemetry collection from app	– User Story 11 – Device-specific Data Retrieval
Epic B.2: Aggregate and evaluate telemetry	– User Story 12 – Aggregated Sensor Data

Table 2: List of user stories mapped to Epics

[Section A.1.1](#) shows when the implementation of each Epic is scheduled.

2.2.2 Non-Functional Requirements

The non-functional requirements (NFRs) define the quality characteristics of the product. They influence how the features specified in the functional requirements are implemented.

In the context of this project, the approach adopted was informed by the quality characteristics delineated in ISO/IEC 25010:2023 [8], [9]. The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL

NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this section are to be interpreted as described in RFC 2119 [10].

In total, 6 NFRs were established, which are listed in the tables below.

ID	1
Topic	The test coverage must be at least 80%.
Characteristic	Maintainability – Testability
Description	The CI/CD pipeline of both the app and backend repository should run all tests on every push and create a code coverage report. Every merge request should show the current code coverage of the branch. If the test coverage falls by more than 3%, the branch must not be merged.
Acceptance Criteria	The test coverage must be at least 80% at the end of the project.
Priority	medium
Creation Date	27.02.2026
Due Date	M3 – CW23 M4 – CW25

Table 3: Description of NFR 1 "Maintainability – Testability"

ID	2
Topic	The app must be easily navigable by members of the target audience with varying degrees of technical proficiency.
Characteristic	Interaction Capability – Learnability
Description	The project should involve the conduction of two usability tests. The selection of test subjects is handled by the industry partner, consisting of members of the target audience. The composition of the audience should include technically adept and technologically indifferent users. The aim is to provide an user interface that must be perceived by both groups as intuitive and functional.
Acceptance Criteria	– The average SUS score of the final UX test must be at least 68 (above average) [11]. – Each UX test must be conducted with at least five participants.
Priority	high
Creation Date	27.02.2026
Due Date	M3 – CW23

Table 4: Description of NFR 2 "Interaction Capability – Learnability"

ID	3
Topic	The integrity of the data from the medication cooler must be ensured.
Characteristic	Functional Suitability – Functional Correctness
Description	The medication cooler app provides users with data that informs their health decisions, which can have a significant impact on their lives. It is thus vital that the data is sensible. Therefore all data received from the cooler must be validated for plausibility.
Acceptance Criteria	All public API functions of the module responsible for parsing data received via BLE must only allow data that has been validated to be passed to other modules within the application.
Priority	high
Creation Date	27.02.2026
Due Date	M3 – CW23

Table 5: Description of NFR 3 “Functional Suitability – Functional Correctness”

ID	4
Topic	All data intended for transmission must be eventually synchronised between the app, medication cooler and backend.
Characteristic	Reliability
Description	The medication cooler synchronises its data with the app over BLE. In the event of disconnection, the app must be able to receive the missing data from the device upon re-establishment of the connection. During the disconnection period, the app must not buffer or send any control commands for the device. Upon reconnection, the app shall resume normal function. The synchronisation between app and the backend which happens over the internet must also be eventually consistent.
Acceptance Criteria	– The cooler will be disconnected from the app for one hour while in cooling mode. Upon reconnection, the app must display the data collected during that hour as if the cooler was connected for the duration of the disconnection. This process must not take more than 10 seconds. – Upon reconnection with the internet, all data expected by the backend must be sent to it within 1 minute.
Priority	high
Creation Date	27.02.2026
Due Date	M3 – CW23

Table 6: Description of NFR 4 “Reliability”

ID	5
Topic	The app must notify the user if the desired temperature cannot be kept.
Characteristic	Safety – Hazard Warning
Description	If the medication cooler is unable to keep the temperature of the cooling chamber within the range specified by the user (e.g. battery empty, exceeding safe operating temperatures, lid not properly closed) the app must issue a clear warning to the user if connected. If the app is not currently connected, the notification must show as soon the connection is established again.
Acceptance Criteria	– The app must return a warning when the cooling chamber temperature is outside the specified range for 30 seconds. – The app must return a warning if the battery level drops below 20%, an urgent warning if it drops below 10% and a critical warning if it drops below 5%. – If the app is not connected when a problem occurs, the warning must be displayed at the latest five seconds after successful synchronisation.
Priority	high
Creation Date	27.02.2026
Due Date	M3 – CW23

Table 7: Description of NFR 5 “Safety – Hazard Warning”

ID	6
Topic	The code passes the rules set by the linters and formatters.
Characteristic	Maintainability - Analysability
Description	A linter and formatter must be configured for both the backend and the app to analyse the code for faults and ensure consistent formatting. These tools should be configured to work within the used IDEs. The CI/CD pipeline must run both tools before merging a branch and should run them on every push. If any problems are detected, the pipeline must fail. When available, the linter for the language should be configured to operate in strict mode to detect more issues than with the default settings. An example of a stricter set would be <code>Rusts clippy :: pedantic</code>. A formatter for Typst may be configured accordingly.
Acceptance Criteria	– Linter and formatters for Dart and Rust are set up within their IDEs – Linter and formatters for Dart and Rust are run in the pipeline on each push – Linter and formatters for Dart and Rust are run in the pipeline on each Merge Request (MR) – Linter are configured with a strict ruleset – Prohibit merging a MR if a linter or formatter fails – A formatter is set up for Typst code (optional)
Priority	medium
Creation Date	29.03.2026
Due Date	M3 – CW23 M4 – CW25

Table 8: Description of NFR 6 “Maintainability - Analysability”

3 Design and Architecture

3.1 System Context

This chapter outlines the context in which the system is developed. It describes the main software components, their interaction with external systems, and the environment and constraints that influence the design and implementation of the solution.

3.1.1 Cooler

The industry partner is developing a portable medication cooler for temperature-sensitive medication. This device is responsible for maintaining the required storage conditions and for measuring relevant data, such as temperature, humidity, battery state, and device status.

In the context of this Bachelor's thesis, the cooler is primarily relevant as an external device which provides data to and receives configuration changes from the mobile application. The internal cooling logic and embedded firmware are not the focus of this project. Nevertheless, the app must still be developed according to the logic of the cooler.

The communication infrastructure between the cooler and the mobile application is based on Bluetooth Low Energy (BLE). As the physical cooler and firmware are still in development, the app is developed and tested using a simulator, consisting of an Arduino with BLE hardware. This simulator replicates the relevant BLE services and characteristics, thereby enabling the app to be developed before the final hardware becomes available.

3.1.1.1 Bluetooth Low Energy Protocol

Bluetooth Low Energy is structured around characteristics and services. A characteristic consists of a value transmitted over Bluetooth, along with its metadata stored on the device offering the characteristic. For this project, the relevant metadata are the address UUID, permissions and the Client Characteristic configuration descriptor (CCCD) [12]. The address UUID is used by the client to access a particular characteristic. Once accessed, the client can perform so-called GATT (Generic Attribute Profile) operations on the characteristics to access its values. There are two types of GATT operations: Manual access and subscriptions. Each characteristic has permissions that determine whether a client can manually access it with read-only, write-only, read-write or none. If none is set, the client can not manually poll this characteristic; this is usually used to make the characteristic accessible only via a subscription. The CCCD defines whether and how a subscription can be created. There are two available modes: `notify` and `indicate`. The former sends an acknowledgement to the sender when data has been received, while the latter does not [13].

Each characteristic belongs to a service, which represents a grouping of related characteristics. The Bluetooth standard offers a range of pre-defined services. Vendors can also create custom characteristics and services, as the industry partner has done. [Table 9](#) shows the services offered by the cooler and used by the app, along with example characteristics for each service.

Service	Description	Example Characteristics
Generic Access [14]	Pre-defined, offers basic information about the device	Device Name, Appearance (e.g. headphones, computer, speaker)
Device Information [15]	Pre-defined, offers more detailed device information	Manufacturer Name, Serial Number, Firmware Revision
Battery [16]	Pre-defined, contains details about the device's battery	Battery Level, Battery Time Status, Battery Level Status (Charging/Discharging)
Log Data Sync	Custom, synchronises historical data with the cooler	Historical Log, Last Counter, Sync Control Point
Live Data	Custom, sends live sensor data to the app	Inside Temperature, Ambient Temperature, Humidity
Settings	Custom, reads and writes cooler settings	Minimum Cooling Temperature, Maximum Cooler Temperature

Table 9: Overview of the pre-defined and custom BLE services offered by the cooler

The characteristics in the Generic Access and Device Info services are static and thus read-only. For more dynamic values, such as those in the Live Data or Battery service, a notify subscription in addition to manual polling is available. This allows the cooler to send updates as soon as they occur. Configuration values, such as temperature limits, are written by the app to the cooler. For the historical synchronisation, the industry partner has implemented a command-like structure: In order to initiate the synchronisation process, a particular value has to be written to the Sync Control Point characteristic, after which the data is sent over the Historical Log characteristic. Once the synchronisation process has concluded, the cooler sends a completion value to the app via an indicate subscription.

The BLE protocol is developed in parallel with the mobile application. Consequently, the interface between the cooler and the app is not yet finalised; changes to the protocol may necessitate corresponding adjustments to the mobile application. Likewise, insights gained during app development may reveal that certain functionality would be better presented in a different manner via the BLE interface of the cooler. In such cases, it is possible to adapt both the protocol and the firmware.

For testing purposes, the industry partner has developed a simulator of the cooler's functionality on an Arduino. The team has been granted access to the simulator and its source code to develop the app. While ultimately, the industry partner is responsible for the BLE protocol, this has nevertheless allowed the team to modify the protocol themselves for prototyping purposes.

Where possible, the software architecture of the app should ensure that BLE-specific code is isolated, so that protocol changes can be integrated without affecting other parts of the application.

3.1.2 App

The mobile application is the primary user-facing component of the system developed for this Bachelor's thesis. While the cooler is responsible for maintaining the storage conditions, the app provides transparency around this process. It enables users to view historical data, monitor the cooler's current state and adjust settings.

A key focus of the app is usability. The measured data and the device status must be presented in a manner that is easy to comprehend, since users with different levels of expertise should be able to quickly assess whether the cooler is and was operating as expected.

Reliability is another central concern. The app must be capable of managing temporary disconnections, missing data and incomplete synchronisation without presenting users with misleading information.

3.1.3 Backend

The backend extends the system beyond a single cooler and mobile device. Its purpose is to provide the necessary server-side infrastructure for collecting and aggregating usage data from coolers. This data can be used by the industry partner for product improvement, predictive maintenance, and optimal control modelling.

Unlike the mobile application, the backend is not intended for direct interaction by end users. Therefore, its focus is not on usability, but on structured access to collected information.

Data from coolers, which is sent over a web API, may arrive at the backend irregularly: Users may travel without internet access, the app may not always be connected to the cooler, and synchronisation with the backend may only happen after a delay. Therefore, the backend should not rely on continuous data transmission.

The backend data model should also remain extensible. Since the BLE protocol and firmware are still evolving, future versions may introduce additional telemetry values or change existing data structures. The backend should therefore avoid making assumptions about a fixed data format and should support future changes without requiring a complete redesign.

3.1.4 Development Environment

The mobile application is expected to run on both iOS and Android devices, as the industry partner has requested a cross-platform solution. They have therefore specified that the app should be developed using the Dart programming language and the Flutter framework. The industry partner will continue development of both the app and backend after the conclusion of this Bachelor's thesis.

As the cooler communicates with the app via a Bluetooth Low Energy (BLE) protocol, the app must accommodate the typical limitations of wireless device communication. These include connection interruptions, limited data transfer rates, restrictions on background execution imposed by the operating system, and differences in behaviour between iOS and Android.

Many of these platform-specific differences are abstracted by Flutter and the library used for BLE communication. This enables the majority of the application logic to be implemented independently of the underlying operating system.

3.1.5 Development and Domain Constraints

The project is subject to several constraints that affect development and testing. One important constraint is that the final cooler hardware and firmware are not yet available. For this reason, development relies on a BLE simulator. While the simulator enables implementation and testing of the app, it cannot fully replace testing with the final hardware. Behaviour such as signal strength or connection stability must be validated once the real hardware is available.

The main development platform is Android, using the Android Studio IDE [17]. While Android Studio provides support for running apps in an Android emulator, this has proven to be of little use as the emulator does not support Bluetooth functionality [18]. Consequently, all development and testing had to be carried out on the team's personal Android phones.

Development and testing of the iOS app have been challenging, as iOS app compilation requires the macOS-exclusive Xcode [19]. Since the team does not have permanent access to an iOS or macOS device, they have been unable to test the iOS app for most of the Bachelor's thesis. There is also no CI/CD build job for iOS, as described in [Section 5.3.2.2](#). Since BLE behaviour can differ between Android and iOS, it is difficult to test all platform-specific cases during development. Therefore, iOS testing must be planned deliberately, especially with regard to BLE permissions and background behaviour. To this end, an iPhone XR has been loaned to the team by the industry partner in sprint 6. Together with temporary access to a OST-owned macOS device, the iOS version could be verified.

The system is developed in a medical context, because incorrect storage of medication can have serious consequences for users. However, the industry partner handles regulatory and safety-related responsibilities. For this thesis, the most relevant consequence is that implementation decisions, and known limitations should be documented clearly. This supports future development and enables the industry partner to evaluate the software in the context of the compliance requirements.

3.2 Domain Model

The domain model describes the problem domain, including its conceptual classes and the interactions between them [20].

In this problem domain there are two types of actors: *App Users* and *Admin Users*, as described in [Section 2.2.1.1](#). The former interact with the *App* and the *Cooler*, whereas the latter only interact with the *Backend* to export *Reports* from the *Measurements* collected from the *Coolers*.

App Users can access *Measurements* done by the *Sensors* in the *Cooler* via the *App*. The *App* is also able to adjust *Settings*, split into *App Settings* and *Cooler Settings*. The former consists of configuration only relevant within the *App* such as adding *Trips*. The latter are for *Settings* stored directly on the *Cooler*, like the temperature range and alarms. This is possible because the *Cooler* is synchronised with the *App* via BLE. The *Cooler* also sends information like its serial number and its current status to the *App* via BLE.

The domain model in [Figure 5](#) illustrates the concepts described above and demonstrates the relationships between the entities.

As defined in “[User Story 24 – Connect App with Multiple Coolers](#)” and “[User Story 25 – Connect Cooler with Multiple Apps](#)”, it should be possible in the future for one app user to use multiple coolers or for one cooler to connect to multiple apps. This concept is shown in the domain model. However, for this Bachelor’s thesis, the stakeholders agreed to keep a 1:1 relationship between app and cooler.

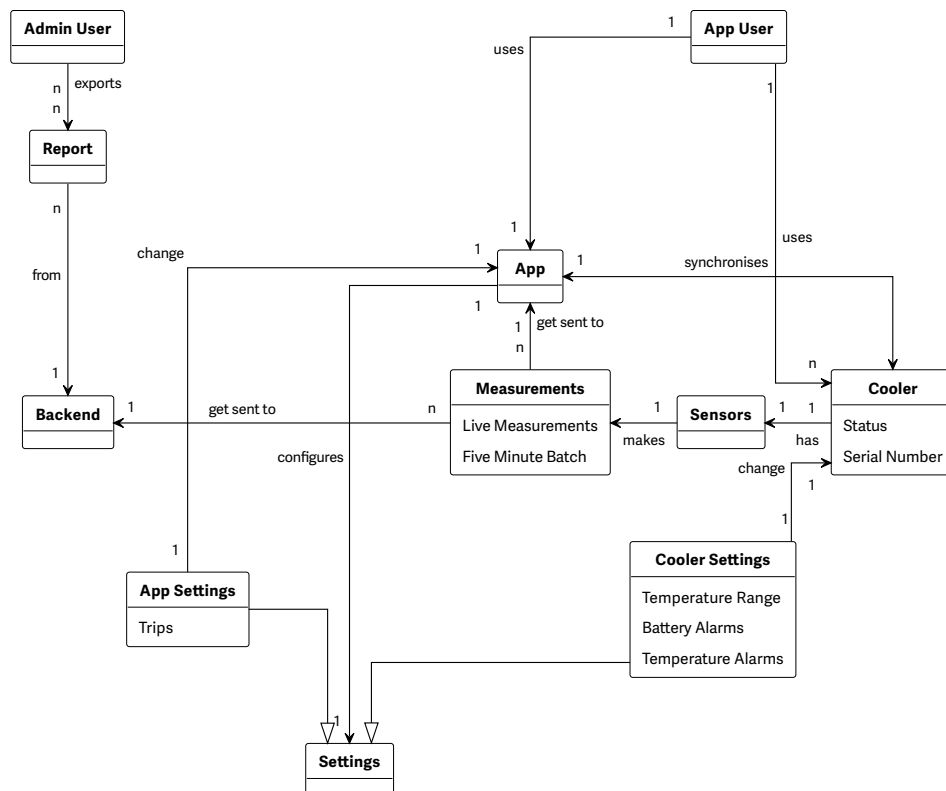


Figure 5: Domain model of the problem domain

3.3 C4 Model

The chosen architecture has been visualised using diagrams following the C4 model [21]. The subsequent figures illustrate the application's structure in different contexts at an increasing level of detail.

3.3.1 System Context

C4's system context provides a broad, non-technical overview of the software architecture by depicting "software systems" [22]. A software system is typically defined as something that can be maintained by a single software development team and delivers value to its users, i.e. it fulfils a user story [23]. Its purpose is to illustrate the systems' interactions with the different types of users, external systems and with each other. External systems represent software systems that directly communicate with the teams' systems, but are not under their control.

Figure 6 depicts the two actors of the application: The app user and the admin user, as outlined in the user stories in [Section 2.2.1](#). Whilst the application is being used by the app user on their mobile device through the app, the admin users are able to access the collected data in the backend via HTTP.

Only the app initiates communication between the internal systems: it sends telemetry data to the backend via HTTP. This communication is one-way, as the backend does not send any requests to the app. The app system is designed to interact with a single external system: The medication cooler from which the application receives data regarding the cooler's status, and to which it sends commands to control it. The backend system does not communicate with any external system.

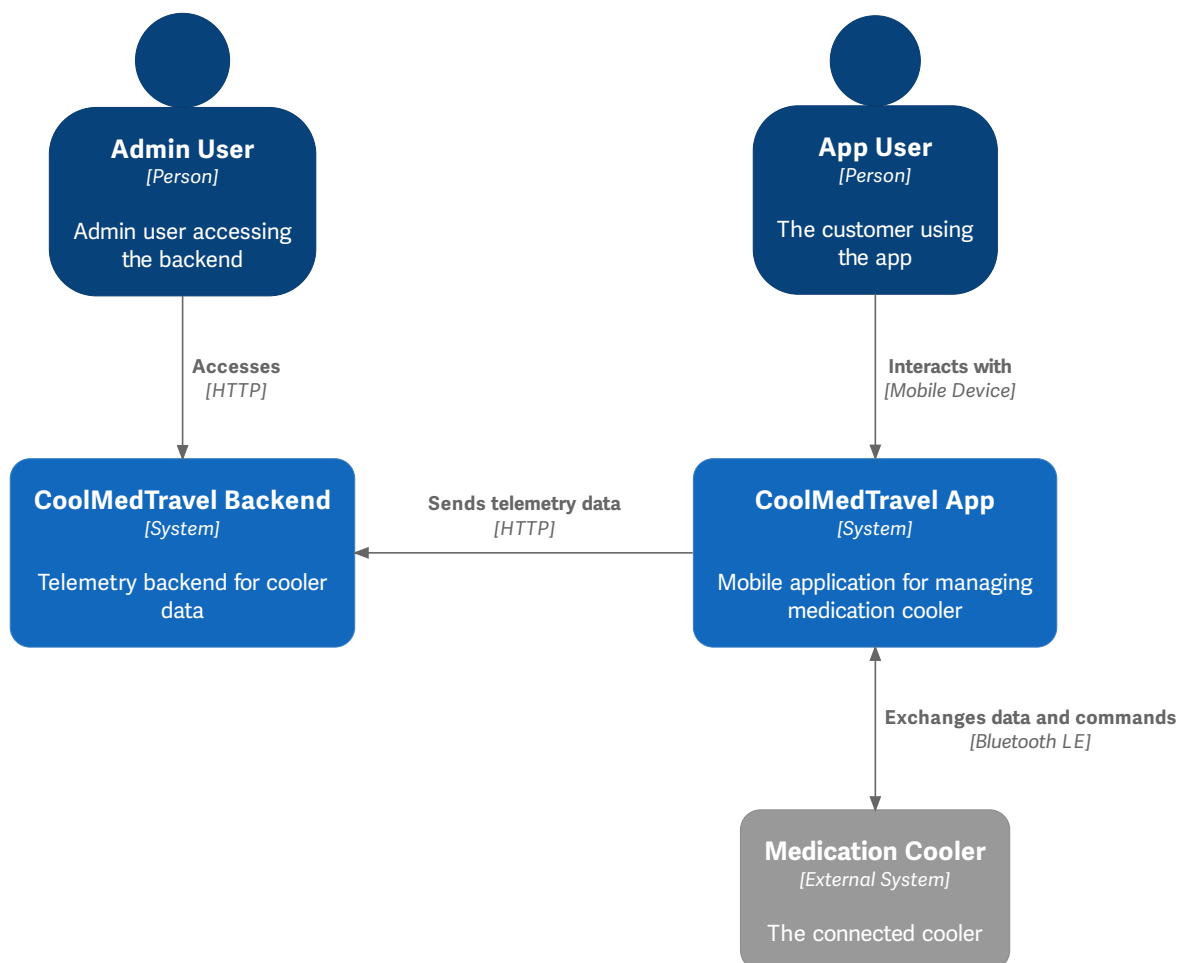


Figure 6: System Context Diagram showing the main systems and its interactions with external entities

3.3.2 Container Context

The second layer of C4 illustrates the high-level technology within a software system in the form of containers [24]. Each container serves as a representation of a distinct application or data store within the corresponding system, as seen in [Figure 7](#) [25]. The app and the backend both communicate with their own database. The interactions with external systems and actors are also more distinctly displayed; the diagram shows what component is responsible for communicating with those entities.

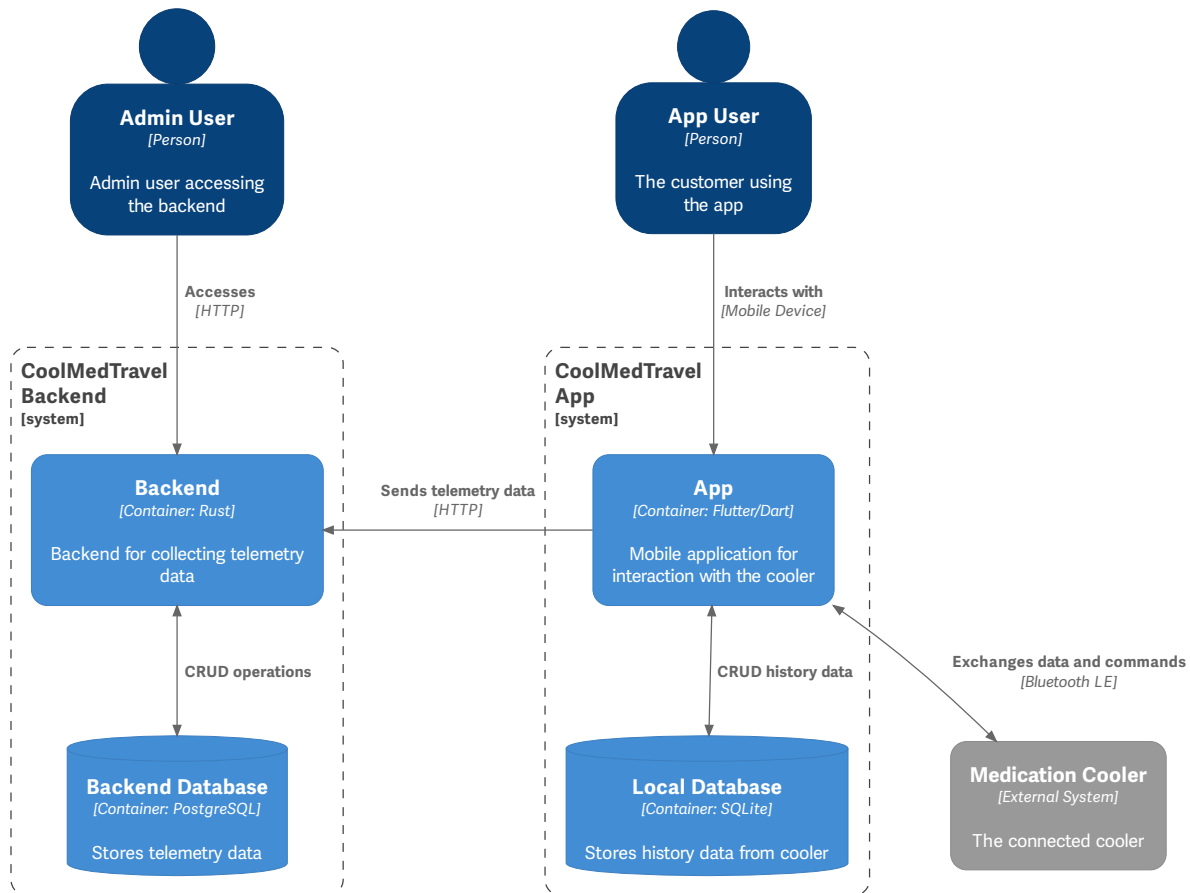


Figure 7: Container Context Diagram showing the different containers within the systems

3.3.3 Component Context

The third layer of the C4 model illustrates the individual components of a system. The team evaluated a component diagram for the app but decided that it would not add sufficient value. Due to the layered MVVM structure, as well as the number of pages, view models, repositories and services, the diagram would have been difficult to read. As the official C4 model website recommends creating component diagrams only when they add value [26], the team decided to use smaller MVVM-style diagrams instead. These are included in [Section 4](#), where they provide a clearer explanation of selected features.

The Gerust-based backend is more suitable for a component diagram, as it contains fewer components than the app and follows a clear crate-based structure. The diagram focuses on the components involved in normal operation. For readability, supporting parts such as configuration management and testing infrastructure are omitted. More information about Gerust and its components can be found in [Section 4.2](#).

The app sends its telemetry data to the backend as JSON over HTTP. The router processes the URL path and sends the request to the relevant controller. The controller then calls the create functions of the entities described in [Section 3.4.2](#), which validate the incoming data before inserting it into the database. An admin user can send an export request, which is processed by the export controller. This controller calls the read functions to collect data from all entities and returns the response as JSON.

[Figure 8](#) shows the component context diagram of the backend.

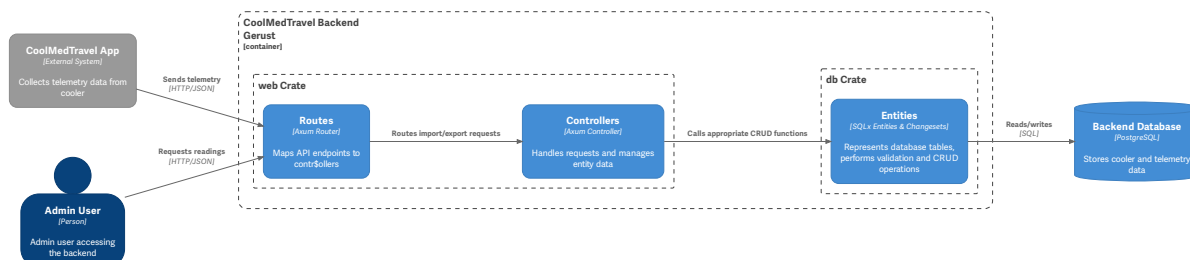


Figure 8: Component Context Diagram for the backend

3.4 Backend Database

As defined in “[User Story 11 – Device-specific Data Retrieval](#)” and “[User Story 12 – Aggregated Sensor Data](#)”, the backend database must be capable of retrieving status data for a specific cooler within a specified time frame, as well as extracting values such as average, minimum and maximum across all coolers per sensor.

These requirements affect both the choice of technology and the structure of the database schema.

3.4.1 Technology

PostgreSQL was selected as the database system because it is natively supported by Gerust, the project generator used for the backend [27]. More about Gerust can be found in [Section 4.2.1](#). This approach avoids additional integration work, ensuring the backend architecture remains aligned with the supported technology stack.

In addition, PostgreSQL is an open-source relational database system with a reputation for reliability, feature robustness and performance [28]. It also supports JSON values [29], which is relevant because the measured telemetry data is stored in this format, as is described further in [Section 3.4.2.1](#).

PostgreSQL can therefore be considered a valid choice for this application.

3.4.2 Database Schema

[Figure 9](#) shows the chosen database schema for the backend. It separates device information, telemetry properties and telemetry readings into different tables.

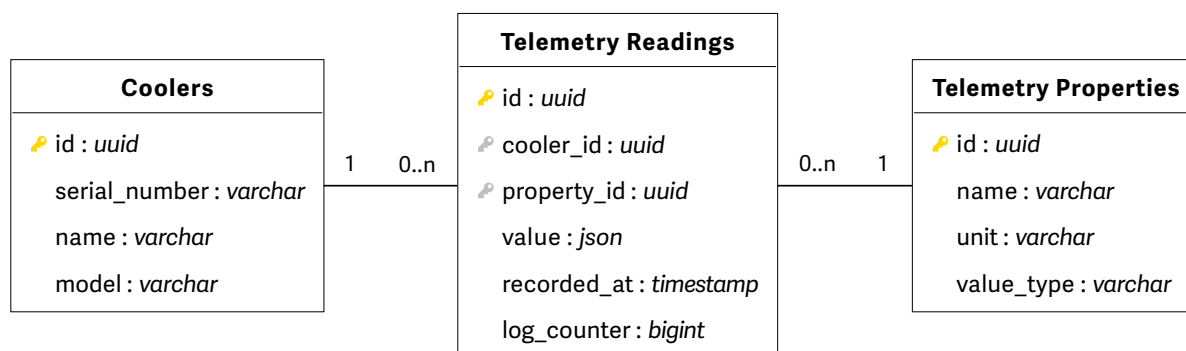


Figure 9: Backend database schema

The `coolers` table contains one row per physical cooler. It stores identifying information about them, such as the serial number, name and model. The serial number is used to retrieve data for a specific cooler. The use of the serial number as the primary key for the database table was discussed but rejected, as this would result in higher coupling with database internals.

The `telemetry_properties` table describes the available telemetry types. It contains data such as the name of the sensor, the measurement unit the values are stored in and the data type of the values. This table makes it possible to add new sensors to the cooler without changing the structure of the `telemetry_readings` table.

The `telemetry_readings` table stores the telemetry data. Each row references one cooler and one telemetry property. The `recorded_at` column identifies the timestamp of the reading, the `value` stores the payload as JSON and the `log_counter` is used for synchronisation with the app. More details about the synchronisation process are provided in [Section 4.1.5](#) and [Section 4.1.7](#).

3.4.2.1 Telemetry Reading Format

Historical telemetry data is transmitted as log entries. Each entry contains a log counter, a timestamp, a telemetry property, and a JSON-encoded value. Depending on the telemetry property, this value may be a scalar, a list or an object containing aggregated values, such as minimum, maximum and average.

The JSON value for telemetry properties containing aggregated temperature values has the structure shown in [Listing 1](#).

```
1 {  
2   "min": 2.1,  
3   "avg": 5.4,  
4   "max": 8.7,  
5 }
```

Listing 1: Example JSON value stored in the telemetry readings table

The decision to store the data in JSON has been made so that the type of data telemetries report is not limited by the column type set in PostgreSQL. For example, some sensor values are reported as integers while others are floating point values. To convert the data back into typed values, the data type is stored in the `value_type` column in the `properties` table.

The industry partner has requested that all data should be stored in a single JSON value, rather than creating a separate column for each value. This approach ensures that the type of data reported by a sensor is not limited by the pre-defined columns and their types.

3.4.3 Requirement Fulfilment

The following example queries show how the database model fulfils the backend-related user stories. The queries are intended as documentation examples and use named parameters such as `:serial_number`, `:start_time` or `:end_time`, which are provided by the backend.

3.4.3.1 Device-specific Data Retrieval

["User Story 11 – Device-specific Data Retrieval"](#) requires that telemetry data can be exported for a specific cooler and over a specific time period. This is achieved by joining the `telemetry_readings` with the `coolers` table and filtering by the cooler serial number and the requested time range.

The example query in [Listing 2](#) returns all telemetry readings for one specific cooler within the requested time range, together with metadata about the cooler and the telemetry type.

```
1 SELECT
2     c.id AS cooler_id,
3     c.serial_number,
4     c.name AS cooler_name,
5     c.model,
6     tp.id AS telemetry_property_id,
7     tp.name AS telemetry_name,
8     tp.unit,
9     tp.value_type,
10    tr.value,
11    tr.recorded_at,
12    tr.log_counter
13 FROM telemetry_readings tr
14 JOIN coolers c ON tr.cooler_id = c.id
15 JOIN telemetry_properties tp ON tr.property_id = tp.id
16 WHERE c.serial_number = :serial_number
17        AND tr.recorded_at ≥ :start_time
18        AND tr.recorded_at ≤ :end_time
19 ORDER BY tr.recorded_at ASC, tp.name ASC;
```

Listing 2: Example query for exporting telemetry readings of one cooler within a time range

3.4.3.2 Aggregated Sensor Data

“[User Story 12 – Aggregated Sensor Data](#)” requires aggregated values such as average, minimum and maximum across all coolers per telemetry. For the readings that already contain the minimum, maximum and average for a defined period of time, the query aggregates these precomputed values from the JSON in the value column.

For each telemetry that uses this value format, the example query in [Listing 3](#) returns the average, minimum and maximum value over all coolers within the requested time period.

```
1 SELECT
2     tp.id AS telemetry_property_id,
3     tp.name AS telemetry_name,
4     tp.unit,
5     tp.value_type,
6     AVG((tr.value ->> 'avg')::numeric) AS average_value,
7     MIN((tr.value ->> 'min')::numeric) AS minimum_value,
8     MAX((tr.value ->> 'max')::numeric) AS maximum_value,
9     COUNT(*) AS aggregation_period_count,
10    MIN(tr.recorded_at) AS first_recorded_at,
11    MAX(tr.recorded_at) AS last_recorded_at
12 FROM telemetry_readings tr
13 JOIN telemetry_properties tp
14     ON tr.property_id = tp.id
15 WHERE tr.recorded_at ≥ :start_time
16        AND tr.recorded_at < :end_time
17        AND tp.value_type = 'json'
18 GROUP BY
19     tp.id,
20     tp.name,
21     tp.unit,
22     tp.value_type
23 ORDER BY tp.name ASC;
```

Listing 3: Example query for exporting aggregated telemetry data across all coolers within a time range

3.4.4 Considered Alternatives

An alternative schema was considered in which each row represents one timestamp and contained one column for each telemetry property. This would simplify the process of reading exports, as all values for a timestamp would be stored in a single row.

However, this design was rejected because it lacks flexibility. Adding a new telemetry property would require a schema migration and a new column. Furthermore, this approach would complicate the storage of telemetry-specific metadata, such as unit and value type. The chosen design separates metadata from readings, allowing the addition of telemetry properties without altering the table structure.

3.4.5 Limitations of the Current Schema

While the chosen schema fulfils the required user stories, it also has some disadvantages. For instance, the extraction and casting of values from the JSON payload increases complexity. Nevertheless, the enhanced flexibility in the value types compensates for the drawbacks.

Another disadvantage is that the schema stores one row for each cooler, telemetry property and timestamp. While this normalised structure makes it easy to add new telemetry properties, it also increases the number of rows in the telemetry readings table as the number of coolers (and possibly sensors) grows. Large exports or aggregations over long time periods may become expensive without additional optimisations. A possible solution to this is described in [Section 3.4.6](#).

3.4.6 Scalability

As the database scales, running queries to aggregate data such as minimum and maximum values across the telemetry readings table will no longer be feasible. In this case, the database can be upgraded with TimescaleDB, an open-source PostgreSQL extension designed for “high-performance real-time analytics on time-series and event data” [30]. This would allow the telemetry readings table to be converted into a “Hypertable”, which automatically generates tables with aggregated data for a specified time frame.

However, as a detailed evaluation of TimescaleDB is outside the scope of this Bachelor’s thesis, it is therefore considered as a potential future optimisation rather than as part of the proposed database design.

3.5 Local Database

As defined in “[User Story 4 – Send Collected Cooling Data to the App](#)” and “[User Story 6 – Temperature and Battery History](#)”, the local database must be capable of persistently storing historical cooler data. Data must be collected from the cooler both synchronously when the app is connected and retroactively after connection interruptions.

It is important to distinguish between live data and historical data. Live data is displayed on the app’s home screen and is sent by the cooler every second. These readings are only used to display the current temperature and are replaced as soon as a newer reading is received. They are not stored in the local database.

On the other hand, historical data is recorded and stored by the cooler in one-minute intervals. Depending on the telemetry, each entry will either show the value at that specific point in time, or the minimum, maximum and average values measured over the previous minute. These log entries are sent to the app either in real-time when the app is connected, or in a batch after the connection is interrupted. Only this historical data is stored in the local database.

3.5.1 Technology

As described in the Flutter documentation, Flutter provides three options for adding persistence to apps: key-value storage, file storage and SQLite [31]. For storing large amounts of structured data on a local device, Flutter recommends using a local SQLite database [32]. Therefore, SQLite is used as the persistence solution for the history data.

3.5.2 Database Schema

Since the app sends telemetry data from the local database to the backend, the local schema has a similar structure to the backend schema. Both schemas separate telemetry metadata from telemetry readings. However, the local database does not contain a `coolers` table, since the app only stores data for the currently connected cooler. Additionally, the local `telemetry_properties` table contains an extra key column that the app uses to query specific telemetry values independently of their display name. [Figure 10](#) displays the schema of the local database.

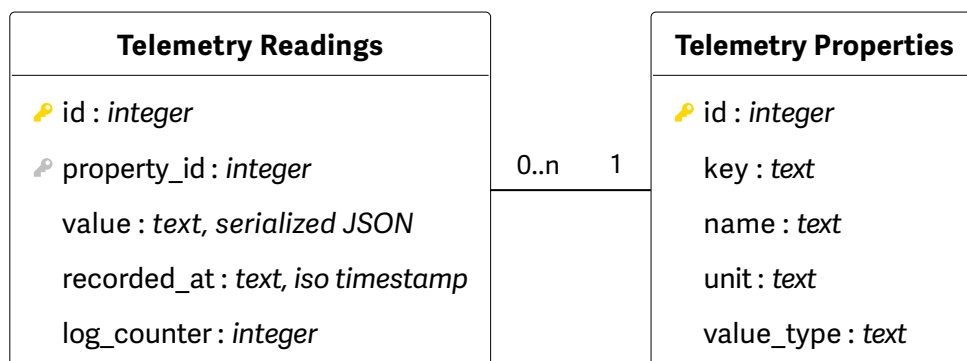


Figure 10: Local database schema

A detailed description of the tables can be found in [Section 3.4.2](#).

Maintaining the schemas of both databases in a similar manner facilitates straightforward data transmission to the backend.

3.5.3 Requirement Fulfilment

The database schema fulfils the requirements defined in “[User Story 4 – Send Collected Cooling Data to the App](#)” through the use of the `log_counter` field. The purpose of this user story is to ensure that no cooler data is lost when the app is not actively used or when the connection is interrupted. The database supports this by storing every received telemetry reading together with a `log_counter`.

If the connection between the app and the cooler is lost, the `log_counter` is used as the synchronisation point once the connection is re-established. All values captured at the same timestamp share the same `log_counter`, and new log entries are created every minute. When the app connects to the cooler, it retrieves the highest `log_counter` currently stored in its local database; if no rows exist yet, it uses 0. This value is then sent to the cooler together with a command to start synchronisation. In response, the cooler transmits each log entry with a `log_counter` greater than the one sent by the app. These entries are inserted into the local database and can then be displayed in the app.

The requirement defined in “[User Story 6 – Temperature and Battery History](#)” can be fulfilled because the database stores all telemetry readings alongside a timestamp, the `log_counter` and a reference to their telemetry property. This enables the app to query the internal database for specific values, such as inside temperature, ambient temperature, battery voltage, or remaining battery charge for a selected time period.

The average, minimum and maximum temperature values can also be determined for a selected time period. The cooler already sends aggregated temperature values for each minute. The app can then use these to calculate and display the required statistics, as shown in [Section 6.1.6](#).

All four acceptance criteria of the two user stories can therefore be fulfilled with the current database design.

3.5.4 Limitations of the Current Schema

Although the schema fulfils the required user stories, there is at least one limitation, which is that telemetry values are stored as `TEXT`. This approach keeps the schema flexible because it enables different value types to be stored in the same column. However, this means that casting is required before calculations can be performed.

Another consequence is that the database does not enforce whether a stored value matches the expected `value_type` defined in the telemetry property. This means that the application logic must ensure type correctness instead.

For the defined requirements, these limitations are acceptable. The schema is sufficient for synchronisation and displaying historical data over selected time periods.

3.6 Design Specifications

The design specifications were provided by the startup as an interactive HTML file, describing the UI functionality, layout and colour schemas, as illustrated in [Figure 11](#). This was used as a foundation for creating wireframes for the initial usability test described in [Section 6.2.2](#).

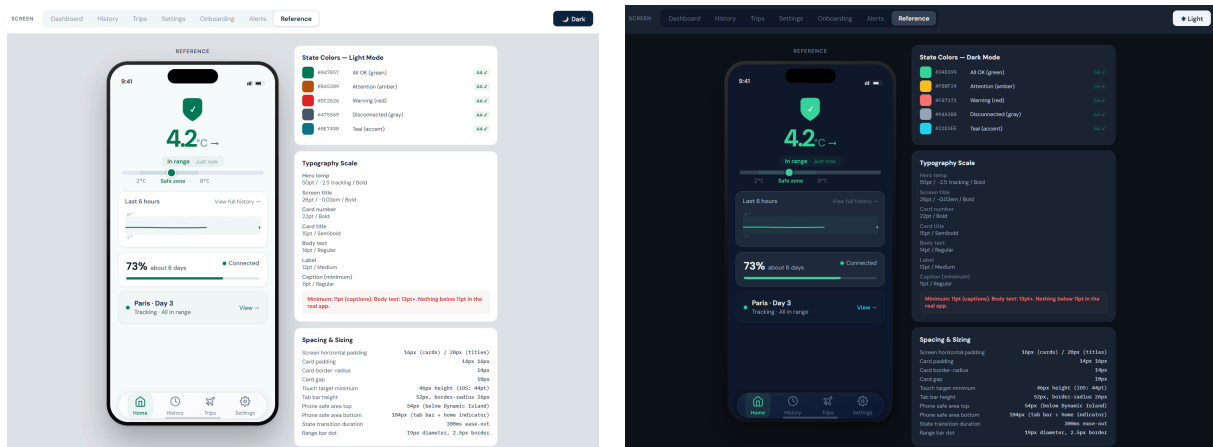


Figure 11: Screenshots of the design specification pages

3.6.1 Wireframes

The wireframes were created with Figma [33]. [Figure 12](#) shows the first group of wireframes, which depict the first run wizard. It appears when starting the app for the first time and guides the user through establishing the connection between app and cooler and adjusting the cooler settings. Since the usability test was conducted in German, the text in the wireframes is in German as well.

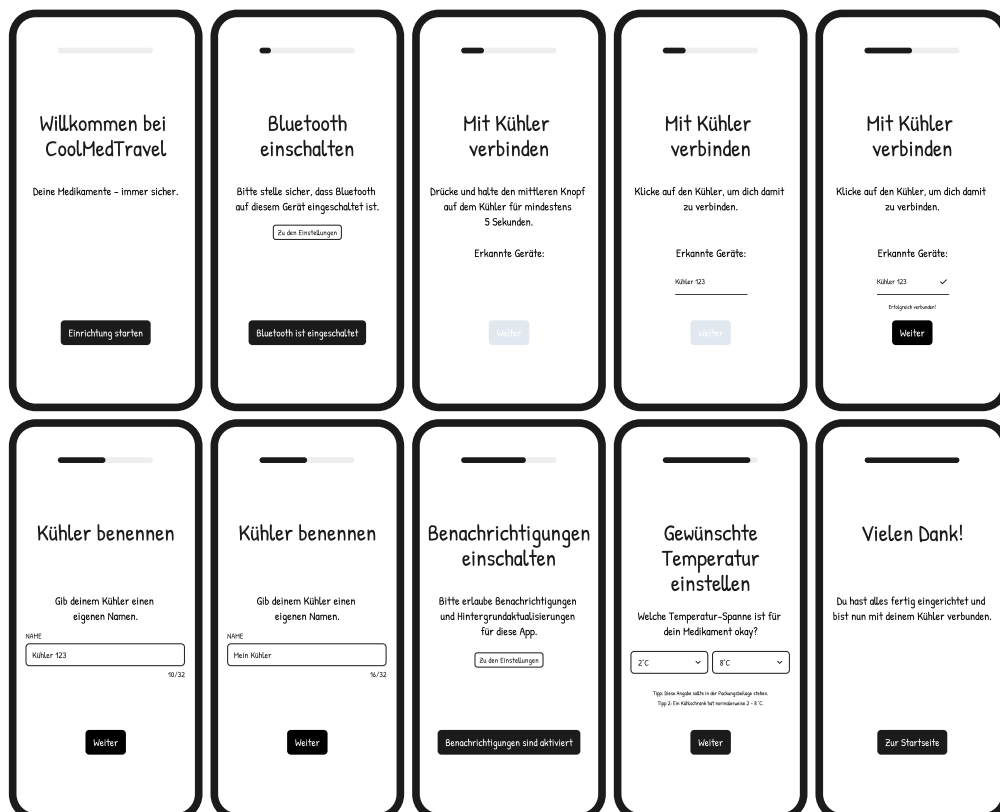


Figure 12: Wireframes of the first run wizard, guiding the user through setting up the app

The home screen is depicted in [Figure 13](#) across its possible states, which represent the status of the cooler. The first image shows the home screen immediately following the setup. As the appearance of the home screen differs between the “OK”, “Warning”, and “Attention” states, distinct colours have been used to highlight these variations, even though wireframes are normally presented without colour.



Figure 13: Home page with different statuses: Connected, OK, Warning, Attention, Not connected

To be able to ask questions about all the user stories defined in [Section 2.2.1](#), the trip, history and settings pages were also created as wireframes, as can be seen in [Figure 14](#).

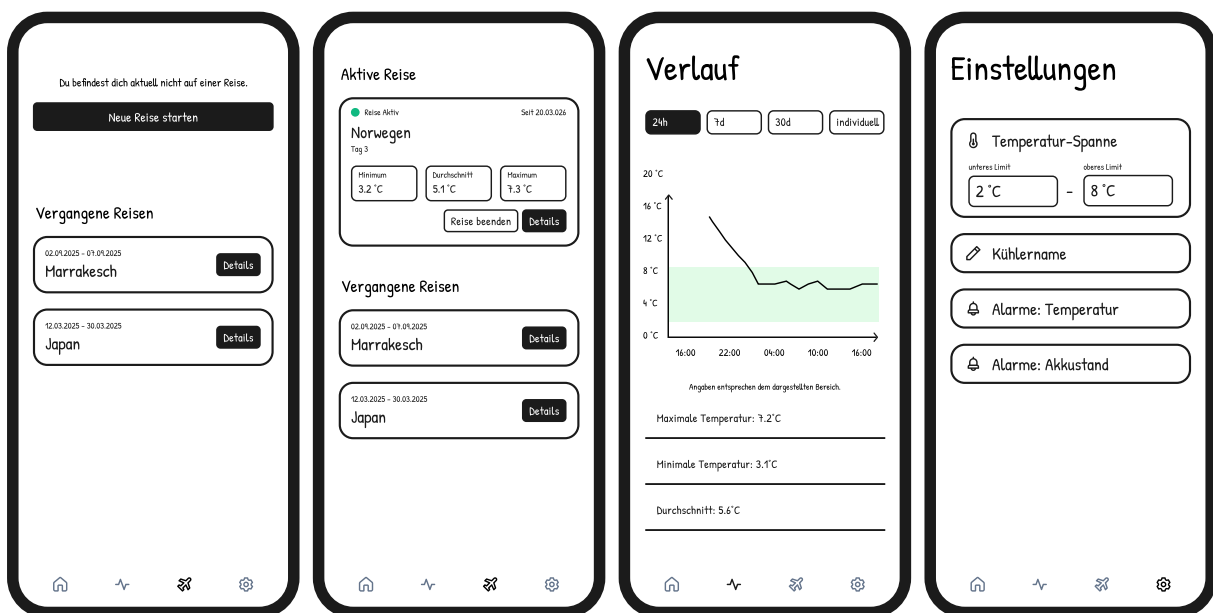


Figure 14: Trip page in different statuses, history page and settings

The wireframes were made into a clickable prototype for the usability test, which could then be sent to the participants as a link.

4 Implementation

4.1 App

This section details the implementation of the mobile app in the Flutter framework, which is build using the Dart programming language. The app's implementation follows the recommended Flutter MVVM (Model-View-ViewModel) architecture [34].

4.1.1 A Brief Introduction to Flutter MVVM

MVVM is designed to fulfil the “Separation of Concerns” principle by dividing the application code into an UI and a data layer. The UI layer itself is split into view and view model components, while the data layer consists of repositories and services.

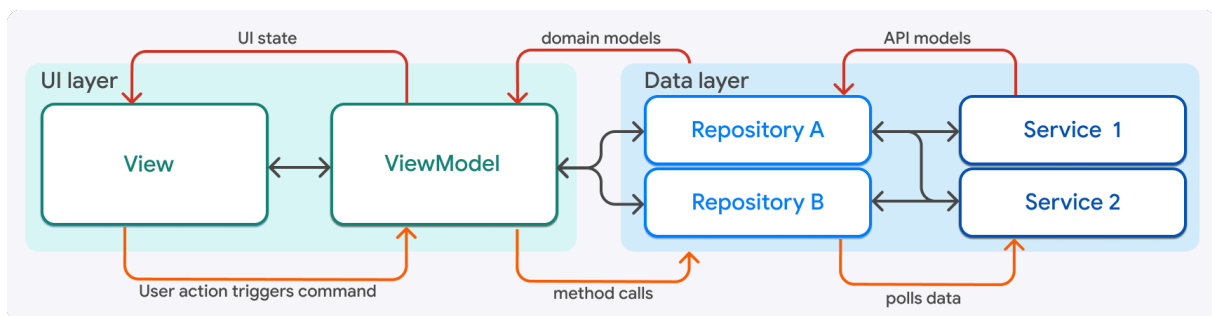


Figure 15: The MVVM architecture as described by Flutter. Source: [34]

Flutter centres itself around the concept of widgets. These are reusable components designed to be combined with each other [35]. A collection of widgets is bundled together to form a view. A view can represent a single page of an application or a component of a page such as a menu bar that appears on multiple pages.

Some widgets, such as buttons and text inputs, are designed for user interaction. Flutter provides pre-defined widgets for the most common scenarios, which contain properties where callbacks can be registered. For example, [Listing 4](#) shows an `ElevatedButton` with an `onPressed` property to which a function can be assigned.

```
1 Column(
2   children: [
3     Text('$_counter'),
4     ElevatedButton(
5       onPressed: () => setState(() { _counter++; }),
6       child: Text('Click me')
7     )
8   ]
9 )
```

Listing 4: Simplified widget example: A `Column` contains a `Text` and an `ElevatedButton`. When the button is pressed, the counter displayed in the `Text` widget increases. Note that the code for state management has been omitted.

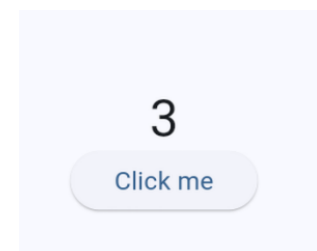


Figure 16: The widgets created in [Listing 4](#) after three clicks

In order to create a new widget, a class needs to inherit from either `StatelessWidget` or `StatefulWidget`. The difference between these two classes is whether or not the widget requires an internal state. The widget in [Listing 4](#) has the `_counter` variable, which stores the number of clicks on the button. This is called “ephemeral state”, as it does not need to be accessed from outside the widget. Ephemeral state can be managed using the `setState` function in a `StatefulWidget` [36]. Most widgets either do not require a state or depend on external data. In the latter case, their state should be managed by the view model. The view itself should only contain simple logic, such as `if`-statements, for handling basic display or animation logic.

Complex UI logic is located within the view model. A view model has three main responsibilities. Firstly, it manages UI state, such as whether certain widgets are disabled or hidden. Secondly, it retrieves app data from one or multiple repositories and formats it for display in the view. Thirdly, it exposes functionality to the view, which can then attach it to UI event handlers such as button presses [34]. While a view model can gather data from multiple repositories, it should belong to exactly one view [37].

Repositories are responsible for storing data in models and handling modifications to that data. They retrieve data from services and transform it for use by view models. They also perform business logic associated with services, such as state handling, caching and retry logic. Typically, repositories expose streams to view models, which can subscribe to those streams to receive updates when a model is updated.

Services wrap external APIs and expose their data to the repositories, such as operating system APIs or REST endpoints. They only act as a thin layer above these endpoints and do not hold state. One service can be used by multiple repositories, and one repository can use multiple services.

Flutter also recommends an optional Domain Layer, sometimes also called Logic Layer, for applications that frequently need to merge data from multiple repositories or reuse logic in different view models. This introduces a new type of class called “Use Case”, which sits between view models and repositories. In the cases mentioned above, the logic can be placed inside use cases for simple reuse [38].

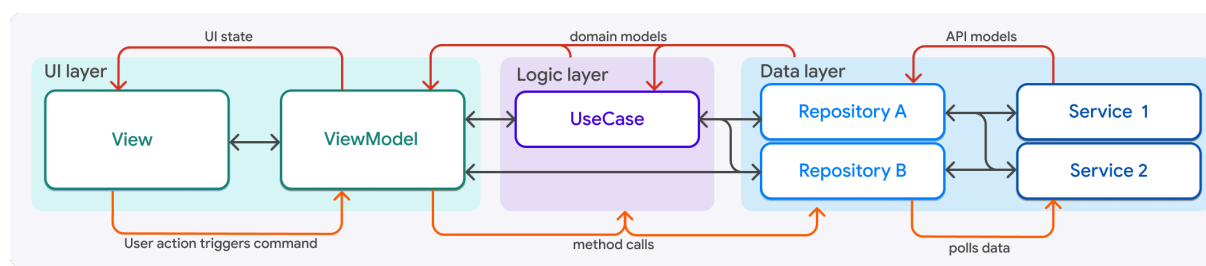


Figure 17: Flutter MVVM with optional Domain/Logic Layer. Source: [38]

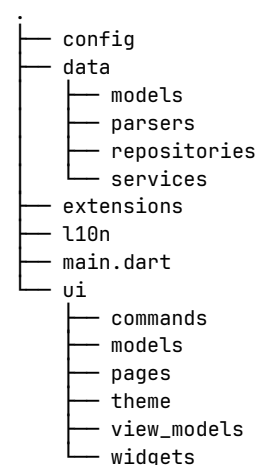
Although this approach offers better code reusability, the team has decided against implementing this optional layer due to the added complexity it would introduce to the architecture and to unit testing. In its current form, the application also does not fall into scenarios where a domain layer would be useful.

In general, the team’s architecture follows the recommended MVVM architecture, but it also diverges at several points. The concrete implementation is discussed in the following sections.

4.1.2 Implementation Overview

The main code of the app is located in the `lib` directory, while the tests are located in the `test` directory. The `lib` directory follows the “by type” package structure recommended by Flutter [39], which involves separating configuration, data handling, UI code, localisation, extensions and reusable widgets, as seen in [Listing 5](#). In addition to MVVM, the app also implements many of the architectural recommendations by the Flutter team, such as the use of `ChangeNotifier` for widget updates, the command pattern and dependency injection [40].

The application is initialised in `main.dart`. It uses the `Provider` library for dependency injection [41]. Services, repositories, commands and view models are registered in the app’s `BuildContext` with `MultiProvider`. Other components can then access the already registered components in their constructors by using `BuildContext.read<T>()`.



Listing 5: The directory structure used in the app

After all components have been registered, the root widget, `MaterialApp`, is created. It handles cross-platform initialisation, localisation and theming. The `PageRoot` widget then sets up the main page structure and displays the individual views of the application.

```
1 MultiProvider(  
2   providers: [  
3     Provider(create: (context) => BluetoothService()),  
4     Provider(  
5       create: (context) => ConnectionRepository(  
6         bluetoothService: context.read<BluetoothService>()  
7       ),  
8       dispose: (_, repository) => repository.dispose()  
9     ),  
10  ],  
11  child: const MaterialApp(home: PageRoot(), theme: /* ... localisation etc. ... */),  
12 )
```

Listing 6: Flutter dependency injection example: `ConnectionRepository` reads the already initialized `BluetoothService` from the `BuildContext`

Services such as `BluetoothService`, `AppDatabaseService` and `HttpService` wrap external APIs. The repositories build on these services to handle the BLE connection state, cooler data, local history storage and backend telemetry synchronisation. Command classes and view models use these repositories to read and write data. Finally, view models are injected into the UI in widgets that require them.

The app uses several external libraries for its core functionality:

- `flutter_reactive_ble` for BLE communication [42]
- `sqflite` for local SQLite storage [43]
- `http` for backend communication [44]
- `provider` for dependency injection [41]
- `command_it` for the command pattern [45]
- `cbor` for decoding historical data packets [46]
- `fl_chart` for chart visualisation [47]
- `shared_preferences` for persisting app settings [48]

The UI layer consists of pages, view models, commands, UI models, themes and reusable widgets. The pages are `HomePage`, `HistoryPage`, `TripsPage` and `SettingsPage`, which are displayed through a shared root widget with a custom navigation bar. View models are implemented as `ChangeNotifier` classes and expose formatted UI state to the view. The views access these models through `Provider`, `Consumer` and `Selector`, which limits rebuilds to the affected widgets only [49], [50].

User-triggered asynchronous actions are encapsulated in command classes. BLE commands coordinate scanning, connection, disconnection and historical synchronisation. Database commands load chart data from the database; settings commands write cooler settings; and telemetry commands send stored readings to the backend.

The implementation deliberately deviates from strict MVVM in certain instances. For example, some repositories depend on others, and shared command classes are used across view models. This approach keeps the repositories and view models smaller and avoids duplicating coordination logic for actions that affect multiple parts of the app.

The following sections describe the implementation from a feature-oriented perspective: Bluetooth connectivity, live data processing, synchronisation of historical data, settings and backend telemetry transfer.

4.1.3 Bluetooth Functionality

Bluetooth communication with the cooler is implemented using Bluetooth Low Energy (BLE). The app uses the `flutter_reactive_ble` package for scanning, connecting, reading, writing and subscribing to BLE characteristics [42]. The implementation is divided into a stateless service, a stateful connection repository and shared command classes, as seen in [Figure 18](#).

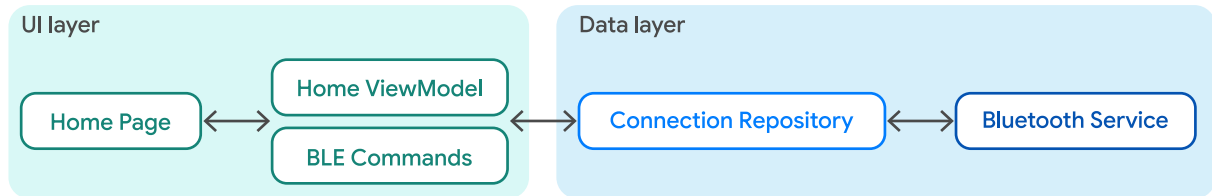


Figure 18: Overview of the Bluetooth connect/disconnect functionality, showing the separation into MVVM layers

4.1.3.1 Bluetooth Service and Data Models

The lowest layer is the `BluetoothService`. This service wraps the `FlutterReactiveBle` API and handles platform-level operations such as checking OS-level Bluetooth permissions, starting and stopping scans for coolers, connecting to devices and reading, writing or subscribing to characteristics. Before starting a scan, the service checks whether the BLE adapter is ready (i.e. switched on and permissions granted). If not, it either requests the necessary operating system permissions or displays a message to the user asking them to switch on Bluetooth.

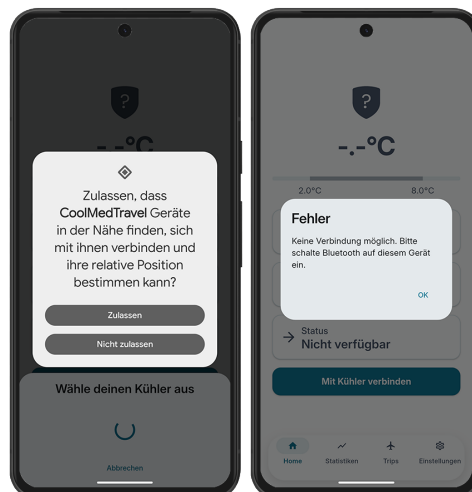


Figure 19: App requesting permissions and asking the user to switch on Bluetooth

The BLE protocol is modelled centrally in `bluetooth_ids.dart`. The cooler exposes multiple services, including battery, live data, historical data, settings and device information. More about the specific implementation of the BLE protocol can be read in [Section 3.1.1.1](#). Each BLE service is handled with a separate enum, implementing the `BleCharacteristicsId` interface. For each characteristic, the interface defines its UUID, the UUID of its parent service, as well as its permission for manual polling and its subscription mode. Implementing services as an enum prevents passing around UUIDs without context. Instead, only members of the correct services can be passed to appropriate functions. Furthermore, the additional data allows improved handling of the individual characteristics, as described in [Section 4.1.4.1](#).

Listing 7 shows the Enum modelling the characteristics of the Bluetooth Battery Service [16]. They provide the battery level, the charging or discharging status and the remaining charging or discharging time.

```

1  /// The IDs of the battery characteristics.
2  enum Battery implements BleCharacteristicsId {
3      batteryLevel._('2A19', BlePermission.read, BleSubscription.notify),
4      batteryLevelStatus._('2BED', BlePermission.read, BleSubscription.notify),
5      batteryTimeStatus._('2BEE', BlePermission.read, BleSubscription.notify);
6
7      /* ... Constructor and field definitions ... */
8
9      /// The UUID of the characteristic.
10     @override
11     Uuid get id => Uuid.parse(_id);
12
13     /// The UUID of the parent service. CoolerBleService contains all BLE service enums
14     @override
15     Uuid get parentServiceId => CoolerBleService.battery.id;
16
17     /// The manual polling mode of the characteristic.
18     @override
19     BlePermission get polling => _polling; // read, write, readWrite or none
20
21     /// The subscription mode of the characteristic.
22     @override
23     BleSubscription get transmissionMode => _transmissionMode; // notify, indicate or none
24 }

```

Listing 7: Example of a BleCharacteristicsId implementation

4.1.3.2 Connection Repository

The ConnectionRepository manages the active BLE connection. It provides streams for devices found during a Bluetooth scan, as well as status updates for the currently connected device. During scanning, discovered devices are collected and sorted by signal strength, meaning that nearby devices are displayed first in the UI.

The scan is configured to only return devices that advertise the BLE service IDs exposed by the cooler. This was intended to limit the results to cooler devices. However, this filtering is not completely effective; as seen in Figure 20, other devices are still shown, albeit fewer than before. Therefore, the app is designed to handle cases where a user selects a non-cooler BLE device. If the app attempts to interact with a non-existing characteristic, the repository terminates the connection and reverts to the disconnected state.



Figure 20: App scanning for Bluetooth devices, showing a cooler and other non-cooler devices

When a user selects a device, the repository stops scanning and establishes a connection through the `BluetoothService`. A `Completer` is then used to wait for the BLE library to report a “Connected” state before the app continues. Without this measure, the iOS app attempts to interact with characteristics before the iOS Bluetooth stack has completed the connection process. Therefore, the subscription request never reaches the cooler, resulting in no data being shown in the app despite the cooler having an active Bluetooth connection. [Listing 8](#) shows the connection process in the `ConnectionRepository`.

```

1  /// Connects to a previously discovered BLE device.
2  Future<void> connect(DiscoveredDevice device) async {
3      // Stop the scanning process and disconnect any previously connected devices
4      await stopScanning();
5      await disconnect();
6
7      // Connect to the cooler and store the device information.
8      final connectionStream = _bluetoothService.connect(device.id);
9      _connectedDevice = device;
10
11     // Start a completer to avoid issues on iOS
12     final completer = Completer<void>();
13
14     // Subscribe to the stream to start the connection
15     _connectedDeviceStatus = connectionStream.listen(
16         // Function to execute when a status update is received from the cooler
17         (update) async {
18             // Add the current state of the BLE connection to the controller
19             final state = update.connectionState;
20             _connectedDeviceStatusController.add(state);
21
22             // When the device connected, complete the completer to continue with
23             // regular code execution.
24             if (state == DeviceConnectionState.connected &&
25                 !completer.isCompleted) {
26                 completer.complete();
27             }
28
29             // Release all resources on disconnect and throw error in completer.
30             if (state == DeviceConnectionState.disconnected ||
31                 state == DeviceConnectionState.disconnecting) {
32                 if (!completer.isCompleted) {
33                     completer.completeError(Exception('Connection failed'));
34                 }
35                 await disconnect();
36             }
37         },
38         // Disconnect the device when an error occurs.
39         onError: (Object e) async {
40             if (!completer.isCompleted) {
41                 completer.completeError(e);
42             }
43             await disconnect();
44         },
45         // Clean up after the stream has been closed
46         onDone: disconnect,
47     );
48     // The code will wait here until completer.complete()/completeError() is
49     // called or the specified timeout is reached.
50     return completer.future.timeout(const Duration(seconds: 10));
51 }

```

Listing 8: Connect to a BLE device in `ConnectionRepository`

The connection repository also provides methods for reading, writing and subscribing to characteristics. Before executing an operation, it checks whether the device is connected and whether the characteristic supports the requested operation. This prevents invalid operations, such as writing to a read-only characteristic.

Disconnection from the connected cooler is handled independently of the `BluetoothService`. The repository merely cancels the connection subscription received from the service and broadcasts a “Disconnected” status to listeners of the device status stream.

4.1.3.3 View Model and UI

User-triggered BLE actions are exposed through `BLECommands`. These commands coordinate scanning, connection, disconnection and historical synchronisation. Once a connection has been established, the connect command activates the BLE data repositories, enabling the app to start receiving data from the cooler. Views are also able to adjust to the status of a command. For example, the history page shows a loading spinner when the history synchronisation command is running.

The UI accesses this functionality through the `HomePage` and the `HomeViewModel`. The home screen can open a device picker to start scanning, connect to or disconnect from a selected device and display an error message if Bluetooth is unavailable. The view model converts the BLE and cooler status streams into UI states, such as “disconnected”, “connecting” or “connected”. It also formats values such as the remaining battery time into UI-ready strings.

4.1.3.4 Limitations

The Bluetooth standard offers two different methods of connecting devices: Pairing and Bonding. The former is ethereal, meaning that once the connection is terminated, it is not possible to automatically reconnect. With Bonding, however, the connection information is stored on the device. When two bonded devices have Bluetooth enabled, they can automatically reconnect [51]. For this prototype, only pairing was implemented due to limitations on the simulator’s side as well as `flutter_reactive_ble` not supporting a straightforward way to implement bonding or access already bonded devices through the OS. This limitation means that the cooler must always be manually reconnected when the app is opened.

Overall, the Bluetooth implementation separates platform-specific BLE access from application-specific connection logic and UI state. The service wraps the external BLE API, the repository manages the connection state, and commands and the view model connect this functionality to the UI.

4.1.4 Live Data Processing

Live data processing is managed by the `LiveDataRepository` and `BatteryDataRepository`, prepared by the `HomeViewModel` and displayed on the `HomePage`. The live data shown on the home screen includes the current cooler temperature, battery percentage, remaining battery time and the Bluetooth connection status. The cooler also transmits additional data, such as humidity or ambient temperature, which is processed in the app and stored in the local database, but not visible to the user.

The `HomeViewModel` receives data from several repositories: The `ConnectionRepository` provides the BLE adapter status (e.g. whether Bluetooth is switched on), the list of discovered devices during a scan for Bluetooth devices and the name of the connected device. The `BatteryDataRepository` parses the current battery data from the pre-defined BLE Battery Service, and the `LiveDataRepository` provides the current temperature values from the Live Data BLE service implemented by the industry partner. Lastly, the `CoolerStatusRepository` exposes the current status of the cooler. All repositories listed above expose their models through streams, which the `HomeViewModel` subscribes to. As all view models implement `ChangeNotifier`, the view model calls `notifyListeners()` after it has received new data from a stream. This causes the affected UI elements on the `HomePage` to update [52].

Figure 21 shows the Live Data processing implementation separated into MVVM layers.

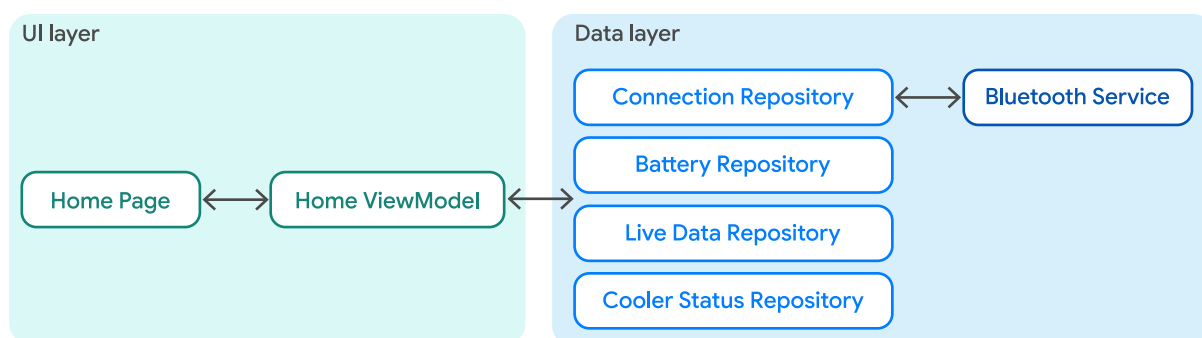


Figure 21: Overview of the Live Data processing implementation, showing the separation into MVVM layers

4.1.4.1 BLE Service Repositories

Each BLE service has its own repository and model, located in the `repositories/cooler_data` and `models/cooler_data` directories, respectively. Since most services use characteristics with subscription capabilities, the `IBleData` interface has been created to standardise subscription handling across these repositories. The interface handles starting and stopping subscriptions for each characteristic, and provides the `updateModel()` member, as seen in Listing 9. The BLE Services repositories must override this method; the repositories receive the raw BLE data and the characteristic from which it was sent for parsing and inserting the data into their respective model. The interface then places the updated model on the respective stream for the subscribed view models.

```

1  /// Returns a updated [Model] based on the received [data] for [characteristic].
2  @visibleForOverriding
3  Model updateModel(Model? current, Id characteristic, List<int> data);
  
```

Listing 9: Signature of `updateModel` in the `IBleData` interface

One of the issues encountered during the implementation of parsing was the correct handling of different data types. BLE encodes all data as little-endian byte arrays [53]. The `flutter_reactive_ble` library then exposes this data as a `List<int>` to the app. The industry partner has encoded the data in the Live Data service in two ways. Firstly, all floating-point numbers have been converted to integers by applying a scaling factor that places all digits before the decimal point. Secondly, the values are stored as variously sized integers. The app must reverse this process: The incoming data must be parsed as the correctly-sized integer and the scaling factor needs to be removed, as shown in Listing 10 and Listing 11. Both the integer size and the scaling factor are stored in the `LiveData` enum, which is described in Section 4.1.3.1.

```

1  @override
2  LiveDataPoint updateModel(LiveDataPoint? current, LiveData characteristic, List<int> data) {
3      final parsedInt = data.asInt(characteristic.dataType);
4      final scaledDouble = parsedInt.toDouble() / characteristic.scalingFactor;
5      final liveData = current ?? LiveDataPoint.empty();
6      return switch (characteristic) {
7          LiveData.temperatureInsideTop => liveData.copyWith(temperatureInsideTop: value),
8          LiveData.temperatureInsideBottom => liveData.copyWith(temperatureInsideBottom: value),
9          /* ... */
10     }
11 }
  
```

```

1  /// BLE sends its data in little endian
2  const Endian endianness = Endian.little;
3
4  /// Extension to parse data from BLE characteristics.
5  extension FromBleExtension on List<int> {
6    /// Parses a [int] from a [List<int>] based on the integer size given in [dataType].
7    int asInt(BleIntegerDataType dataType) {
8      // Wrap the received data into a ByteData to convert it from the respective
9      // C int size into Dart's int type
10     final byteData = ByteData.sublistView(Uint8List.fromList(this));
11     return switch (dataType) {
12       BleIntegerDataType.u8 => byteData.getUint8(0),
13       BleIntegerDataType.u16 => byteData.getUint16(0, endianness),
14       BleIntegerDataType.u32 => byteData.getUint32(0, endianness),
15       BleIntegerDataType.u64 => byteData.getUint64(0, endianness),
16       BleIntegerDataType.i16 => byteData.getInt16(0, endianness),
17     };
18   }
19 }

```

The implementation of the pre-defined Battery Service has also not been straightforward. Since the characteristic data is often encoded as bit flags, manual bit manipulation is necessary [54], [55]. For example, [Listing 12](#) shows the parsing of the Battery Level Status characteristic. The necessary information is stored in bits 13 and 14. Therefore, the bytes containing these bits must be extracted and bit shifting and masking needs to be performed.

```

1  /// Parses the bitfields from [Battery.batteryLevelStatus] into usable fields.
2  /// Currently, only [BatteryData.batteryChargeState] is implemented.
3  ///
4  /// Structure of the Battery Level Status as necessary for this project:
5  /// - Bits 0-7: Flags
6  /// - Bits 8-24: Power State
7  ///   - Bit 8: Battery Present
8  ///   - Bits 9-10: Wired External Power Source Connected
9  ///   - Bits 11-12: Wireless External Power Source Connected
10  ///   - Bits 13-14: Battery Level Status
11  BatteryData _parseBatteryLevelStatus(BatteryData battery, List<int> data) {
12    if (data.length < 3) {
13      throw StateError('BatteryLevelStatus expected at least 3 bytes, got ${data.length}');
14    }
15
16    // First, drop the flags as they are not needed
17    // The flags take up the first 8 bit, so the first entry can be skipped.
18    // Then the Power State bits can be taken.
19    final powerState = data.skip(1).take(2).toList();
20
21    // Get the battery charge state from the power state, contained in bits 6-7.
22    // Get the first 8 bits of powerState in index 0 and shift them to bits 0-1.
23    // Remove all other bits with a bitmask.
24    final batteryChargeState = (powerState[0] >> 5) & 0x03;
25
26    return battery.copyWith(
27      batteryChargeState: BatteryChargeState.values.parse(batteryChargeState),
28    );
29 }

```

Listing 12: Parsing of the BLE Battery Service's Battery Level Status characteristic

After the BLE data parsing has concluded, the `IBleData` interface places the new model returned from `updateModel()` onto a stream that `HomeViewModel` has subscribed to. The view model will thus be notified of any changes in the BLE data.

4.1.4.2 View and View Model of the Home Page

The view model converts repository data into UI-specific values. It derives the current temperature from the latest LiveDataPoint, the battery percentage and remaining battery time from the latest battery data and the displayed status from the BLE and cooler status. The status calculation also handles intermediate states. If the app is connected but has not yet received battery or temperature data, the view model reports the UI state as connecting.

[Listing 13](#) shows how the HomeViewModel subscribes to the repository streams and updates its internal state when new data is received.

```
1  /// List of all active subscriptions for cleanup.
2  final List<StreamSubscription<dynamic>> subscriptions = [];
3
4  /// Handles the subscription of all streams in the view model.
5  void _subscribeToStreams() {
6    /// Helper to listen and notify with error handling. Specify the variable to
7    /// store the value with [setter].
8    /// `(value) => model = value;`
9    void listen<T>(Stream<T> stream, void Function(T) setter) {
10      stream
11        .listen(
12          (data) { // Function executed when stream sends new data
13            setter(data);
14            notifyListeners();
15          },
16          onError: (_) => dispose(),
17        )
18        .addTo(subscriptions);
19    }
20
21    listen(_connectionRepository.bleStatus, (s) => _bleStatus = s);
22    listen(_connectionRepository.discoveredDevices, (d) => _discoveredDevices = d);
23    listen(_batteryRepository.characteristicsStream, (d) => _batteryData = d);
24    listen(_liveDataRepository.characteristicsStream, (d) => _liveDataPoint = d);
25    listen(_coolerStatusRepository.status, (s) => _coolerStatus = s);
26  }
```

Listing 13: Subscription handling for live data updates in the HomeViewModel

The home screen uses the Consumer and Selector from the “provider” library to access the view model [49], [50]. The Selector is used for widgets that only depend on selected values, such as the current status, temperature, or battery data, while the Consumer rebuilds widgets whenever any value in the view model updates. This prevents the entire page from rebuilding whenever only one value changes.

The UI state determines the visual appearance of the home screen. A disconnected cooler is shown in grey, a connected cooler in blue, and warnings and errors are shown in orange and red respectively. The status icon, which is in the form of a shield, uses the same colours and respective icons to quickly indicate the cooler’s condition. If a warning or error state is active, a message explaining the problem and possible solutions is displayed above the icon, as seen in [Figure 22](#). This enables users to recognise the cooler’s state through both colour and text.



Figure 22: Home Screen in its different states

The current temperature is displayed using the `TemperatureRange` widget. This widget receives the current temperature and the safe temperature range from the view model. The current value is displayed as a large number, ensuring that the most important live value is the most prominent feature. The widget also shows where the current temperature lies in relation to the configured safe range.

As the cooler is equipped with multiple temperature sensors, the `LiveDataRepository` calculates the current temperature displayed in the 'TemperatureWidget' as the highest reported temperature by any sensor, in accordance with the industry partner's request.

Battery information is displayed in an `InfoBox`. The view model formats the battery percentage, selects the correct battery icon, and converts the remaining battery time into a localised string. If the cooler is charging, the text shows the remaining charging time. Otherwise, it shows the estimated remaining operating time.

The Bluetooth box and connection button are also based on the UI status exposed by the view model. Depending on the current state, the button either opens the device picker to connect a cooler, or disconnects from the currently connected cooler.

The cooling status box currently only distinguishes between "Not Monitoring" and "Stable". A more detailed status display was not implemented as the industry partner had not yet defined how such information should be presented to the user in the final product.

Overall, live data processing adheres to the MVVM architecture employed throughout the app. The repositories provide raw data and connection state. The view model then transforms this data into UI-ready values, which are displayed on the home page through reusable widgets.

4.1.5 History Synchronisation

The history synchronisation transfers historical telemetry logs from the cooler to the app. Unlike the ephemeral live data, historical data is stored locally in a SQLite database. This enables the app to display temperature statistics even after the cooler has been disconnected.

Once the cooler is connected to the app, historical telemetry logs are automatically sent to the app as they are generated. This process is similar to that described in [Section 4.1.4](#) for live data, which is sent via a `notify` subscription. The historical data is then stored in the database. However, this only covers data generated while an active BLE connection exists. To retrieve logs recorded when the cooler was disconnected, a separate historical data synchronisation mechanism had to be implemented.

The implementation separates the BLE synchronisation logic from local persistence, visualised in [Figure 23](#). The `HistoricalDataRepository` handles the BLE transfer, packet reassembly and parsing of historical log data. The `HistoryRepository`, on the other hand, accesses the local database through the `AppDatabaseService`. This distinction ensures that communication with the cooler is kept separate from the storage and retrieval of already synchronised data.

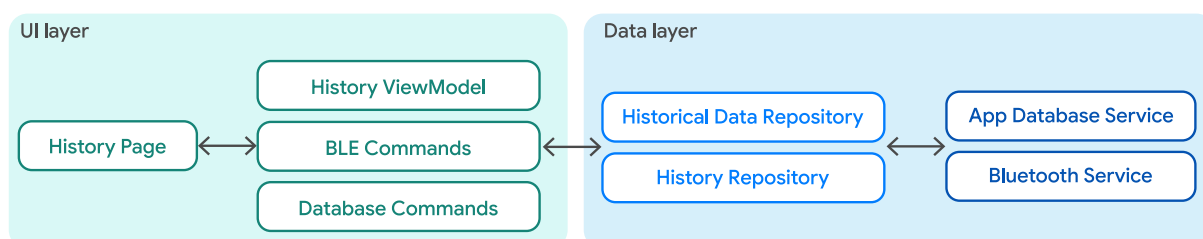


Figure 23: Overview of the historical synchronisation implementation, showing the separation into MVVM layers

4.1.5.1 Starting a Synchronisation

Synchronisation is triggered when the user opens the history page. The `HistoryViewModel` first checks whether a cooler is connected. If not, it only reloads existing data from the local database. If a cooler is connected, the historical sync command is started and the chart data is reloaded afterwards.

The historical sync command itself is defined in `BleCommands`. It calls `syncHistoricalData()` on the `HistoricalDataRepository` and then waits until the repository reports that synchronisation has completed. Depending on the number of logs that need to be synchronised, the process may take a considerable amount of time. The usage of a command in this scenario proves to be advantageous. The history page can listen to `syncHistoricalData.isRunning` to check whether the sync is still ongoing and if so, show a loading spinner. In its current form, there is no progress indicator, possibility irritating users when there is no visible progress on this long operation. The industry partner has expressed interest in expanding the BLE protocol to enable the app to read the latest log counter from the cooler and display a progress bar.

A timeout is used in the command to ensure that the UI does not remain in the loading state indefinitely if the cooler's completion signal does not reach the app.

Before a new synchronisation begins, the `HistoricalDataRepository` clears any old packet buffers. It then requests the highest stored log counter from the `HistoryRepository` and writes this value to the cooler's `lastCounter` characteristic. This informs the cooler of the point from which it should start sending historical data, as described in [Section 3.5.3](#). After a short delay for the iOS Bluetooth stack, the repository writes `0x01` to the cooler's `syncControlPoint` characteristic to initiate the transfer.

This process is shown in [Listing 14](#).

```
1  /// Value to send to [HistoricalData.syncControlPoint] to start
2  /// synchronizing history data.
3  const historicalDataStartSync = [0x01];
4
5  /// Triggers the synchronisation of historical data.
6  Future<void> syncHistoricalData() async {
7      // Clear the old buffer when a new sync starts
8      _chunkBuffers.clear();
9
10     // Check the database what the last received data index from the cooler is.
11     final lastCounter = await _historyRepository.getLatestLogCounter();
12
13     // Set a new completer for syncing.
14     _isSyncing = Completer<void>();
15
16     try {
17         // Send the counter to the cooler to tell it from what index the cooler
18         // should start sending data.
19         await connectionRepository.writeCharacteristic(
20             HistoricalData.lastCounter,
21             _uint32ToBytes(lastCounter),
22         );
23
24         // The iOS Bluetooth stack needs some delay in between characteristics.
25         await Future<void>.delayed(const Duration(milliseconds: 100));
26
27         // Trigger the synchronisation of historical data by writing
28         // '0x01' to the sync control point
29         await connectionRepository.writeCharacteristic(
30             HistoricalData.syncControlPoint,
31             historicalDataStartSync,
32         );
33     } catch (error) {
34         _isSyncing?.completeError(error);
35         // Empty catch to allow for re-throw.
36         await _isSyncing?.future.catchError((_) {});
37         throw Exception('Failed to sync historical data: $error');
38     }
39 }
```

Listing 14: Starting historical synchronisation by sending the latest stored log counter and triggering the sync control point

4.1.5.2 The Synchronisation Process

The cooler sends historical data through the `historicalLog` characteristic, which includes data not included in the live data, such as error flags or the current temperature range. To ensure efficient transfer, the industry partner has implemented CBOR encoding (Concise Binary Object Representation) for the historical data. After converting a log to CBOR, its total size is 119 bytes.

Older devices with Bluetooth 4.0 only support packets of up to 20 bytes [56]. To transfer the data with this limitation, the team has modified the BLE protocol to divide the data into smaller chunks and add metadata before encoding it with CBOR. Each chunk contains a message ID, a chunk index, the total number of chunks belonging to the same message and the payload. The repository parses the metadata incoming chunks and stores the payload in a buffer until all chunks for a message have been received. It then reassembles the complete CBOR payload and removes the buffer for that message.

The reassembly process is shown in [Listing 15](#).

```
1  /// Try to reassemble a complete CBOR packet from a chunked BLE packet in [data].
2  /// Returns the complete CBOR payload or null if the packet is incomplete.
3  CborPayload? _tryReassemblePacket(HistoricalData characteristic, List<int> data) {
4      // Create a CBOR chunk from the raw data
5      final chunk = CborChunk.fromList(data);
6
7      if (chunk == null) return null;
8
9      // Create a buffer for each message ID in the size of `totalChunks`
10     final chunks = _chunkBuffers.putIfAbsent(
11         chunk.messageId,
12         () => CborPayloadChunks.filled(chunk.totalChunks, null),
13     );
14
15     // If a message ID gets reused while an old buffer still exists,
16     // reset the buffer.
17     final key = chunk.messageId;
18     if (chunks.length != chunk.totalChunks) {
19         _chunkBuffers[key] = CborPayloadChunks.filled(chunk.totalChunks, null);
20     }
21
22     // Store the payload of the chunks in the buffer
23     _chunkBuffers[key]![chunk.chunkIndex] = chunk.payload;
24
25     // Check if all chunks are present
26     final currentChunks = _chunkBuffers[key]!;
27     final hasAllChunks = currentChunks.every((chunk) => chunk != null);
28
29     // If not, this means the package could not be transmitted, return null
30     if (!hasAllChunks) {
31         return null;
32     }
33
34     // Reassemble the full CBOR payload from the chunks
35     final completePayload = <int>[
36         for (final chunk in currentChunks) ...chunk!,
37     ];
38
39     // Remove the buffer for this message ID
40     _chunkBuffers.remove(key);
41
42     // Return the packet
43     return completePayload;
44 }
```

Listing 15: Reassembling chunked BLE packets into a complete CBOR payload

The reassembled payload is then decoded by the `HistoricalDataParser`, as can be seen in [Listing 16](#). To reduce the size of the BLE payload, the cooler sends each log entry as a CBOR map using integer keys instead of textual property names. The parser interprets these keys and maps the decoded values to the corresponding fields of `HistoricalLogPoint`.

```

1  /// Recieves the CBOR payload from the cooler and parses it into a [HistoricalLogPoint].
2  HistoricalLogPoint parse(List<int> cborData) {
3      // Decode the byte array from CBOR into a Dart object.
4      final Object? decoded;
5      try {
6          decoded = cborDecode(cborData).toObject();
7      } on FormatException catch (e) {
8          throw FormatException('Failed to decode historical log CBOR: $e', e);
9      }
10
11     // If the decoded object is not a map, the data has not been received in the
12     // expected format.
13     if (decoded is! Map) {
14         throw FormatException('Expected CBOR map, got ${decoded.runtimeType}');
15     }
16
17     // Cast all keys of the decoded CBOR map to integers
18     final map = decoded.cast<int, dynamic>();
19
20     /// Gets the value of the specified `characteristics` from the integer CBOR map.
21     /// If [CoolerDataCategory] is set, it applies the scaling value.
22     /// Returns `null` if the key does not exist.
23     T? at<T>(HistoricalLogCbor characteristic) {
24         final value = map[characteristic.index];
25         if (value == null) return null;
26
27         // Handle scaling for values with a category (temperature, humidity etc)
28         if (characteristic.category != null &&
29             characteristic.category!.scalingFactor > 1) {
30             return (value as num) / characteristic.category!.scalingFactor as T?;
31         }
32
33         // DateTime should be stored as Unix timestamps in seconds
34         if (T == DateTime) {
35             return DateTime.fromMillisecondsSinceEpoch((value as int) * 1000) as T?;
36         }
37
38         // Handle parsing lists
39         if (T == List<int>) {
40             return (value as List).cast<int>() as T?;
41         }
42
43         // All other types do not need special treatment, simply cast them
44         return value as T?;
45     }
46
47     return HistoricalLogPoint(
48         recordedAt: at(HistoricalLogCbor.recordedAt)!, // Required
49         logCounter: at(HistoricalLogCbor.logCounter)!, // Required
50         temperatureInsideTopAvg: at(HistoricalLogCbor.temperatureInsideTopAvg), // Optional
51         temperatureInsideBottomAvg: at(HistoricalLogCbor.temperatureInsideBottomAvg), // Optional
52         ...
53     );
54 }

```

Listing 16: Decoding a CBOR payload and mapping integer keys to a `HistoricalLogPoint`

4.1.5.3 Storing in Local Database

After parsing, the log point is written to the local database. Before it is inserted, the structured log point is converted into an individual `TelemetryReading` entry by the `HistoricalLogPoint.toReadings()` function. Null values are skipped, enabling historical records with missing values to still be stored. This also makes the data model more flexible in case the set of telemetry values changes in the future.

The value `current_temperature` is derived from the higher value of the top and bottom average inside temperatures, in line with the live data, as discussed in [Section 4.1.4.2](#). This creates a single temperature value that can be used by the history chart.

The `AppDatabaseService` is responsible for storing readings in a generic telemetry schema. As seen in [Section 3.5](#), this schema consists of telemetry properties and telemetry readings. Each reading contains the property, the recorded value, the timestamp and the log counter. Duplicate readings are disregarded based on property and log counter, thereby ensuring more reliable synchronisation and avoiding duplicated content in the database. The detailed structure of the local database is described in [Section 3.5](#).

The history page does not display data directly from the BLE stream. Instead, it loads chart data from the local database through the `DatabaseCommands` class. The command retrieves the `current_temperature` property and converts the stored readings into chart data points. The `HistoryViewModel` then filters these points according to the selected time frame and calculates the minimum, maximum and average values, which are shown on the `HistoryPage` below the chart.

Two chart types are visible on the `HistoryPage`. A line chart shows the temperature development over the last 24 hours. For the last 7 days, the readings are grouped by day and displayed as a bar chart showing the minimum and maximum temperatures recorded on each day. Users can switch between these views using a toggle button. Both chart types show the configured safe temperature range, enabling users to compare recorded temperatures with the permitted storage range.

Overall, history synchronisation separates BLE transfer, CBOR parsing, local persistence and visualisation. The command starts and monitors the synchronisation process. The `HistoricalDataRepository` receives and reassembles packets, the parser converts CBOR data into domain models, the `HistoryRepository` and database service store the resulting readings, and the `HistoryViewModel` prepares the stored data for display on the `HistoryPage`.

4.1.6 Settings

The settings implementation distinguishes between app settings and cooler settings. App settings only affect the mobile application, while cooler settings are written to the connected cooler via BLE. There are currently two available settings: theme mode is an app setting, whereas the configured temperature range is both stored locally and sent to the cooler.

App settings are persisted through the `AppSettingsRepository` and `PreferenceService`, while cooler settings are additionally written to the cooler in `CoolerSettingsRepository`. The `SettingsViewModel` is responsible for handling validation of the settings, while `SettingsCommands` covers error handling and debouncing on the `SettingsPage`. This interaction is visible in [Figure 24](#).

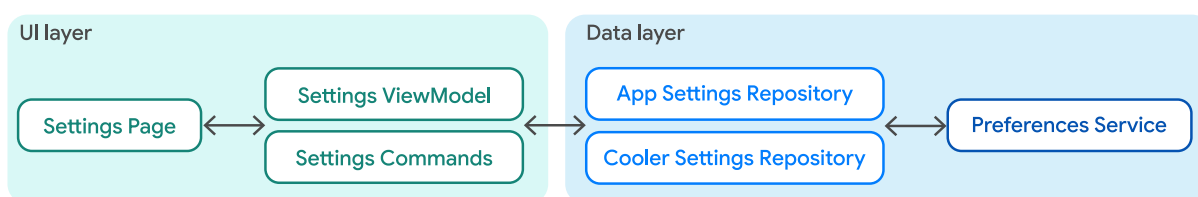


Figure 24: Overview of the settings implementation, showing the separation into MVVM layers

4.1.6.1 App and Cooler Settings Handling

The app settings are represented by the `AppSettings` model. It contains the minimum and maximum temperatures and the selected theme mode. Default values are defined in the model, with a default temperature range from 2°C to 8°C and the system theme as the default theme mode.

The local persistence of app settings is handled by the `AppSettingsRepository`. This repository stores its current state in a `ValueNotifier`, enabling view models to register callback functions in order to respond to changes [57]. For persistent storage, the repository delegates to the `PreferencesService`, which wraps the `SharedPreferencesAsync` API of the `shared_preferences` plugin [48]. This Flutter library provides a simple key-value storage mechanism and wraps the platform-specific preference systems. The temperature range and theme mode are loaded when the app starts. When the app is started for the first time and no persistent data is present, the default values are written to the preferences.

The cooler settings are handled separately by the `CoolerSettingsRepository`. When a user saves a new temperature range, the values are written to the cooler's BLE settings characteristics. Before being sent to the cooler, the values are converted into 16-bit little-endian byte arrays to conform with BLE [53].

The write process is coordinated by `SettingsCommands`. First, the `saveTemperatureRangeCommand` writes the new range to the cooler. Only if this succeeds is the local copy in the `AppSettingsRepository` updated. This prevents the app from showing a temperature range as saved when the cooler did not accept the setting. This order is shown in [Listing 17](#).

```
1 late final Command<({double min, double max}), void> saveTemperatureRangeCommand =
2   Command.createAsync((range) async {
3     // Write to the cooler first.
4     // If this fails, the local app settings are not updated.
5     await _coolerSettingsRepository.updateTemperatureRange(
6       range.min,
7       range.max,
8     );
9
10    // Store a local copy after the cooler write succeeded.
11    await _appSettingsRepository.updateTemperatureRange(
12      range.min,
13      range.max,
14    );
15  },
16  initialValue: null,
17  // Listeners can subscribe to .error to handle errors.
18  errorFilterFn: (error, stackTrace) => ErrorReaction.localHandler,
19 );
```

Listing 17: Saving the temperature range by first writing it to the cooler and then updating the local app settings

One limitation of the current implementation is that, although temperature settings are written to the cooler, they are not read back from it. Consequently, any changes made by another application are not automatically reflected in the settings when this application reconnects to the same cooler. This issue is discussed further in [Section 6.1.8](#).

4.1.6.2 Settings Validation

The `SettingsViewModel` validates and prepares user input before executing commands. It ensures that the minimum temperature is not below zero, that it is smaller than the maximum temperature, and that the range is at least one degree wide. If validation fails, the view model reverts to the last saved values and displays an error message in the form of a red `SnackBar` [58] to inform the user what caused the problem. When the temperature range is saved successfully, a green `SnackBar` is shown to confirm the change. Decimal input is accepted in either comma or dot notation. The validation and command execution are shown in [Listing 18](#).

```

1 Future<void> saveTemperatureRange() async {
2   // Don't save if there is currently another save running or no changes.
3   if (isSavingTemperatureRange || !_hasUnsavedTemperatureChanges) {
4     return;
5   }
6
7   // Don't save if the temperature range is invalid.
8   if (!isTemperatureRangeValid) {
9     // Get the appropriate error message for the user.
10    final message = switch (true) {
11      _ when minTemperature < 0 => SettingsMessage.minTemperatureBelowFreezing,
12      _ when minTemperature >= maxTemperature =>
13        SettingsMessage.minTemperatureAboveMaxTemperature,
14      _ when maxTemperature - minTemperature < minimumTemperatureRangeDelta =>
15        SettingsMessage.temperatureRangeTooSmall,
16      _ => SettingsMessage.temperatureRangeSaveFailed,
17    };
18
19    _messageController.add(message);
20    _revertToLastSaved();
21    notifyListeners();
22    return;
23  }
24
25  try {
26    await commands.saveTemperatureRangeCommand.runAsync((
27      min: minTemperature,
28      max: maxTemperature,
29    ));
30  } on Object {
31    // Error is handled via listener
32  }
33 }

```

Listing 18: Validating the temperature range before executing the save command

4.1.6.3 Settings Page

The settings page is implemented through the `SettingsPage` and the `SettingsViewModel`. The page contains two main sections: cooler and appearance settings. The cooler section enables the user to configure the minimum and maximum temperatures of the safe cooling range. The appearance section contains a switch for toggling between light and dark mode. The page also contains an “About” tile that opens the standard Flutter About dialog, which displays the version number of the app, as well as licences of all dependencies.

The temperature range input visible on the `SettingsPage` uses the reusable `TemperatureRangeBox` widget. This widget contains two number fields for the lower and upper temperature limits. The settings page saves the values when focus is lost from both fields, which allows users to edit both values before the range is validated and written. While the save command is running, the input fields are disabled. In the future, a more explicit “save” button would likely make this interaction clearer. However, the design document provided by the industry partner did not include such a button, so the current implementation follows the given design.

The dark mode is integrated through Flutter’s `ThemeMode`. The selected theme mode is stored in the `AppSettingsRepository` and exposed by the `SettingsViewModel`. In `main.dart`, the root `MaterialApp` reads this value from the `SettingsViewModel` and applies either the light, dark or system theme to the entire application.

4.1.6.4 Localisation

The groundwork for localisation was completed by configuring Flutter’s localisation system and moving strings intended for the user into localisation files. Currently, only German strings are provided, but the structure allows additional languages to be added with minimal effort at a later stage. New translations can be introduced by adding another localisation file. Once additional languages are available, a language switcher can be added on the settings page to enable users to switch between them.

4.1.6.5 Permeate Settings Through the Whole App

When the user changes the cooling temperature range on the settings page, all other locations where the temperature range is displayed are also updated accordingly. To achieve this, the `AppSettingsRepository` stores the settings in a `ValueNotifier` [57]. Pages displaying the range, such as the Home and History page, have the capability to add listeners to `ValueNotifier.value`. As with streams, this function is triggered when the value changes. This mechanism guarantees that any changes made by the user to the settings are consistently synchronised across the entire application.

Overall, the settings implementation separates local app preferences from the configuration of the BLE-based cooler configuration. The settings page collects user input, which the view model then validates and formats. Commands coordinate save operations, while repositories persist local settings or write cooler settings. Services wrap the underlying storage and BLE APIs.

4.1.7 Send Telemetry to Backend

For data analysis purposes, the app sends telemetry data to the backend via HTTP requests. As part of this Bachelor's thesis, this data consists of the historical readings stored in the local database. The transmission of telemetry data is initiated after the history synchronisation described in [Section 4.1.5](#) has been completed, and the newly received values have been written to the local database. This ensures that any newly synchronised readings are available locally before the transfer to the backend starts.

When the history page is accessed, a command is triggered that initiates the process in the `BackendTelemetryRepository`. The repository retrieves information about the cooler from `DeviceInformationRepository` and then sends HTTP requests via `HTTPService`, as shown in [Figure 25](#).

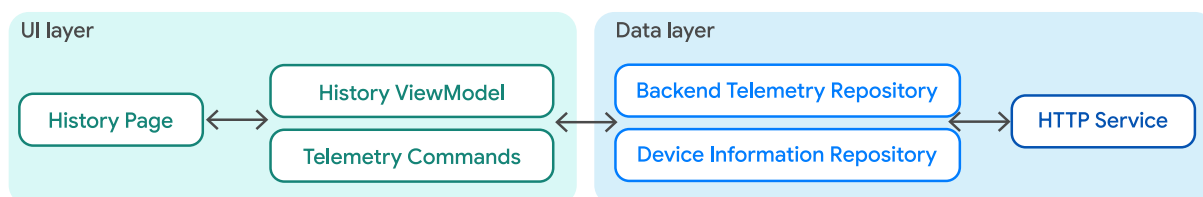


Figure 25: Overview of the backend telemetry implementation, showing the separation into MVVM layers

4.1.7.1 Reading the Cooler's Device Information

The user-triggered action is wrapped in `TelemetryCommands`. The `sendTelemetryToBackend()` command delegates the actual synchronisation logic to the `BackendTelemetryRepository`. Hiding the backend upload behind a command ensures consistency with the app's other asynchronous operations and enables the view model to trigger the upload without relying directly on the HTTP service.

The `BackendTelemetryRepository` itself coordinates the data transfer. Before sending any readings, it retrieves the cooler's serial number, firmware version and hardware revision from the `DeviceInfoRepository`. These values are required to associate the uploaded telemetry data with the correct cooler in the backend. If the device information is unavailable, the repository throws an error instead of sending incomplete data.

The `DeviceInfoRepository` is a BLE characteristics repository comparable to `LiveDataRepository` or `HistoricalDataRepository`. However, it does not implement the `IBleData` interface, as shown in [Section 4.1.4.1](#). This is due the Device Info service not offering any characteristics with subscription functionality, thus making `IBleData` superfluous. It instead listens to changes to the cooler status in `CoolerStatusRepository`. When a cooler is connected, the cooler status changes to `connected` and the `HistoricalDataRepository` reads the serial number, firmware revision and hardware version once and stores it. When a telemetry command is run, the device information should already be present in the repository and does not need to be fetched again during the active connection.

4.1.7.2 Sending the HTTP requests

The backend synchronisation itself uses two HTTP requests. First, the app sends a GET request to the backend to retrieve the latest log counter already stored for the current cooler; this is the same log counter used for the historical synchronisation. It then queries the local database for all readings with a higher log counter. This prevents readings that have already been uploaded to the backend from being sent again.

This process is shown in [Listing 19](#).

```
1 Future<void> syncWithBackend() async {
2     final serialNumber = _deviceInfoRepository.serialNumber;
3     final firmwareRevision = _deviceInfoRepository.firmwareRevision;
4     final hardwareRevision = _deviceInfoRepository.hardwareVersion;
5
6     if (serialNumber == null || firmwareRevision == null || hardwareRevision == null) {
7         throw StateError(
8             'Device information must be available before sending data to the backend',
9         );
10    }
11
12    try {
13        /// Fetch the latest log_counter uploaded to the backend
14        final latestCounter = await _httpClient.getLatestCounter(serialNumber);
15
16        /// Read every log with a log_counter > latestCounter
17        final readings = await _historyRepository.getReadingsSinceCounter(latestCounter);
18
19        if (readings.isEmpty) {
20            log('No new readings to sync for serial $serialNumber', name: 'TelemetryHttpSync');
21            return;
22        }
23
24        /// Send the actual readings to the backend
25        await _httpClient.postReadings(
26            serialNumber,
27            firmwareRevision,
28            hardwareRevision,
29            readings,
30        );
31
32        log(
33            'Successfully synced ${readings.length} readings for serial $serialNumber',
34            name: 'TelemetryHttpSync',
35        );
36    } on Exception catch (error) {
37        // Offline-compatible: Just log the error and don't throw,
38        // so the app flow isn't interrupted.
39        log(
40            'Failed to sync telemetry with backend: $error',
41            name: 'TelemetryHttpSync',
42            error: error,
43        );
44    }
45 }
```

Listing 19: Backend telemetry synchronisation using the latest backend log counter and locally stored readings

If no new readings are available, the synchronisation stops without sending a second request. If new readings exist, the app sends them to the backend using a POST request. The request body contains the cooler's serial number, the firmware and hardware revisions and the list of telemetry readings. The readings are serialised as JSON using the `toJson()` method of `TelemetryReading`, which is automatically called when using `jsonEncode` [59].

The low-level HTTP communication is implemented in the `HttpClient`. This class defines the backend URLs and sends both the GET request for the latest counter and the POST request containing the readings. The backend base URL is stored centrally in `BackendConfig`, meaning that the address can be changed without the need to modify the repository logic.

The backend upload is designed to be compatible with offline use. If an HTTP request fails during synchronisation, the error is logged but not rethrown. This means that the app flow is not interrupted if the backend is temporarily unavailable. In this case, the readings remain stored in the local database and can be uploaded during a subsequent synchronisation attempt.

Overall, the telemetry upload separates command handling, repository coordination and HTTP communication. The `HistoryViewModel` triggers the upload after historical synchronisation, `TelemetryCommands` exposes the action, `BackendTelemetryRepository` decides which readings need to be sent, `HistoryRepository` provides the stored readings from the local database, and `HttpService` performs the HTTP requests to the backend.

4.2 Backend

The backend receives, stores and exports telemetry data collected by the mobile application. Unlike the app, the backend is not user-facing; it is only used by industry partners. The focus of its implementation is therefore on providing a stable API, persisting telemetry readings and enabling the later analysis of collected cooler data.

Initially, defining the backend architecture from scratch proved challenging. While the industry partner has requested to use the Axum crate for the web server [2], required functionality for creating the infrastructure of a fully-fledged web API is split across several crates across the Rust ecosystem. The backend would require database access, configuration handling, database migrations, entity definitions, tracing and unit testing. In contrast to Flutter, Rust and Axum do not provide a recommended architecture. When the team researched architecture proposals, it became apparent that each examined project had its own unique approach.

Following the advisors suggestion, the team therefore decided to use Gerust as the basis for the backend implementation. Gerust provides a generated Rust project structure and predefined crates for common backend functionality. This enabled the team to concentrate on implementing the telemetry-specific data model and API endpoints instead of manually setting up the entire backend infrastructure.

As part of this Bachelor's thesis, the backend and database are automatically deployed onto a virtual machine provided by OST. Details can be found in [Section 5.3.3.6](#).

4.2.1 Introduction to Gerust

Gerust is a project generator for Rust backend projects [60]. It builds on Axum and SQLx [61] to create a comprehensive web API. Once a project is generated, Gerust provides five different crates that each handle a different aspect of the backend, shown in [Listing 20](#).

```
.
├── cli      // CLI tools for e.g. running DB migrations or generating project files
├── config   // Builds the configuration from environment-specific TOML files and env vars
├── db       // Encapsulates database access, migrations, as well as entity definitions
├── macros   // Contains macros, e.g. for application tests
└── web      // The web interface as well as tests for it
```

Listing 20: Overview of the Rust crates generated by Gerust

The main crate of the application is the `web` crate. It contains the main executable that starts up an Axum web server and defines the endpoints together with their controllers. It also allows middleware, such as authentication or request filtering, to be added and provides the infrastructure for API tests, as discussed in [Section 5.2.3.2](#). The second most important crate is `db`, which contains database access functionality and entities for persisting data into the database. The `cli` crate contains a command-line program to execute various database operations, as well as generating new files for entities, controllers, tests and middlewares. The `config` crate provides infrastructure to load configurations from different environments. Finally, the `macros` crate contains code for generating database tests.

For most uses, the code in the `cli`, `config` and `macros` crates does not need to be edited. Most use cases only need to edit `db` and `web`.

4.2.1.1 Database Interaction

While SQLx natively supports PostgreSQL, MariaDB and SQLite, Gerust only has built-in support for PostgreSQL. Therefore, PostgreSQL was used for the backend, as specified in [Section 3.4](#). A freshly generated project contains a Docker Compose file with two different databases for development and testing, respectively.

To begin, the Gerust CLI tool can be used to generate a new database entity: `cargo generate entity` generates a new file with the entity struct, a change set and basic CRUD functions for that entity. Once the entities have been set up, a database migration can be created. This is again possible with Gerust's CLI by using `cargo generate migration`, which will create a `up` and `down` migration file. Unfortunately, Gerust simply creates empty files, it is up to the developers to fill them with SQL statements. The migration can then be applied to the database with `cargo db migrate`.

The macros used by SQLx connect to the database at compile time to verify the entered SQL queries. To compile the code without a running database instance, offline queries can be used. When a database is running, the `cargo db prepare` command can be used to generate cached queries. By setting the `SQLX_OFFLINE=1` environment variable, the code can now be compiled without losing compile time safety.

4.2.2 Database implementation

The backend database stores three main entities: coolers, telemetry properties and telemetry readings, as described in [Section 3.4](#).

Each database entity is represented by a Rust struct in the `db` crate. In addition to these entity structs, the backend uses changesets for values that are inserted or updated. These changesets are validated before database operations are executed through the use of the `validator` crate [62]. For instance, the log counter in a `TelemetryReading` is a positive integer. However, PostgreSQL does not support unsigned integers, which is why in the Rust code, `log_counter` is of type `i64`. To prevent negative values to be written to the database, a `#[validate]` annotation can be added to the property. This prevents invalid data from being written to the database.

The entity modules provide access to the database via CRUD functions such as `load`, `load_all`, `create`, `update` and `delete`. In addition to these generic operations, some entity-specific queries are implemented. For instance, coolers can be loaded by serial number, telemetry properties by name, and telemetry readings by cooler, date range and property. The telemetry readings module also provides a query to retrieve the latest log counter value for a cooler. This value is used by the app to determine which readings still need to be uploaded. This process is described in detail in [Section 4.1.5](#) and [Section 4.1.7](#).

Listing 21 shows the changeset of a telemetry reading and the insert function using that changeset.

```
1  #[derive(Deserialize, Validate, Clone)]
2  pub struct TelemetryReadingChangeset {
3      pub cooler_id: Uuid,
4      pub property_id: Uuid,
5      pub value: JsonValue,
6      pub recorded_at: NaiveDateTime,
7      #[validate(min = 0)]
8      pub log_counter: i64
9  }
10
11  /// Create a [`TelemetryReading`] in the database with the data in the
12  /// [`TelemetryReadingChangeset`]. Returns the created [`TelemetryReading`] on success.
13  ///
14  /// # Errors
15  /// - If there is a generic database error, a [`DbError`] will be returned.
16  /// - If the data in the changeset is invalid, a [`ValidationError`] will be returned
17  pub async fn create(
18      telemetry_reading: TelemetryReadingChangeset,
19      executor: impl sqlx::Executor<'_, Database = Postgres>,
20  ) → Result<TelemetryReading, crate::Error> {
21      telemetry_reading.validate()?;
22
23      let record = sqlx::query!(
24          "INSERT INTO telemetry_readings
25          (cooler_id, property_id, value, recorded_at, log_counter)
26          VALUES ($1, $2, $3, $4, $5) RETURNING id",
27          telemetry_reading.cooler_id,
28          telemetry_reading.property_id,
29          telemetry_reading.value,
30          telemetry_reading.recorded_at,
31          telemetry_reading.log_counter,
32      )
33      .fetch_one(executor)
34      .await
35      .map_err(crate::Error::DbError)?;
36
37      Ok(TelemetryReading {
38          id: record.id,
39          cooler_id: telemetry_reading.cooler_id,
40          property_id: telemetry_reading.property_id,
41          value: telemetry_reading.value,
42          recorded_at: telemetry_reading.recorded_at,
43          log_counter: telemetry_reading.log_counter,
44      })
45  }
```

Listing 21: Changeset and insert function for a TelemetryReading

4.2.3 API

The HTTP API is implemented in the web crate using Axum controllers. These controllers receive request parameters, call the relevant functions in the db crate and then return JSON responses or HTTP status codes. Currently, the backend contains separate controllers for importing telemetry data from the app and exporting stored data for later use.

Testing of the API is discussed in [Section 5.2.3.2](#).

4.2.3.1 Importing Telemetry Data

The telemetry import process is handled by two endpoints. The first endpoint returns the latest log counter stored for a given cooler's serial number. The app uses this value before uploading telemetry data to ensure that only newer readings are sent. If there are no existing readings for the cooler, the backend returns 0 as the latest log counter.

The second import endpoint receives telemetry readings from the app. The request body contains the cooler's serial number, name and model, as well as a list of readings as JSON. The incoming JSON is parsed into Rust structs; if it can not be deserialized due to invalid data, a HTTP 422 Unprocessable Content error is returned. For each reading, the backend first resolves the telemetry property by name, and then loads the corresponding cooler by serial number. If no cooler with this serial number exists yet, a new cooler entry is created automatically. The readings are then converted into telemetry reading changesets and inserted into the database.

4.2.3.2 Exporting Telemetry Data

The export controller provides access to stored telemetry data. Readings can be requested for a specific cooler's serial number and optional filters can be applied through query parameters, such as date range and telemetry property. Internally, the controller first loads the cooler and then queries the matching readings from the database. Since the readings store the telemetry property as an ID, the controller also loads the corresponding property names and replaces the IDs in the response. This makes the exported data easier to process, as clients receive property names instead of database identifiers.

4.2.4 Limitations and Conclusion

Due to the stakeholders deciding to focus on the app, the backend has received comparatively little development time. It therefore only offers simple import and export functionality for a single cooler. Future development can expand on this foundation to realise [User Story 12 – Aggregated Sensor Data](#) and comprehensive data analysis practices.

Before deploying the backend in production, security concerns should be addressed. Due to the limited development time, no security measures were built in to the backend. Transmission is handled unencrypted via HTTP instead of HTTPS and the backend does not offer any authentication. Every HTTP client is able to send and receive data from any route. The most critical route is `GET /readings`, which only members of the industry partners should be able to use. While the exported data does not contain personally identifiable information, it can nonetheless offer insight into users' behaviour and allow guesses on their stored medication. `POST /readings` can also be abused. An attacker can insert data for any existing cooler, or insert data for a non-existing cooler. Finally, an attacker can use `/latest_counter` to confirm a certain serial number is active and to calculate the approximate uptime of any cooler. In addition, there is no rate limiting in place. Attackers can send an unlimited number of requests to cause a Denial-of-Service on the backend.

While working with Gerust was generally enjoyable, there were some issues in Gerust itself that were addressed during this Bachelor's thesis. When initially setting up Gerust, the code it generated failed the more strict linting checks of `clippy::pedantic`, described in [Section 5.1.7.1](#). Some of the documentation was also unclear and required clarifications. Once the team had resolved these issues, the resulting patches were submitted as pull requests to the Gerust GitHub repository, most of which were merged by the Gerust team [63].

Overall, the backend implementation involves separating data modelling, persistence and web API logic. Gerust provides the workspace structure and common backend infrastructure. The `db` crate defines the database schema and entity access functions, and the `web` crate exposes these functions via HTTP endpoints for importing and exporting telemetry data.

5 Quality Assurance

5.1 Guidelines

The guidelines define which workflows and evaluation criteria have been applied while working on this project.

5.1.1 Version Control

For version control, this project uses Git with GitLab as the remote repository. All code and documentation contributions to the project must be managed with Git and pushed to the GitLab repository. In order to provide a clear overview of where and what changes are caused by a commit within the project, commit messages must adhere to the “Conventional Commits” standard [64]. If necessary, a multi-line commit message can be used to provide further information about the change or to note known problems.

The project comprises of three different repositories:

- **App:** The mobile application used for controlling the medication cooler
- **Backend:** The web service used for collecting telemetric data from users
- **Documentation:** The Bachelor’s thesis submitted at the end of the project

5.1.2 Git Strategy

For the app and documentation repositories, a “Git Feature Branch” strategy with rebasing and squashing is used [65]. This workflow was chosen because there will not be regular releases or deployments in these repositories, making a separate `develop` branch, as used in “Git Flow” [66], for example, superfluous.

For every change to the code or documentation, a new branch is created from `main`, the “feature branch”. Commits are only made on this branch. Once the changes are ready for review according to the criteria in [Section 5.1.4](#), a merge request is created and the other person is added as a reviewer. The reviewer then performs a code review and checks whether the criteria in [Section 5.1.5](#) are met.

If the reviewer approves the change, the MR can be merged into the `main` branch. Before doing so, the branch to be merged should be rebased onto `main`. This ensures that only fast-forward merges take place in the Git history. To force rebases, the “Fast-forward merge” option in all GitLab repositories of the project was enabled. This does not allow a merge if no rebase has been performed beforehand and also foregoes a merge commit that would clutter the history of `main`. Rebasing means that the Git history does not always correspond to reality, but it is extremely linear, clear, traceable and always moves forward. In addition, merge conflicts can already be resolved during rebasing.

When merging, the commits should be squashed. This also serves to clarify the Git history on the `main` branch: Since a large number of commits are often created within an MR, the Git history can quickly become confusing without squashing. When squashing, all commits in an MR are merged into a single commit. To do this, the “Encourage squash commits when merging” option in GitLab has been activated so that squashing is automatically enabled for every MR. After the merge, the exact commit history can still be viewed within the corresponding MR when required.

5.1.3 Issues

All user stories and tasks are tracked via a separate GitLab issue. The issue contains all important information about the task or user stories in its description (e.g. a checklist with all necessary steps). Comments may be used to discuss the issue and provide updates. Each issue is assigned a categorisation and an Epic label. The categories refer to the type of the task (e.g. documentation, CI/CD, feature) and the Epic label which Epic the issue belongs to, if applicable.

To start working on a new task, the feature branch should be preferably be created through the “Create Branch” button in the respective issue, in order to link the feature branch to the issue within GitLab.

5.1.4 Definition of Ready (DoR)

Once a task is done, according to the criteria listed below, a merge request is created from the feature branch to merge the changes into the main branch. The merge request is assigned to the person working the task and the other person is placed as the reviewer. The description of the MR should also contain the issue ID connected to this task to improve visibility of the matching issues and MRs. If an issue is closed with a merge, a closing pattern can be used to close the issue automatically (e.g. “Closes #1”).

For documentation changes, only the first two criteria must be met.

- ☐ All tasks associated with the issue are finished
- ☐ The corresponding issue has been linked in the MR description and contains a closing pattern, if appropriate
- ☐ Tests for the new functionality have been written
- ☐ All tests pass, including newly written ones

If changes are to be reviewed before these criteria are met, the merge request should be created as a draft. Once the criteria have been met, the merge request can be marked as “ready” and then follows the normal procedure.

5.1.5 Definition of Done (DoD)

The reviewer of the merge request is responsible for validating that the following criteria have been met. The reviewer can then approve and merge the MR.

DoD on Documentation Changes

- ☐ The changes in the source branch address the tasks mentioned in the issue
- ☐ The new content is clearly structured and linguistically correct
- ☐ The source branch has been rebased on the target branch after the guidelines in [Section 5.1.2](#)
- ☐ The merge request pipeline completed successfully (documentation compiles)

DoD in Development

- ☐ The changes in the source branch address the tasks mentioned in the issue
- ☐ Short manual tests of the changes show the expected behaviour and no misbehaviour
- ☐ The code quality meets the guidelines in [Section 5.1.7](#)
- ☐ The test coverage decreases by a maximum of 3%
- ☐ The source branch has been rebased on the target branch after the guidelines in [Section 5.1.2](#)
- ☐ The merge request pipeline completed successfully (program compiles, tests run successfully)

The checklists were stored in GitLab as merge request templates so that they are visible for every merge request and the items can be ticked off. Successful completion of the CI/CD pipeline in the merge request is automatically checked by GitLab, as the “Pipelines must succeed” option has been activated for all repositories. For this reason, the item is not listed separately in the merge request template.

5.1.6 Time Tracking

The time tracking is done through GitLab's time tracking feature [67]. The person assigned to the issue logs their time onto the issue, while the reviewer of the resulting MR tracks it there. This method enables a visualisation of the time spent implementing a change compared to reviewing it.

The time tracking data will be processed by GitLab Time-Report [68], a program that downloads the data through the GitLab API and creates charts and a dashboard from the time tracking data. The process is explained in more detail in [Section 5.3.1](#).

5.1.7 Code Guidelines

To ensure a clean architecture, both the App and Backend have been developed in accordance with the SOLID principles [69].

Single Responsibility Principle

Each class or module should have exactly one responsibility. In Flutter, this is achieved by separating the User Interface (UI) and data management code. Flutter recommends the use of the Model-View-ViewModel (MVVM) pattern [34]. This architecture divides each feature of the application into three components: the model, which defines the data structures; the view, which represents the UI; and the view model, which exposes the data required by the view and contains most of the application logic. This separation ensures compliance with the Single Responsibility Principle.

In Rust, and more specifically when using Axum, no standard architecture is recommended. Therefore, the Single Responsibility Principle is applied more directly. Each independent functionality is organised into its own module, containing only methods related to that specific concern. If a module grows too large, individual parts are further divided into submodules. Functions within these modules are designed to perform a single, well-defined task.

Open-Closed Principle

The Open-Closed Principle states that the source code of classes should not need to be modified when their behaviors is extended. In Flutter, this principle is widely applied, as all UI widgets extend one of the Flutter widget classes such as `StatelessWidget` or `StatefulWidget` by inheriting from them [70]. Widgets can also be composed by wrapping them around other widgets to modify them. Existing widgets should therefore be reused through inheritance or composition, although the latter should be favoured due to its looser coupling [71].

For non-UI Dart code, interfaces provide an additional mechanism [72]. Classes can interact with any implementation of a given interface without requiring modifications, thereby supporting extensibility.

In Rust, functions, structs and their fields are private by default, preventing external modifications [73]. Methods are designed to minimize the need for changes elsewhere when extending functionality. However, the use of `match` statements which need to cover every possibility can make strict adherence to this principle more difficult. Nevertheless, it is the opinion of the team that the advantages outweigh the disadvantages, and as such, violations of this principle are accepted in this case.

The Open-Closed Principle can also be solved in Rust with `dyn` traits to enable dynamic dispatch [74] or through generics and trait implementation for static dispatch [75], [76].

```

1 trait Behavior {
2     // Define a function to be callable on all structs implementing 'Behavior'
3     fn solve_problem(&self) → i32;
4 }
5
6 struct Context {
7     // Define a container in which all structs implementing 'Behavior' can be placed
8     pub behavior: Box<dyn Behavior>,
9 }
10
11 impl Context {
12     pub fn some_calculation(&self) → i32 {
13         // Calling functions of different types through static dispatch
14         self.behavior.solve_problem() + 1
15     }
16 }
17
18 mod a_solution {
19     use super::Behavior;
20     pub struct Foo;
21     impl Behavior for Foo {
22         fn solve_problem(&self) → i32 {
23             41
24         }
25     }
26 }
27
28 mod b_solution {
29     use super::Behavior;
30     pub struct Bar;
31     impl Behavior for Bar {
32         fn solve_problem(&self) → i32 {
33             44
34         }
35     }
36 }
37
38 fn main() {
39     let mut ctx = Context {
40         behavior: Box::new(a_solution::Foo),
41     };
42     println!("{}", ctx.some_calculation());
43     // Changing types at runtime with dynamic dispatch
44     ctx.behavior = Box::new(b_solution::Bar);
45     println!("{}", ctx.some_calculation());
46 }

```

Listing 22: Demonstrating static and dynamic dispatch in Rust

As seen in [Listing 22](#), static dispatch can be used to call a trait function of any struct implementing that trait, while dynamic dispatch serves to swap the type of the element with any other struct implementing that trait. Both make changes to the underlying struct unnecessary when another struct implements the same trait, thus satisfying the Open-Closed Principle.

Liskov Substitution Principle

In Dart, the Liskov Substitution Principle is realised through inheritance and interfaces. A base class can substitute for all its child classes and any class implementing an interface can be substituted by any other implementation of the same interface. Although inheritance is widely used for widgets, interfaces are the preferred approach for data-related components.

In contrast, Rust does not offer inheritance [77]. Instead, generic functionality is achieved through traits, which provide functionality similar to interfaces.

Both interfaces and traits can offer any type of functionality, but to adhere to the Liskov Substitution Principle, the behaviors of the implementing elements should have similar results within the same interface or trait, i.e. their behaviors should not be surprising to the user. This is similar to the informal “Laws” of Type Classes in the Haskell language, where types can be syntactically correct, but can lead to unexpected behaviors if the type does not adhere to these laws [78].

Interface Segregation Principle

The number of methods offered by interfaces and traits should be kept to a minimum and include only the functionality required by all implementers. Rather than imposing a strict limit on the number of functions an interface or trait is allowed to offer, the team has decided on a more practical approach: if a method cannot be sensibly implemented (e.g. requiring the use of the `unimplemented!()` macro [79]), the interface or trait should be divided into smaller, more specific abstractions.

Dependency Inversion Principle

Classes that are dependent on other classes or external resources, such as network access or file systems, should rely on abstractions in the form of interfaces or traits. Dependencies are not initialised within the class itself; rather, they are instead passed as a parameter. This approach promotes loose coupling and enhances testability, as dependencies can be substituted with mock objects in unit tests.

5.1.7.1 Linter

To enable automatic verification of the code, linters for both Dart and Rust code have been configured in their respective IDEs and CI/CD pipelines (see [Section 5.3](#)). These tools perform static analysis of the code in order to detect potential issues. To ensure code quality, all linter pipeline jobs must run without reporting any problems.

Dart

The app code is analysed using `analyzer`, the linter library developed by the Dart team [80]. It is invoked through the Flutter CLI with `flutter analyze`. Dart and Flutter provide three lint sets, which are included in the `lints` and `flutter_lints` packages [81], [82], [83]:

- `core`: Lints for critical issues likely to cause problems. All code should pass these lints.
- `recommended`: Lints that help identify additional issues that may lead to problems when running or consuming Dart code, and well as lints that enforce a consistent, idiomatic style and format. All code is encouraged to pass these lints. The recommended lints include all core lints.
- `flutter_lints`: Extends the recommended Dart lints with Flutter-specific rules.

When creating a new Flutter application, the project is automatically configured with `flutter_lints` [83]. While these lints provide a solid default, they remain relatively minimal. The Dart language itself offers a much more extensive list of rules, many of which are not included in the linter sets listed above [84]. Although it is possible to copy the complete list of lints into a Dart project, this is not recommended, as some rules conflict with each other, are experimental, or are known to produce false positives [85].

To enable most of these additional lints without having to manually exclude problematic ones, the `very_good_analysis` package is used [86]. In conjunction with `flutter_lints`, it provides a comprehensive set of rules to ensure code quality. However, two lints were disabled due to personal preference: `always_use_package_imports` and `public_member_api_docs`. The former enforces the use of the `package: prefix` in imports, even for files in the same project [87]. To make internal and external imports easier to distinguish visually, this rule was deactivated. The latter requires all public members to be documented [88]. While documenting most public members is generally beneficial, it is not necessary in every case, especially since the Flutter app does not offer a API to consume. Therefore, this rule was also disabled, allowing developers to decide individually which members warrant documentation.

Additionally, the `lines_longer_than_80_chars` lint was disabled which emits a warning when a line exceeds 80 characters [89]. In most cases, the line width is already enforced through the formatter, thus rendering the lint redundant, with the exception of long string literals. The decision was made to allow certain lines of code to exceed the length limit to enhance readability. For example, this was done in switch statements where only some of the cases were too long. In such cases, both the formatter and the lint rule had to be disabled for the designated area of code. To avoid this redundancy, the lint rule was disabled across the entire project.

Rust

The backend code is analysed using the `clippy` linter, which is included in the Rust toolchain [90]. Clippy divides its lints into several categories, such as `clippy::correctness` for code that is incorrect or useless, `clippy::style` for code that should be written in a more idiomatic way, or `clippy::performance` for code that can be optimised for better performance. These lints already prevent many common code issues.

As with Dart, Clippy also provides more comprehensive lint sets that are not enabled by default. The team enabled the `clippy::pedantic` rule set [91], which checks the code for additional issues, such as values that could be passed as references, possible casting problems, or `if-not-else` statements.

5.1.7.2 Formatter

In addition to the linters, formatters were also utilised within the project. Formatters ensure that the code is formatted consistently and adheres to the configured formatting rules. The team set up formatters for Dart, Rust and Typst code in their respective IDEs and CI/CD pipelines (see [Section 5.3](#)). If the code does not match the expected formatter output, the pipeline fails.

The Dart code is formatted using the `dart_style` formatter, developed by the Dart team [92]. It follows the conventions defined in the Dart Style Guide [93] and is invoked through the Dart CLI with `dart format`. Although a number of configuration options are available to adjust the formatting style, the team is satisfied with the default settings.

For the backend, the `rustfmt` formatter included in the Rust toolchain is used [94]. It formats the code according to the Rust Style Guide [95]. In RustRover, it is executed automatically on every commit, and it is also run in the CI/CD pipeline, where a Diff of the current and expected code is shown on failure. As with `dart_style`, `rustfmt` provides several customisation options for its rules, but all of them have been left unchanged.

The Typst code written for the Bachelor's thesis is also formatted. For this purpose, the `typstyle` linter integrated in the VS Code `tinymist` extension was selected [96], [97]. It formats not only Typst code itself, but also the regular text contained within it. In both Tinymist and the CI/CD pipeline, automatic line breaking has been enabled, so that all lines of text are formatted to a maximum length of 120 characters. This helps to keep diffs manageable and avoids the need for horizontal scrolling on GitLab, where the code preview does not support line wrapping. However, this approach has limitations. A textual change at the beginning of a paragraph can shift words later in the same paragraph onto the next line. A more effective solution would be to place each sentence on a separate line. The team has decided against this approach, as there is currently no convenient option to format Typst code in this manner automatically. At present, an issue requesting this feature is open in the `typstyle` GitHub repository [98].

5.1.8 Documentation Guidelines

There are two different types of documentation used within the project: The Bachelor's thesis paper and the code documentation within the source code files.

5.1.8.1 Bachelor's Thesis Paper

The papers structure follows the guidelines set by the "Orientierungshilfe" and "Leitfaden für SA/BA" documents provided by OST [99], [100]. The chapters and their content are based on the SE Project LaTeX template [101]. The paper itself is written in Typst with a modified template created by Dario Glasl [102]. All content must adhere to the guidelines in the documents referenced above and be verified during the DoR and DoD procedures (see [Section 5.1.4](#), [Section 5.1.5](#)).

In addition to the `typstyle` linter as mentioned in [Section 5.1.7.2](#), style guidelines which the linter does not verify have been defined. Typst uses labels such as `<project-plan-image>` to attach a name to elements like headings and figures [103], which can then be referenced by using the `@` symbol: `@project-plan-image` [104]. Bibliographic references also have names and can be referenced in the same way as labels. To visually distinguish between these two types, the decision was made to write the names of document references in kebab case and the name of bibliography references in lower camel case [105].

5.1.8.2 Code Documentation

Both Dart and Rust support the use of documentation comments to annotate functions, classes and members. These comments can be used to describe the functionality and intent behind an element. They can be exported to a HTML reference of the code and serve as a manual for the codebase. Documentation comments also support Markdown, which can be used to include semantic elements, such as headings, tables or code snippets with proper syntax highlighting. In Rust specifically, these code snippets are compiled and run during test execution. These so-called "Doc-tests" verify that the code examples in the documentation continue to work with the actual code over time.

The team has created documentation comments for most functions, classes and members, providing at least a brief description of their purpose. When necessary, additional information such as usage instructions or parameter explanations are also included. For both the app and backend, a HTML reference is generated within their respective pipelines, as described in [Section 5.3](#).

5.2 Testing

To guarantee the functionality of the code to the best extent possible, different types of code testing are employed. The app uses unit and widget tests, while unit and API tests are used for the backend.

5.2.1 Unit Tests

Unit tests verify an application on the individual function level. They verify that a function produces the expected output from a given input and validate eventual side effects. Unit tests are the most granular type of code test.

Classes and modules often have external dependencies. To test a unit in isolation, these external components are mocked. All Dart and Rust code dealing with external components apply the Dependency Injection pattern [37], thus conforming to the dependency inversion principle described in [Section 5.1.7](#). With Dependency Injection, dependencies are not instantiated within the class itself, but are passed at runtime to the constructor of a class. In unit tests, these dependencies can be substituted with mock versions that are configured to always return a specific result. This approach enables modularity and the independent testing of each module.

5.2.1.1 App

In Flutter, unit tests are created using the `flutter_test` package [106]. The test runner is configured to locate unit tests in the `test` directory in the project root, in files ending with `_test.dart` [107]. It is further recommended that the structure of the `test` directory be aligned with that of `lib`, which contains the primary code.

Each test file consists of a main function that contains the test setup and the tests themselves. Optionally, it can also contain a `tearDown` function. A test is created using the `test()` function provided by `flutter_test`, which contains the test name and code. The test library also provides assertions via the `expect()` function. It takes a property, such as the return value of the tested function, and compares it to a matcher [107]. While the matcher can be a simple constant such as a number, more specialised matchers are usually used to provide clearer error messages about the expected state when a test fails, as shown in [Listing 23](#).

```
1 test('Raw number as matcher', () {
2   final list = List<int>.empty();
3   expect(list.length, 2);
4 });
```

Raw number as matcher [E]
Expected: <2>
Actual: <0>

```
1 test('Dedicated matcher', () {
2   final list = List<int>.empty();
3   expect(list, hasLength(2));
4 });
```

Dedicated matcher [E]
Expected: an object with length of <2>
Actual: []
Which: has length of <0>

Listing 23: Difference in error output when using a raw value versus a dedicated matcher in Flutter tests

Mocking functionality is provided through the `mockito` package [108]. To add mocks to a test, the `main()` function in the test files must have the `@GenerateNiceMocks([MockSpec<ClassToMock>()])` annotation. This marks the respective class to be mocked and creates a default value for each of its members [109]. The mock class can then be generated using the `dart run build_runner build` command, which creates a `.mocks.dart` file next to the test file [108], [110]. The mocks can then be used in two different ways: Adding stubs to return specific values from the mocks and verifying that a specific function was called n-times in a mock, as demonstrated in [Listing 24](#).

```
1 // Generates a Mock for BluetoothService with default return values for all members
2 @GenerateNiceMocks([MockSpec<BluetoothService>()])
3 void main() {
4   late ConnectionRepository connectionRepository; // The real repository to test
5   late MockBluetoothService mockBluetoothService; // The mocked dependency
6   final device1 = DiscoveredDevice(name: 'Test Cooler', /* ... */);
7   setUp(() { // Code that is run before all unit and widget tests in this file
8     mockBluetoothService = MockBluetoothService();
9     connectionRepository = ConnectionRepository(bluetoothService: mockBluetoothService);
10  });
11
12  test('calls checkPermissions and starts scan', () async {
13    // Create a stub for mocked service: OS is ready for Bluetooth connections
14    when(mockBluetoothService.statusStream).thenAnswer((_) => Stream.value(BleStatus.ready));
15
16    // Call the function to test
17    await connectionRepository.startScanning();
18
19    // Check that these functions from the Bluetooth service have been called exactly once
20    verify(mockBluetoothService.checkDeviceBluetoothPermissions()).called(1);
21    verify(mockBluetoothService.startScan()).called(1);
22
23    // Check that the added device is returned by the repository
24    fakeScanController.add(device1);
25    await expectLater(connectionRepository.discoveredDevices, emits([device1]));
26  });
27 }
```

Listing 24: Unit test from `ConnectionRepository` showcasing Mockito mocks and stubs

A total of 169 unit tests have been written for the app. The app's total test coverage is discussed in [Section 6.2.1](#).

5.2.1.2 Backend

By convention, unit tests in Rust are written in the same file as the main code. For this purpose, a new module is created, typically named `tests`. This module is annotated with `#[cfg(test)]`, meaning it is only compiled when running unit tests and not when creating debug or release builds [111].

Since the unit tests are located in a separate module, the main code's functions are not directly accessible. Instead, all symbols of the parent module are brought into scope using the `use super::*` statement. Because `tests` is a submodule of the module to be tested, it can also access private members. Therefore, unlike in Dart, it is possible to write unit tests for private functions [112].

Individual test functions are marked with the `#[test]` annotation. The results of a test can be validated using the macros of the `assert!()` family: `assert!()` evaluates a Boolean condition [113], while `assert_eq!()` and `assert_ne!()` test for equality or inequality [114], [115]. If an assert is not fulfilled, a panic is triggered and the test fails.

Since most of the backend code was generated by Gerust rather than written by the team, the Gerust code was treated like any other dependency. It is not the team's responsibility to test the functionality of external code. Consequently, the team decided to exclude crates not intended for modification, such as the `macros` or `cli` crate, from the code coverage report using the `#![cfg(not(tarpaulin_include))]` annotation.

This left the `web` and `db` crates to be tested. Due to the predominantly straightforward nature of the code and the time constraints, it was not deemed viable to write unit tests for the controllers or entity CRUD functions. Instead, it was decided to limit the scope to only write API tests, with the understanding that this would cover both cases.

5.2.2 Widget Tests

Widget tests, known as component tests in other UI frameworks, are a type of test aimed at verifying the UI [116]. They build up a headless version of the UI in a minimal application where only the widget under test is created, enabling the appearance of the UI and interactions with the widget to be tested. Widget tests are treated the same as regular unit tests, except they are created using the `testWidget()` function and have a specialised `tester` object that can be used to manipulate widgets [117].

As pages and views are merely collections of widgets, widget tests can be used to test these as well, thus serving as UI tests for the app. In [Listing 25](#), a widget test for the Home Page is shown. As its functionality depends solely on the `HomeViewModel`, this is the only component that needs to be mocked. The mock is initialised in the `setUp` function, which runs before every unit and widget test. To simplify initialisation, the process of creating the widget with the root `MaterialApp` widget has been extracted into a separate function. This function returns a widget, which is then created using `tester.pumpWidget()`, which builds the widget without creating the actual graphics of the UI. A finder is then used to locate the temperature text inside a `TemperatureRange` widget. Finally, it is asserted that the finder has found exactly one widget that matches this condition.

```

1  @GenerateNiceMocks([MockSpec<HomeViewModel>()])
2  void main() {
3    late MockHomeViewModel mockViewModel;
4
5    setUp(() {
6      // Initialise mocked view model
7      mockViewModel = MockHomeViewModel();
8    });
9
10   // Helper function to wrap the widget into a minimal app
11   Widget createWidgetUnderTest({Key? key}) {
12     // Default root widget in the Android "Material" theme
13     return MaterialApp(
14       localizationsDelegates: AppLocalizations.localizationsDelegates,
15       supportedLocales: AppLocalizations.supportedLocales,
16       theme: AppTheme.lightTheme,
17       home: ChangeNotifierProvider<HomeViewModel>.value(
18         value: mockViewModel,
19         child: HomePage(key: key), // Create the HomePage widget
20       ),
21     );
22   }
23
24   testWidgets('Shows temperature when data is available', (tester) async {
25     const temperature = 5.1;
26     // Set return value of mocked function
27     when(mockViewModel.currentTemperature).thenReturn(temperature);
28
29     // Create HomePage widget
30     await tester.pumpWidget(createWidgetUnderTest());
31     // Wait until widget is loaded and animations are completed
32     await tester.pumpAndSettle();
33
34     // Find the temperature text inside a "TemperatureRange" widget
35     final findConnectedTemperatureRange = find.descendant(
36       of: find.byType(TemperatureRange),
37       matching: find.text('$temperature°C'),
38     );
39     // Verify that exactly one widget was found
40     expect(findConnectedTemperatureRange, findsOneWidget);
41   });
42 }

```

Listing 25: Example Widget Test of the HomePage widget

Widget tests ensure that the UI elements react correctly to state changes, such as incoming data from View Models and user input. The project contains a total of 76 widget tests. Since the Flutter test runner treats widget tests the same as unit tests, they have been included in the unit tests' code coverage number given in [Section 6.2.1](#).

5.2.3 Integration Tests

Integration tests run the application from start to finish, verifying that the entire code ran produced the expected output from the received input. They initialise the application with a particular environment setup, such as command line arguments, environment variables or files, and then verify the output once the program has halted.

5.2.3.1 App

While Flutter recommends integration tests as the third type of tests that provides the highest level of confidence in the code, they were not included in this project. This is because the app's core functionality depends on Bluetooth Low Energy communication, which cannot be tested in an emulator or a service such as Google's "Firebase Test Lab" [118].

Therefore, integration tests would require a physical Android or iOS device, as well as a BLE simulator to expose the cooler's services and characteristics. Setting up and maintaining this test environment was deemed too complex within the scope of this project [116], [119].

5.2.3.2 Backend

Since the backend is a web server that waits for connections and only terminates after manual intervention, it is not sensible to write traditional integration tests for it. There are no command line arguments to test and the reading of environment variables is already covered by the Gerust unit tests. However, the API tests of the web crate are set up as Rust integration tests in a separate test directory [120]. Similar to Flutter tests, they revolve around a context that provides access to the web API and the database. In [Listing 26](#), the `/latest_counter` route is tested by first inserting sample data into the database and then calling the API with the context object.

```
1  #[db_test]
2  async fn test_get_latest_counter_with_cooler_present(context: &DbTestContext) {
3      const MAX_COUNTER: i64 = 20;
4
5      // Get existing property
6      let property = entities::telemetry_properties::load_by_name(
7          "Current Temperature".to_string(),
8          &context.db_pool,
9      )
10     .await
11     .unwrap();
12
13     // Create a new cooler
14     let mut cooler_changeset: CoolerChangeset = Faker.fake();
15     cooler_changeset.serial_number = SERIAL_NUMBER.to_string();
16     let cooler = entities::coolers::create(cooler_changeset.clone(), &context.db_pool)
17         .await
18         .unwrap();
19
20     // Create readings for the created cooler
21     for counter in 0..=MAX_COUNTER {
22         let mut telemetry_reading: TelemetryReadingChangeset = Faker.fake();
23         telemetry_reading.log_counter = counter;
24         telemetry_reading.cooler_id = cooler.id;
25         telemetry_reading.property_id = property.id;
26
27         entities::telemetry_readings::create(telemetry_reading.clone(), &context.db_pool)
28             .await
29             .unwrap();
30     }
31
32     // Send a GET request to the API
33     let response = context
34         .app
35         .request(&format!("/latest_counter?serial_number={SERIAL_NUMBER}"))
36         .method(Method::GET)
37         .send()
38         .await;
39
40     // Verify that the response matches the expected counter
41     assert_eq!(response.status(), StatusCode::OK);
42     assert_eq!(response.into_body().into_json::<i64>().await, MAX_COUNTER);
43 }
```

Listing 26: An API test of the `/latest_counter` route in the backend

Test data can be created using the `Faker.fake()` function through the use of the `fake` crate [121]. As demonstrated in [Listing 27](#), the changesets to be inserted into the database can be annotated with generators provided by the library. These are wrapped in a `#[cfg_attr(feature = "test-helpers")]` annotation to ensure that the code for fake data is only generated during test runs. Additionally, it is also possible to validate real and test data coming from the API by adding a `#[validate]` annotation [62], [122].

```

1 // Limit imports to only when the "test-helpers" feature is enabled
2 #[cfg(feature = "test-helpers")]
3 use fake::{faker::chrono::en::DateTime, uuid::Uuidv4, Dummy};
4
5
6 // The derived functions on the struct
7 #[derive(Deserialize, Validate, Clone)]
8 // Limiting derives to only when the "test-helpers" feature is enabled.
9 #[cfg_attr(feature = "test-helpers", derive(Serialize, Dummy))]
10 pub struct TelemetryReadingChangeset {
11
12     // Generates a random UUID, has to be overwritten in tests with a valid foreign key
13     #[cfg_attr(feature = "test-helpers", dummy(faker = "Uuidv4"))]
14     pub cooler_id: Uuid,
15
16     #[cfg_attr(feature = "test-helpers", dummy(faker = "Uuidv4"))]
17     pub property_id: Uuid,
18
19     // Generates a JSON value with a simple key-value pair
20     #[cfg_attr(
21         feature = "test-helpers",
22         dummy(expr = "serde_json::json!({\"key\": \"value\"})")
23     )]
24     pub value: JsonValue,
25
26     // Generates a random Date and Time
27     #[cfg_attr(feature = "test-helpers", dummy(faker = "DateTime()"))]
28     pub recorded_at: NaiveDateTime,
29
30     // Generates a random number larger than 0 for the test cases
31     #[cfg_attr(feature = "test-helpers", dummy(faker = "0.."))]
32     // Validates that data from the API is always > 0
33     #[validate(range(min = 0))]
34     pub log_counter: i64,
35 }

```

Listing 27: The changeset for the `TelemetryReadings` database table with test data generator and validation attributes

A running database is required for the tests to execute. Gerust provides a separate test database for this purpose in the Docker Compose file. Once the container has been started, any required database migrations must be run and seeds inserted via the Gerust CLI. The tests can then be started with `cargo test`. When running the tests locally, the `DATABASE_URL` environment variable must be set to the connection string of the running test database, otherwise the tests will not compile.

In total, the team has created 5 API tests for the backend. More about the test coverage can be read in [Section 6.2.1](#).

5.2.4 Manual Testing

Since the Android emulator and web target used by the team in the early stages of development did not support Bluetooth, the team had to use their own Android phones to test the app once the Bluetooth functionality had been implemented. Consequently, all code was manually tested on real Android devices.

The team was able to compile the app for iOS for the first time in sprint 6 using a Mac provided by OST and an iPhone XR provided by the industry partner. However, as neither team member was familiar with the MacOS development workflow, setting up and compiling the app took over an hour. The team could not figure out how to compile the Flutter app from inside Xcode, so they had to use the Flutter SDK instead. Additionally, some dependencies relied on the CocoaPods package manager [123], which also required adjustments to run correctly on the Mac.

Once the setup was complete, the team had to adjust the iOS build files due to a syntax error introduced when setting up the name and permissions of the app in a previous change. Following this adjustment, the app finally compiled and launched on the iPhone. However, when the BLE simulator was connected, no data was displayed on the home screen.

The team eventually discovered that the iOS Bluetooth stack experienced issues when a large number of Bluetooth actions were executed in quick succession. In particular, the connection process must be completed in full before attempting to subscribe to characteristics, otherwise the subscription attempts will fail. After adding various delays between BLE actions, the data was transmitted and the app worked the same way as it did on Android.



Figure 26: Image of the app running on an iPhone XR

5.2.5 Usability Testing

In order to guarantee that the app fulfils the established usability requirements, two tests were conducted: one with participants from the “app user” target audience, and one evaluation by a UX expert.

5.2.5.1 Testing Procedure First Usability Test

The testing is conducted individually with each subject and is divided into several sections [124].

- **Introduction:** This step establishes the context and purpose of the usability testing session. It explains the purpose, procedure and duration of the test. It also reassures participants that the product, not the participant, is being evaluated.
- **Warm-up Questions:** In this stage, simple background questions about the participant are asked to provide context about the users experience and habits.
- **Task Walkthrough:** During this phase, the participants are asked to complete predefined scenarios using the app while talking about their approaches and problems. Their actions are observed to identify usability issues and interaction patterns.
- **Follow-up Questions:** After the tasks are completed, the subject is asked to answer questions based on the SUS statements about the app and the experience of using it. These questions are used to generate the SUS-Score [11], [125].
- **Conclusion:** This closes the testing session by thanking the participant and inviting final comments.

The wireframes used in the first testing can be found in [Section 3.6.1](#), while the full questionnaire and results are available in [Section B](#) and [Section 6.2.2](#).

5.2.5.2 Testing Procedure Second Usability Test: Expert Evaluation

The expert evaluation is conducted individually with one UX expert and follows a similar structure to the first usability test.

- **Introduction:** This step establishes the context and purpose of the expert evaluation. It presents the current state of the project, explains the purpose and procedure of the evaluation and clarifies which parts of the app are ready for evaluation.
- **Task Walkthrough:** During this phase, the expert is asked to walk through the predefined scenarios from the first usability test that can be completed with the current state of the app. The interaction is observed by the team to identify usability issues, inconsistencies and possible improvements.
- **Discussion:** After the task walkthrough, a general discussion about the usability of the app is conducted. This enables the expert to provide additional feedback, identify potential issues and propose improvements.
- **Follow-up Questions:** Following the discussion, the expert is asked to answer questions based on the SUS statements about the app and the experience of using it. These questions are used to generate the SUS-Score, as seen in the first usability testing.
- **Conclusion:** This closes the evaluation by thanking the expert and inviting final comments.

The results of the expert evaluation are available in [Section B](#) and [Section 6.2.2](#).

5.3 CI/CD

A CI/CD pipeline is essential for quality assurance, as it can run various verifications and automations on the code. The entire project utilises GitLab CI/CD pipelines [126]. They run whenever there is a push to the GitLab repository and notify the maintainers if the compilation or tests fail.

Each of the three repositories has its own pipeline configuration. A pipeline consists of jobs, each of which perform a specific task. These jobs are then grouped into stages. Jobs within a stage can run in parallel, but those in subsequent stages will only run once all jobs of the previous stage have completed successfully [126].

To reuse job configurations, templates can be created. Their name usually starts with a period to hide them from the job list and prevent them from being executed automatically. Other jobs can use the template by specifying it with the `extends` keyword [127].

5.3.1 Documentation Pipeline

The pipeline in the documentation repository has the stages “lint”, “build” and “deploy”.

5.3.1.1 Lint Jobs

- **lint:typstyle:** The team utilises the Visual Studio Code IDE for writing Typst code, which is configured to automatically install and run the Typst linter “typstyle”, as described in [Section 5.1.7.2](#). This job runs it again to verify that the linter has been executed correctly. The IDE has been observed to be unreliable when applying formatting, so executing typstyle again in the pipeline ensures that the formatting of the Typst files is consistent. When the pushed code does not match the linter’s output, the job will fail and a diff is printed in the job log.

5.3.1.2 Build Jobs

- **build:gitlab-time-report:** In order to process the time tracking data stored in the GitLab issues and merge requests, the job downloads GitLab Time-Report [68]. The program then fetches the time logs from all three repositories and generates charts and a dashboard showing the team’s working hours. This only occurs when pushing into the `main` branch, as it is not needed on feature branches.
- **build:documentation:** This job compiles this Bachelor’s thesis with the Typst compiler. If the `gitlab-time-report` job has been run, the artifacts from that job are downloaded and the placeholder images used for local compilation are overwritten with the current time tracking charts. The HTML dashboard is also included in the paper as a PDF attachment. The Typst code is then compiled to PDF, which is saved as an artifact.

The team has also defined three different build levels for the thesis in the Typst template. A level can be set with a manual run of the pipeline through the `DOCUMENT_BUILD` CI/CD variable: `draft` includes all content, `preview` (the default) hides comments and todos and `release` additionally excludes all sensitive content not meant for the public release of the paper.

- **build:presentations:** The interim and final presentation slides were also created with Typst. This job searches the `Presentations` directory for files ending in `presentation.typ` and compiles them into two variants: A handout and a presentation. The latter includes speaker notes and animations and is presented using the `pympress` program [128].

As with the thesis paper, the presentations also include the current short Git commit hash on the first page. This makes it easier to track which version of the codebase the PDFs were created from.

5.3.1.3 Deploy Jobs

- **deploy:pages:** This job deploys the documentation, the presentations and the time tracking dashboard on GitLab Pages on every push to the main branch. The links to these documents can be found in the README of the documentation repository. A more detailed description of the GitLab Pages deploy job can be found in [Section 5.3.4](#).

5.3.2 App Pipeline

The pipeline in the app repository consists of the stages “build”, “lint”, “test”, “doc” and “deploy”. The lint jobs are placed in a separate stage to avoid running the computationally expensive tests if the lint jobs fail.

5.3.2.1 Job Templates

- **.flutter-template:** Used by all jobs except the `deploy:pages` job. It specifies the Docker Image, the location of the Flutter dependency cache and the GitLab cache. Previously, the Docker image prebuilt by Cirrus Labs was used [129], but due their acquisition by OpenAI, the repository is no longer being updated with newer Flutter versions [130]. A fork of the repository has appeared, which provides the same images [129]. The team has successfully migrated to this fork.

In addition, the `PATH` environment variable is set to include the path to Flutter tools that jobs may install. Due to `PATH` being exported before being evaluated by the pipeline, it is not possible to set it in the `variables` section; it has to be manually set in the `before_script` section [131].

5.3.2.2 Build Jobs

- **build:debug:** This job builds the Flutter app as a debug build for the web. As it is one of the more simple targets to compile for, it has been chosen to verify if the code can be compiled.
- **build:android:** This job builds the Flutter app as a release build APK for Android. It only runs for commits on the `main` branch, as it should represent a release build for the customers.

A iOS build job has been omitted from the pipeline due to technical reasons: The necessary tooling to build iOS apps is only available on the MacOS-exclusive XCode IDE [19]. There are two ways to create a iOS build job. The recommended option is to set up a GitLab Runner with the shell executor on a MacOS device and install XCode on it [132]. This option was dismissed because the team does not have permanent access to a MacOS system.

The second option is to use third-party tools that allow XCode to be used within a Docker container, such as Docker-OSX [133]. However, using Docker-OSX was not a viable option due to the container requiring special Docker configuration not supported by GitLab CI/CD. Manual testing involving running the container on the teams’ machines also revealed that the container takes a long time to start up, install XCode and compile. This would have a significant impact on the runtime of the pipeline, even if the compatibility issue could be resolved. For the aforementioned reasons, it was decided not to proceed with an iOS build within the pipeline.

5.3.2.3 Lint Jobs

- **lint:analyze:** In this job, a basic static analysis using the `analyzer` package is run on the codebase [80]. This helps to prevent bugs and ensures the reliability of the code.
- **lint:format:** To have consistent formatting of the Dart files, the `dart format` tool is run [134]. If the code does not match the formatting rules, the offending files are listed and the job fails.

5.3.2.4 Test Jobs

- **test:** This job runs all Flutter unit and widget tests and generates a JUnit report with the “junitreport” package to display test results within the GitLab UI [135].
- **test:coverage:** In this job, the line coverage by all unit and widget tests is calculated and a Cobertura report is generated to display coverage in Merge Requests. An HTML report, where the coverage of each line can be viewed, is also created and added as an artifact. Several files where test coverage is unnecessary, such as localization, theming, generated files and the tests themselves, are excluded from the total test coverage score.

By default, Flutter only calculates code coverage for source code files that have been loaded during test execution. Consequently, files where no lines are tested do not decrease the code coverage percentage. In order to address this issue, the job utilises the “full_coverage” package, which generates a Dart unit test file that imports every other Dart file from the project [136].

5.3.2.5 Doc Jobs

- **doc:dart-documentation:** Dart is capable of generating an HTML documentation that lists all classes, functions, constructors and properties in the codebase, including documentation comments written by the developers directly in the code. This document can serve as a manual of the code base for the developers. The job only runs on the main branch, as generating the documentation in CI is not necessary during development on a feature branch.

5.3.2.6 Deploy Jobs

- **deploy:pages:** The HTML test coverage report and the code documentation are deployed on GitLab Pages. This only occurs when pushing to the main branch. A more detailed description of the GitLab Pages deploy job can be found in [Section 5.3.4](#).

5.3.3 Backend Pipeline

The pipeline in the backend repository consists of the stages “build”, “lint”, “test”, “doc” and “deploy”. Most jobs utilise the official Rust container in the version 1.94 [137]. This pipeline contains job templates for Rust jobs and for jobs requiring a running database.

The backend pipeline also features deployment jobs to roll out new versions onto a server. To this end, the team utilised a virtual machine on a server provided by OST for the duration of the Bachelor’s thesis. This virtual machine acts as the production server for deploying the backend. The setup script, `setup-deploy.sh`, sets up apt-based Linux distributions for deployment.

5.3.3.1 Job Templates

- **.rust-template:** This template defines the used container, the variables for the used tools and the caching settings (as described in [Section 5.3.3.8](#)). Prior to executing each job, the Cargo bin directory is added to the system PATH to ensure that installed tools are accessible. To install additional Rust tooling, `cargo-bininstall` is installed and used to download `sccache` and `crates` as specified by the `CARGO_CRATES` variable in each job [138]. Instead of the standard Cargo installation command, which results in the recompilation of the installed crates, `cargo-bininstall` downloads prebuilt binaries. This shortens the pipeline’s runtime considerably.
- **.database-template:** The database template adds a PostgreSQL container to be started alongside the jobs and switches the `SQLX_OFFLINE` variable to 0, which enables actual database queries [139]. The template reuses the `before_script` elements of the Rust template, so that required tools are also installed. In addition, it prepares the database by creating the necessary tables and applying all database migrations using Gerusts db tool.

5.3.3.2 Build Jobs

- **build:debug:** Compiles all the Rust code in the project using the test configuration. The advantage of this configuration over the default dev configuration is that it automatically checks whether the unit tests can be compiled as well.

5.3.3.3 Lint Jobs

- **lint:clippy:** This job runs the Clippy linter on the program code and on the unit tests [90]. The job fails if Clippy reports any errors or warnings.
- **lint:rustfmt:** This job defines a formatting check for the Rust codebase using `rustfmt` [94]. Its purpose is to ensure that all source code follows a consistent style. The job fails if `rustfmt` detects improperly formatted code.

5.3.3.4 Test Jobs

- **test:** This is the main test execution step of the pipeline. It uses both templates and produces structured reports for integration into GitLab tooling. The job runs all unit and integration tests in the project. If any tests fail, the job will be unsuccessful. The `cargo2junit` crate generates JUnit reports for all tests, regardless of their results, which are then processed by GitLab to display the test results in Merge Requests [140], [141].

For the generation of the reports it is necessary to use the unstable features `--format json` and `--report-time` in `cargo test`. These features can be enabled using the `-Z unstable-options` flag, which is only available in the nightly builds of the Rust compiler. To address this limitation, the `RUSTC_BOOTSTRAP=1` variable is set to provide a workaround. Although this approach is not recommended by the Rust developers [142], it is more efficient than downloading the entire nightly build of the compiler and it is also the workflow recommended by the author of the `cargo2junit` crate [140].

- **test:coverage:** This job involves generating code coverage reports with `cargo tarpaulin` to measure the extent to which the source code is covered by unit tests [143]. These reports are generated as an HTML file and as a Cobertura report. The former is deployed on GitLab Pages, while the latter is used to display the coverage in merge requests [141].

5.3.3.5 Doc Jobs

- **doc:rust-documentation:** This job utilises the `cargo doc` command to generate HTML pages from the documentation comments in the code, which serve as a manual for the functionality of the program. This job only runs on the `main` branch.

5.3.3.6 Deploy Jobs

- **deploy:pages:** The code coverage report and the Rust documentation are deployed on GitLab Pages on every push to the `main` branch. A more detailed description of the GitLab Pages deploy job can be found in [Section 5.3.4](#).
- **deploy:docker:** To easily deploy the backend, the application is packaged with Docker. The Dockerfile is based on the `debian:latest` image and creates a new user to run the backend. The backend gets copied into the container and the command to execute is set. The Docker CLI builds the container and uploads the result to the repositories GitLab Container Registry [144].
- **deploy:backend-prod:** This is the main deployment job which starts the backend. It sends the `compose.yml` file containing the Docker Compose configuration and all environment variables to the OST VM with SCP. Afterwards, an SSH connection is established to run commands on the VM: The script logs into the GitLab Container registry to download the container added in `deploy:docker` and starts the backend and database Docker container via the `compose.yml` file.

The configuration required to access the server via SSH is rather complex. GitLab recommends using a public-private key pair, which can be generated with `ssh-keygen` [145]. To use it within the pipeline, the keypair must be created without a password, as otherwise the job will prompt for it. Once generated, the private key needs to be added as a CI/CD file variable.

Inside the job, it is necessary to start the `ssh-agent`, which is responsible for handling SSH private keys. When running the agent normally, it outputs commands to set the necessary environment variables. These commands are not executed directly; rather, they need to be executed via `eval` [146]. The file containing the private key is then given the proper permissions and added to the agent store with `ssh-add`. Finally, the public key of the keypair needs to be placed into `~/.ssh/authorized_keys` to be able to connect with the server.

Unfortunately, even after these steps, the environment is still not functioning correctly. When attempting to connect to the server, a “Host key verification failed” error message appears. This is due to a SSH feature called “Strict Host Key Checking”, that only allows connections from SSH keys that have their public key listed in the `known_hosts` file on the server [147]. While it is simple to disable this check with the `-o StrictHostKeyChecking=no` option, this weakens protections against Man-in-the-Middle attacks. To enable connections with this protection turned on, it is necessary to run the `ssh-keyscan <DOMAIN-NAME>` command on the deployment server [148]. Its output lists all public SSH keys in `known_hosts` together with the specified domain name and port to connect to. This information has to be added to a GitLab CI/CD file variable. The job then moves the file to `~/.ssh/known_hosts` and grants it the necessary permissions. The SSH connection can now be established securely.

- **deploy:migrate:** The final step in the deployment process consists of applying any pending migrations to the production database. This is automated via the `db` CLI generated by Gerust. Once this is complete, the new version of the backend is ready to use.

5.3.3.7 Merge Request Jobs

For efficiency reasons, some jobs are configured to run only when their associated files are modified. However, this has caused issues when non-code files, such as the README or the pipeline configuration, have been changed. Since no job was configured to run, no pipeline was executed. Due to the configuration that merging a Merge Request is only possible after a successful pipeline run, merging became impossible. As a workaround, a dummy job was added.

- **dummy-merge-request-job:** Runs on every merge request event to ensure a pipeline is always generated. Executes an `echo` command.

5.3.3.8 Rust Caching

Compared to other languages, the Rust compiler takes a relatively long time to compile code. This is due to several factors such as the numerous compile-time checks (including borrow checking), and the fact that dependencies are downloaded as source code and must be compiled the first time they are used.

As most `cargo` commands used within the jobs trigger a recompilation, the compiler output is cached within the pipeline. This prevents dependencies from being downloaded again and enables the compilation to be completed incrementally.

In GitLab, both directories and files can be cached. Caching is controlled via a key: if the same key is used in multiple jobs, the same cache is used. Using the hash of the `Cargo.lock` file as the key, which maps all (transitive) dependencies and their version numbers, enables compilation to be cached across branches. The cache is only rebuilt when a (transitive) dependency is altered, and thus the cache key is updated.

Initially, the Rust build jobs used a basic GitLab cache for the `target` directory, where Rust stores all its build files. When working with Rust locally, only the changed crates need to be recompiled with the build data in that directory. This simple caching was therefore implemented to speed up build times. Unfortunately, the build cache was ignored in most jobs, and Cargo recompiled all dependencies regardless. The only time the caching helped was when the Rust source code was unmodified from the previous pipeline execution such as on manual pipeline runs. Additionally, the number of files in the cache continued to grow, slowing down downloading and unpacking at the start of a job, as well as packing and uploading the cache after the job had finished. This resulted in the cache operations taking several minutes, greatly exceeding the execution time of the job tasks themselves.

When searching for alternatives, the team discovered `sccache`, a tool developed by Mozilla and used for building Firefox [149]. Acting as a wrapper around the `rustc` compiler, it creates a single hash of all elements of the compilation, including source files, environment variables, compiler binary and arguments. `sccache` then checks whether the hash already exists in the cache; if so, it returns the already-compiled artifact. Otherwise, the artifact is compiled normally and then added to the cache [150].

Using `sccache` has decreased the build time of the `cargo db` crate from 21 seconds to 8 seconds and reduced the `deploy:migrate` runtime from 5 minutes to 53 seconds [151].

5.3.4 GitLab Pages Deployment

As can be seen in the chapters above, some of the artifacts from each pipeline are deployed to GitLab Pages, providing a straightforward method to view these documents [152]. However, each new GitLab Pages deployment removes the existing pages content. This means that only the files from the current pipeline are being deployed, while the old files from the last pipeline run are deleted.

This is an issue when some jobs, such as the Rust code coverage job, are only run when needed. Initially, the straightforward solution of executing all tasks on the `main` branch was implemented. This approach ensured that all artifacts were present in the Pages deployment job. While this approach is effective, it is inefficient as the jobs are executed even when the resulting artifacts are identical to those already deployed.

Subsequently, a different method was implemented: The Pages job downloads the artifacts from the last successful Pages job on `main`, which is equivalent to the files currently deployed on GitLab Pages. The job then verifies the presence of the artifacts from a previous stage. If found, the downloaded artifact is overwritten. The resulting files are then deployed. This approach, which is used in all three pipelines, is designed to guarantee that all artifacts are deployed while ensuring that the artifact-generating jobs are only executed as required.

5.4 External Code Reviews

To ensure the quality of the app's architecture, the team has requested an external code review with a focus on the MVVM implementation. As the team had no prior experience of implementing MVVM, the primary objective of the review was to obtain feedback regarding the adherence to MVVM practices and to receive clarifications for any architectural difficulties encountered.

Unfortunately, the supervisors did not know any Dart or Flutter experts to review the app, but instead offered a review by Clemens Meier from the Institute for Software at OST, who specialises in Frontend Architecture and Frameworks. The review was conducted shortly after the implementation of the MVVM layers in sprint 4. Following the review, certain issues were addressed immediately, while others were deliberately omitted. This chapter offers a comprehensive overview of both types of feedback.

The full review can be found in [Section C](#).

Feedback	Implementation
Introduce commands in the UI layers	Commands encapsulate interaction logic between the view and view model and run actions in the data layer. They allow to store and react to UI state, such as "loading", "complete" or "error". The team has implemented commands with the <code>command_it</code> package.
Remove direct dependencies from UI models into the data layer	<code>ui/models/user_message.dart</code> did import <code>data/models/cooler_status.dart</code> and defined a <code>CoolerStatus</code> getter. In a clean MVVM architecture, such a direct dependency from UI models to data models should not exist. As a mitigation, the <code>UiStatus</code> enum has been created. This can model not only the Bluetooth connection status of the cooler, but also other application states, such as "Bluetooth disabled on the device" and "Cooler out of temperature range". The mapping between <code>CoolerStatus</code> and <code>UiStatus</code> has been moved into <code>HomeViewModel</code> .
Improve error handling for BLE connection errors	The BLE service and repositories did not have proper error handling. Subscriptions now terminate on errors and clear their state. Failures in the Bluetooth service, such as failing to search for BLE devices because Bluetooth is disabled on the device, now throw errors that are handled in the <code>HomeViewModel</code> .
Missing dispose in main	In the Dependency Injection setup in the <code>main()</code> function, the repositories and view models were missing calls to their <code>dispose()</code> functions. This could lead to memory leaks and a "dirty" state. The respective <code>Provider</code> instances have been updated.
Location of averaging multiple sensors	The team asked whether the averaging of multiple sensors should be placed in the view model or the repository. As the view model should only contain UI state, this functionality was moved to the repository.

Table 10: Implemented feedback from external code review

Feedback	Reason for not implementing Feedback
Implement the optional "Use Cases" layer of Flutter MVVM	For larger projects, Flutter recommends adding an additional logic layer with "Use Cases" in between the repositories and the view models [38]. Although the reviewer acknowledged that incorporating a logic layer might be over-engineering for the size of the app, he nonetheless recommended that the team proceed with its implementation. This would demonstrate the team's ability to implement clean architecture, which is a key aspect of this Bachelor's thesis. However, as Flutter only recommends this approach for larger projects, and because of the time constraints of the Bachelor's thesis, the team decided to focus on maturing the existing architecture instead.
Use a Dependency Injection container	The suggestion was to use a Dependency Injection container to better separate dependencies from UI widgets and improve life cycle handling. It was decided to not implement this, as the current provider-based architecture is recommended by Flutter, and the aforementioned points did not affect the project [37].
CoolerStatusRepository is not a real repository, as it only transforms state from existing repositories	At the time, the cooler status repository only mirrored the Bluetooth connection status. It was already planned to include more statuses in the future. This has been partially addressed by the introduction of UiStatus described above.
IBleService contains state	IBleService is the interface that provides common functionality to the repositories responsible for parsing data from their respective BLE service. Due to the naming of the interface and its placement in data/models/ble_service_interface.dart, the reviewer incorrectly assumed this was a service in the MVVM sense. To eliminate this ambiguity, the file has been moved to data/repositories/cooler_data/ble_data_interface.dart and the interface has been renamed to IBleData. There have been no changes to the content.
Improve state transformation in HomeViewModel	The view model for the home page exposed internal types such as CoolerStatus, which the view then used in a switch statement to display the correct UserMessage. The reviewer suggested moving CoolerStatus into the View Model. Although this is technically a MVVM violation, the team has decided to keep it in the View because of its single usage within it and the low logic content of the switch statement.
Decouple Bluetooth and data repositories and create a new service	In most cases, repositories should be independent of each other. However, the team chose to split the Bluetooth state functionality into multiple repositories. The ConnectionRepository handles searching for and connecting to Bluetooth devices, while the repositories in the cooler_data directory handle the data from each individual BLE service. All cooler_data repositories depend on ConnectionRepository. The reviewer suggested creating a service between them. However, the team believes that implementing a service like this would introduce needless complexity. The current split adheres to the Single Responsibility Principle by separating functionality; therefore, a violation of the repository independence is acceptable.

Table 11: Not implemented feedback from external code review

6 Results

6.1 Fulfilment of Functional Requirements

This chapter verifies the fulfilment of the functional requirements defined in [Section 2.2.1](#) and their acceptance criteria. The evaluation is based on the final state of the app and backend prototype at the end of the Bachelor's thesis.

Some requirements were adapted or removed from the scope of the implementation during the project due to time constraints, changes in the industry partner's priorities, or findings from usability testing. These adjustments are detailed in the relevant user story sections. The acceptance criteria are marked with "x" if the team was unable to fulfil them and "†" if the industry partner has decided to drop them.

6.1.1 User Story 1 – Establish Connection to Cooler

The initial requirement specified the inclusion of a first run wizard that guides the user through the connection process when the app is started for the first time. While wireframe designs for the wizard were created and used in the first usability testing (see [Section 5.2.5](#)), the wizard was removed from the implementation scope due to time constraints. However, to make the connection process as accessible as possible, the home page contains a clear, actionable button for connecting the app to the cooler. Tapping this button opens a device picker that lists the available Bluetooth devices found during scanning. [Figure 27](#) shows the connection process.

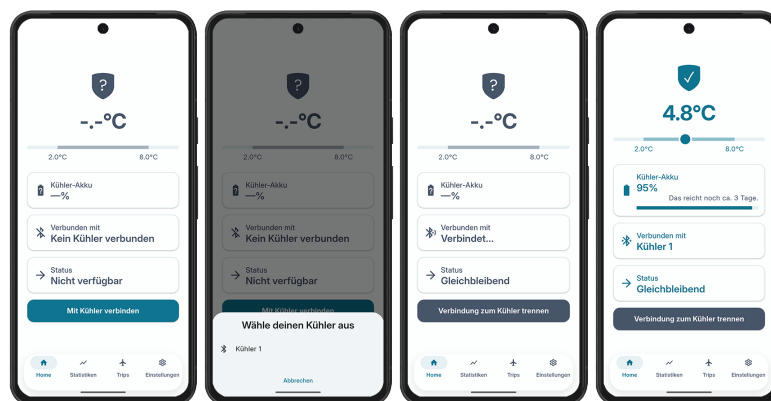


Figure 27: Connecting to a cooler

The implemented connection flow supports unlinking the current cooler and connecting to another one by using the "Disconnect" button on the home page. The same button can then be used to open the device picker again and connect to a different cooler. This fulfils the requirement that a damaged, lost or replaced cooler can be exchanged without resetting the application.

Status	Acceptance Criteria
x	A first run wizard guides the user through the connection process when the app is started for the first time
✓	It is possible to unlink the current cooler and connect with a new one (e.g. the old cooler is damaged or lost)

Table 12: Status of the acceptance criteria for user story 1

Since the first run wizard was not implemented, not all acceptance criteria of this user story are fulfilled. However, the core connection and reconnection functionality is available in the final prototype.

For future development, it was suggested that the first run wizard should also be accessible from the settings page. This would enable users to repeat the guided connection process when replacing or reconnecting a cooler without having to reset their app.

6.1.2 User Story 2 – System Status

The app's home screen displays the cooler's key status information, such as the current temperature, whether it is within the correct temperature range, the battery charge level, remaining battery time and any warnings and errors. These values enable users to quickly assess whether the cooler is operating correctly and how long it can continue to work.

After discussion with the industry partner, the ambient temperature was removed from the implementation scope, as it was decided that it could cause more confusion than benefit. Instead, the home page focuses on information most relevant to the user's immediate decision-making, such as the inside temperature, battery state and warning messages.

Problems such as a low battery level or temperature outside the configured range are displayed in on the app's home screen. These warning states are reflected both visually and through user messages, so that the user not only sees that a problem has occurred, but also receives guidance on what it means and how to fix it. [Figure 28](#) shows the different states of the home screen.



Figure 28: Home screen in different states: disconnected, connected with charging and discharging battery, battery warnings, battery errors and temperature errors.

Status	Acceptance Criteria
✓	The current temperature inside the cooler can be seen on the app's home screen
†	The current ambient temperature can be seen on the app's home screen
✓	The remaining battery charge can be seen on the app's home screen
✓	The remaining battery time can be seen on the app's home screen
✓	If a problem occurs, it is displayed on the app's home screen

Table 13: Status of the acceptance criteria for user story 2

As all the acceptance criteria required by the industry partner at the end of the project have been met, this user story can be considered complete for the final prototype.

6.1.3 User Story 3 – Continuous Data Retrieval from Cooler

Once connected to the cooler, the app receives live data via BLE and displays it on the home page. Battery and temperature data are continuously synchronised, with live updates occurring approximately once per second. This means that the home screen can show the current state of the cooler without the user having to refresh it manually.

As mentioned in the previous chapter, the final prototype does not display the ambient temperature because it was removed from the scope of implementation following discussions with the industry partner.

Warning states are updated as soon as the app receives the relevant status data from the cooler.

Status	Acceptance Criteria
✓	When connected, the app displays the current live data from the cooler, as defined in " User Story 2 – System Status "

Table 14: Status of the acceptance criteria for user story 3

This user story is fulfilled. The app displays the implemented live data defined for the final version of [User Story 2 – System Status](#) when connected to the cooler.

6.1.4 User Story 4 – Send Collected Cooling Data to the App

Historical data is stored in the local database so that the app can display the cooler's data over time. While the app is connected to the cooler, new historical entries are saved in the local database at regular intervals. The history page can then retrieve this data and present it in temperature history charts.

The app also supports synchronisation following an interruption. When it reconnects to the cooler, the app requests any historical entries that have not yet been stored in the local database. The cooler then sends these entries, which are inserted into the local database and become available on the history page once the process has finished. This mechanism ensures that data collected by the cooler while the app was offline can still be displayed later. [Section 4.1.5](#) describes the details of this process.

[Figure 29](#) shows the process of synchronising the history data after a connection interruption.

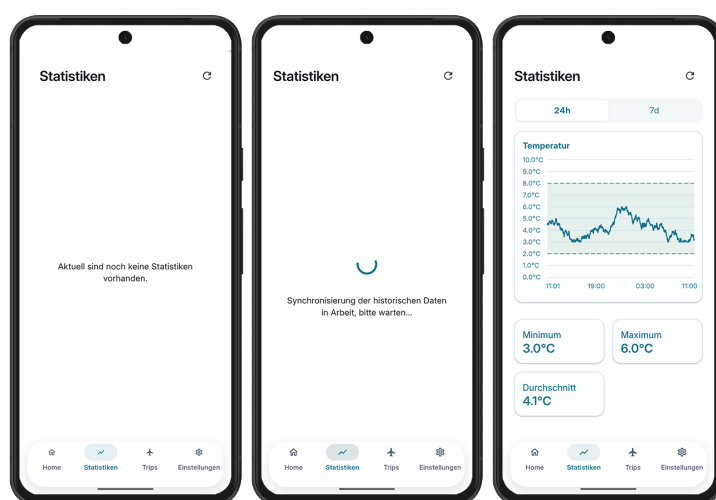


Figure 29: History synchronisation process

Status	Acceptance Criteria
✓	When the app is reopened, data collected while it was not running in the foreground must be synchronised and displayed
✓	If the connection between the app and the cooler is lost and then re-established, data collected during the interruption must be synchronised and displayed in the app

Table 15: Status of the acceptance criteria for user story 4

This user story is fulfilled. Historical data can be synchronised when the app is reopened, or when the connection to the cooler is interrupted and then re-established.

6.1.5 User Story 5 – Status Notification

As long as an active BLE connection exists, the app will continue to synchronise data with the cooler. This means that the local database can receive historical data while the app is connected. However, due to time constraints, push notifications were removed from the scope of the implementation.

Consequently, warnings such as low battery or temperature outside the configured range are displayed in the app as seen in [Figure 28](#), but are not sent as push notifications to the operating system while the app is running in the background. Therefore, while the app fulfils the synchronisation-related part of this user story, it does not fulfil the notification-related acceptance criteria.

Status	Acceptance Criteria
✓	The app will continue synchronising data with the cooler while it is connected and running in the background
×	The app will send a push notification if the temperature exceeds a defined limit while the app is running in the background
×	The app sends a push notification if the battery level falls below a defined limit while the app is running in the background

Table 16: Status of the acceptance criteria for user story 5

This user story has only been partially fulfilled. Although continuous synchronisation while connected has been implemented, push notifications have not been included in the final prototype due to time constraints.

6.1.6 User Story 6 – Temperature and Battery History

The app provides a history page for viewing the temperature inside the cooler over different time periods. There are two different chart types offered: For the last 24 hours, the temperature history is displayed as a line chart, enabling users to inspect detailed changes over time. For the previous seven days, the data is displayed as a bar chart with aggregated daily values. Each bar represents the minimum and maximum temperature the cooler reached on that day. In addition to the charts, the minimum, maximum and average temperature for the selected time period are displayed in boxes below the chart.

Users can switch between the 24-hour and seven-day views using a toggle. When the selected time period changes, the minimum, maximum and average boxes are updated accordingly to always refer to the current chart data.

The charts also support tooltips for a more detailed inspection. In the 24-hour line chart, tapping on a data point displays the exact measured value and timestamp. In the seven-day bar chart, tapping on a bar displays the minimum, maximum and average temperature for that day.

The safe temperature range is also visualised in the charts. Additionally, chart elements are highlighted differently when temperature values are outside the configured range. This helps users to identify discrepancies without having to interpret the raw data manually.

Figure 30 shows the different views of the history page.

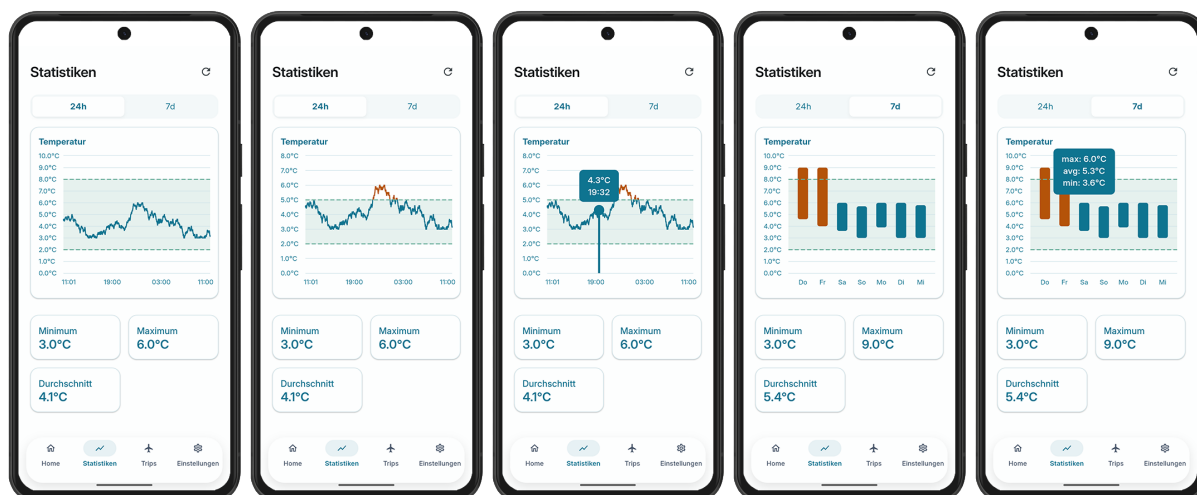


Figure 30: History page with a line chart showing temperatures within and outside the target range, including a tooltip view, and a bar chart shown both with and without a tooltip.

The ambient temperature history was removed from the scope of the implementation because the industry partner decided that ambient temperature provides little value to the users. The battery history was also removed due to time constraints. Consequently, the implemented history functionality focuses on the most relevant safety-related value: the inside temperature of the cooler.

Status	Acceptance Criteria
✓	The app shows the inside temperature history over a specified time period
†	The app shows the ambient temperature history over a specified time period
×	The app shows the battery history over a specified time period
✓	The app shows the average temperature over a specified time period
✓	The app shows the minimum and maximum temperature over a specified time period

Table 17: Status of the acceptance criteria for user story 6

This user story has been partially fulfilled. The implemented history page meets the requirements for displaying inside temperature history and providing temperature statistics. However, ambient temperature and battery history are not shown as the former was dropped by the industry partner, while the latter was not implemented due to time constraints.

6.1.7 User Story 7 – Trips

The trips feature was removed from the scope of implementation. This decision was based on two reasons. Firstly, the initial usability test revealed that users did not fully comprehend the purpose of the feature. Secondly, implementing the feature would have required additional data modelling, UI screens and filtering logic and this could not be completed within the available time.

Consequently, the app does not enable users to start or stop trips, name them, or view data grouped by trip. However, historical data can still be viewed through the regular history page, just not through trip-specific views.

Figure 31 shows the current trips page.



Figure 31: The trips page shows a coming soon text

Status	Acceptance Criteria
†	The app lets the user start and stop a trip
†	The trip can be given a custom name
†	Data from past trips can be viewed in detail
†	Data from the active trip can be viewed in detail

Table 18: Status of the acceptance criteria for user story 7

This user story is not fulfilled because the feature was removed from the final prototype.

6.1.8 User Story 8 – Set Temperature Range

The app enables users to adjust the target temperature range of the cooler from the settings page. To prevent invalid input, several constraints are applied before the values are sent to the cooler: The cooler must be connected, the minimum and maximum temperatures must be within the permitted range and the minimum temperature must also be at least 1°C lower than the maximum temperature.

Once a valid temperature range has been successfully sent to the cooler, the values are saved in the app's local key-value storage. This means that the app can continue to display the last saved values even after it has been restarted.

Figure 32 shows the process of changing the temperature range.

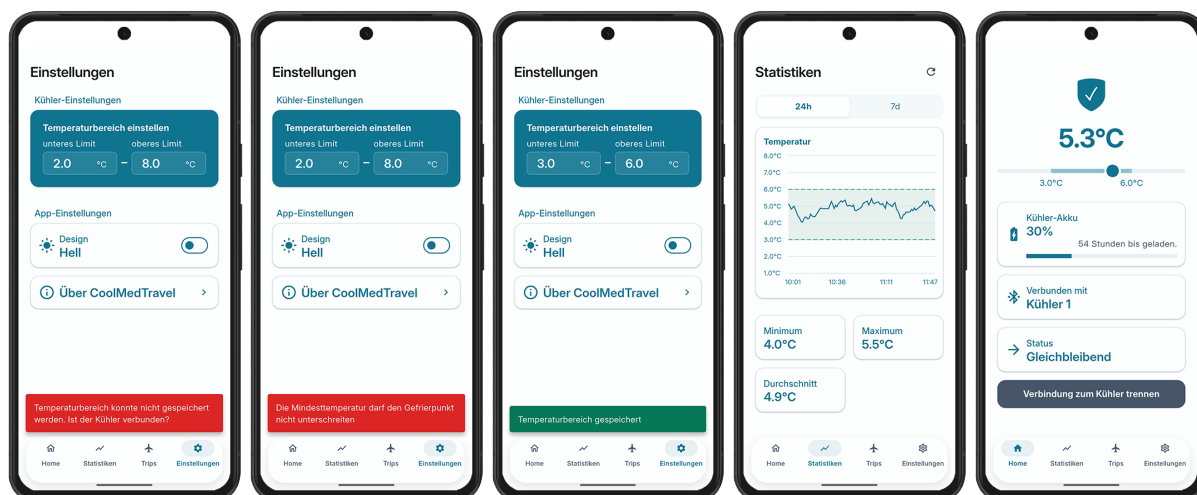


Figure 32: Settings page displaying error and success messages, as well as the updated interface after a successful temperature range change.

The current implementation has one limitation: the temperature range cannot be read back from the cooler. The app starts with the default range of 2°C to 8°C, after which it stores values locally. If a different app connects to the same cooler and changes the values, these changes are not synchronised with this app. Therefore, while the implementation fulfils the requirement to set the temperature range from the app, it does not yet provide full bidirectional synchronisation of the cooler settings.

Status	Acceptance Criteria
✓	The cooler's temperature range can be set using the app
×	The app displays the set temperature range from the cooler after connecting

Table 19: Status of the acceptance criteria for user story 8

This user story is partially fulfilled. The cooler's temperature range can be set using the app, with the limitation that the current values are stored locally and are not read back from the cooler.

6.1.9 User Story 9 – Rapid Cooling

The rapid cooling feature was removed from the app's user workflow following the initial usability test. Test participants did not clearly understand when or why they should manually enable this mode. Furthermore, the industry partner and the project team concluded that there was no realistic scenario in which a user would decide against rapid cooling when the required conditions were met.

For this reason, the manual rapid cooling control has been removed from the final prototype. The intention is that the cooler will start rapid cooling automatically when it is useful and technically possible. As the user will no longer control this mode directly, the app will not provide a separate information screen for enabling it manually.

Status	Acceptance Criteria
†	Rapid cooling can be initiated from within the app
†	Information about this mode is provided when it is enabled

Table 20: Status of the acceptance criteria for user story 9

The user story cannot be fulfilled in its original form because the feature has been deliberately removed from the scope by the industry partner.

6.1.10 User Story 10 – Configure Alarms

The custom alarm configuration feature was not implemented in the final prototype. This feature depends on the notification system because the configured alarm values would be used primarily to trigger push notifications for low battery levels or temperature deviations. As push notifications were removed from the scope of the implementation due to time constraints, custom alarm configuration was removed as well. Both of these features will be developed by the industry partner in the future.

While the app is open and connected to the cooler, warning states are still displayed, but users cannot define their own alarm thresholds in addition to the predefined warning levels. It would have been possible to configure custom threshold values for in-app messages only. However, this was not implemented due to time constraints, as the main value of the feature depends on background notifications.

Status	Acceptance Criteria
×	The user is able to set a custom value for the alarm in addition to the pre-set alarms

Table 21: Status of the acceptance criteria for user story 10

This user story has not been fulfilled because push notifications were not implemented in the final prototype. Therefore, warning thresholds were also removed from the scope.

6.1.11 User Story 11 – Device-specific Data Retrieval

The backend provides an endpoint for exporting telemetry readings of a specific cooler, identified by its serial number. The request can also include start and end dates, allowing the exported data to be limited to a specific time period.

In addition to these filters, the endpoint supports filtering by telemetry property ID. While this is not part of the acceptance criteria for this user story, it could be a useful extension by allowing to request only the readings of a selected telemetry property for a specific cooler. However, this functionality is currently limited in practice as the filter requires the UUID of the telemetry property and the backend does not yet provide an API endpoint for retrieving these. Consequently, although the filter has been implemented, it is not yet very useful for API users.

[Listing 28](#) shows a sample request for retrieving readings from the backend.

```
1 http://srbsci-144.ost.ch:3000/readings?serial_number=3195841541&from=2026-06-01&to=2026-06-12&property_id=16f6f81c-d61f-475e-b1d8-7620d9ab0df5
```

Listing 28: Example request for retrieving filtered telemetry readings of a specific cooler

The response contains the readings that match the requested filters, together with metadata about the cooler and the telemetry property, as seen in [Listing 29](#). This makes the endpoint ideal for diagnostics and support cases, since an admin user can retrieve the stored telemetry data of a cooler without having to manually query the database.

```

1  {
2    "readings": [
3      {
4        "id": "129ba0b3-55dd-49d1-89d8-f59868ec8b0d",
5        "property_name": "Current Avg 1",
6        "value": 0.5,
7        "recorded_at": "2026-06-01 10:57:00",
8        "log_counter": 1
9      },
10     {
11       "id": "f80c6530-995a-481b-879e-fc396cc9a714",
12       "property_name": "Inside Top Temperature",
13       "value": {"avg": 2.95, "max": 3.01, "min": 2.84},
14       "recorded_at": "2026-06-01 10:57:00",
15       "log_counter": 1
16     },
17     /*...*/
18   ],
19   "cooler": {
20     "id": "4a62e368-9cbe-47dc-973b-17457262f858",
21     "serial_number": "3195841541",
22     "name": "1.1.1",
23     "model": "0"
24   }
25 }

```

Listing 29: Example response containing filtered telemetry readings and cooler metadata

Status	Acceptance Criteria
✓	Data from a specific cooler can be exported from the backend
✓	Data can be exported for a specific time period

Table 22: Status of the acceptance criteria for user story 11

This user story has been completed. The backend can provide telemetry readings for a specific cooler and restrict the results to a requested time period.

6.1.12 User Story 12 – Aggregated Sensor Data

This user story was not fulfilled. The database model stores telemetry readings with references to both coolers and telemetry properties, so the data model could support this feature in the future. However, the required backend endpoint for exporting telemetry data across all coolers was not implemented due to time constraints.

Status	Acceptance Criteria
×	Data from all coolers can be exported from the backend in a single request
×	The user can specify what properties they would like to export

Table 23: Status of the acceptance criteria for user story 12

6.1.13 User Stories About Possible Extensions

As it was not possible to fully implement all the main user stories, the majority of the optional user stories were excluded from the final project scope.

The exceptions were the language switcher and dark mode. Preparatory work on the language switcher was completed by defining all strings displayed to the user in localisation files through Flutter's `l10n` library [153]. While the app currently only supports German, this implementation will enable adding additional languages in the future with minimal effort.

Dark mode was implemented as part of the project. Although it was originally defined as an extension feature, the application structure and theme configuration were designed with this feature in mind. Consequently, it could be added during development with minimal additional effort. The app will open in the system's default theme on first run. Users can manually switch between light and dark mode on the settings page.

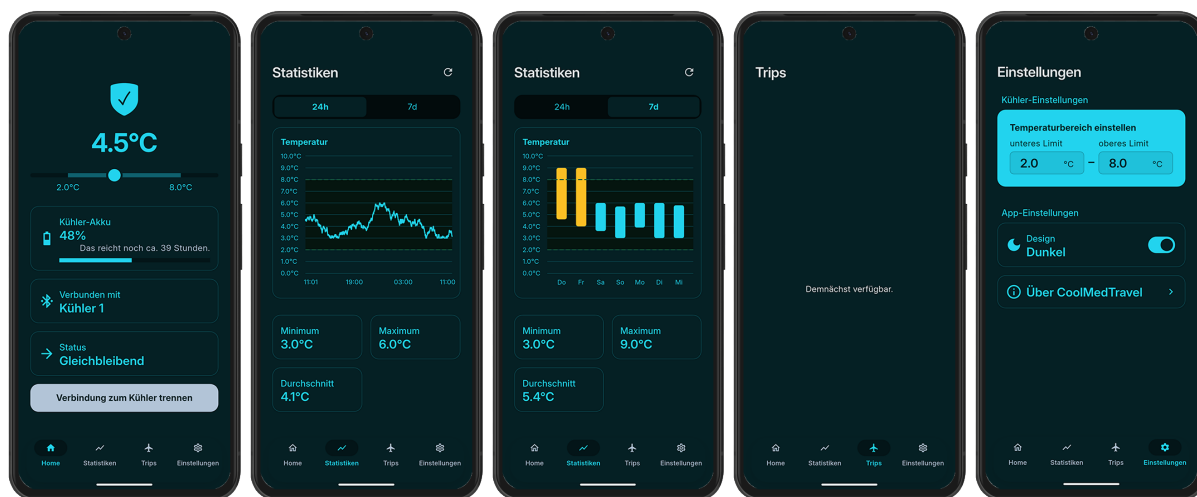


Figure 33: The pages of the app in dark mode

No other user stories related to possible extensions were implemented.

6.1.14 Summary

The final prototype fulfilled the main functional requirements to varying degrees. Several core features of the app were successfully implemented, including connection to the cooler, display of live status data, synchronisation of historical data and temperature history view. The backend also provides the ability to store telemetry data and export readings for a specific cooler over a selected time period.

Some requirements were only partially or not at all fulfilled because certain acceptance criteria were removed from the scope of implementation during the project. These changes to the scope were mainly caused by the industry partner's changed priorities, findings from usability testing, or time constraints. The affected app features include the first run wizard, ambient temperature display, battery history, push notifications, trips, rapid cooling, configuration of the cooler's temperature range and custom alarm configuration.

The export of telemetry readings for a specific cooler was implemented for the backend. However, broader export functionality for aggregated telemetry data across all coolers was not implemented. Consequently, the backend supports the export of telemetry data for individual coolers, but not for all coolers at once.

The user stories about possible extensions were mostly not implemented, as discussed in the previous section. [Table 24](#) provides an overview of the fulfilment status of the main functional requirements.

User Story	Status	Reason
User Story 1 – Establish Connection to Cooler	(✓)	Time constraints
User Story 2 – System Status	(✓)	Scope changes
User Story 3 – Continuous Data Retrieval from Cooler	✓	
User Story 4 – Send Collected Cooling Data to the App	✓	
User Story 5 – Status Notification	(✓)	Time constraints
User Story 6 – Temperature and Battery History	(✓)	Time constraints and scope changes
User Story 7 – Trips	†	Usability Test feedback and time constraints
User Story 8 – Set Temperature Range	(✓)	Time constraints
User Story 9 – Rapid Cooling	†	Usability Test feedback
User Story 10 – Configure Alarms	×	Time constraints
User Story 11 – Device-specific Data Retrieval	✓	
User Story 12 – Aggregated Sensor Data	×	Time constraints

Table 24: Summary of the main functional requirement fulfilment

In addition, the optional [User Story 21 – Dark Mode](#) was also fulfilled.

6.2 Fulfilment of Non-Functional Requirements

This chapter verifies the fulfilment of the non-functional requirements defined in [Section 2.2.2](#) and their acceptance criteria.

6.2.1 NFR1: Maintainability – Testability

The [Description of NFR 1 “Maintainability – Testability”](#) states that the test coverage must be at least 80%.

A total of 245 unit and widget tests were written for the implemented functionality of the app. These tests cover key elements of the app’s architecture, including UI elements, repositories, services, commands and view models, as detailed in [Section 5.2.1.1](#) and [Section 5.2.2](#). The tests are automatically run in the app repository’s CI/CD pipeline and a coverage report is generated, as described in [Section 5.3.2.4](#). The coverage reports are visible in [Figure 34](#) and [Figure 35](#).

With these tests, a line coverage of 85.5% was reached in the app, thus fulfilling the acceptance criteria for the app.

As discussed in [Section 5.2.3.1](#), integration tests were not implemented due to their impracticality within the project’s scope. Therefore, the current test suite verifies individual units of the app, including their public members exposed to other MVVM layers. At present, it does not automatically test complete interaction flows across the app. The addition of integration tests remains future work for the industry partner.

The backend has a limited number of tests, as the majority of functionality is provided by Gerust or Axum. The only components that required testing were the CRUD functions for the database entities and the functionality of the API controllers. As outlined in [Section 5.2.1.2](#) and [Section 5.2.3.2](#) only 5 API tests were written for the backend, bringing the code coverage of the backend to 51.21%.

The 3% coverage limit was generally adhered to, but not consistently. For instance, some larger merge requests for the MVVM restructuring were merged before all the necessary tests had been added so that other branches could be rebased safely. The missing tests were then added in subsequent merge requests.

LCOV - code coverage report

Current view: top level		Coverage		Total	Hit
Test: lcof_cleaned.info		Lines:	85.5 %	2339	2001
Test Date: 2026-06-12 08:07:29		Functions:	-	0	0

Directory	Line Coverage ↕			Function Coverage ↕		
	Rate	Total	Hit	Rate	Total	Hit
lib/data/models/cooler_data/lib/data/models/cooler_data	49.3 %	67	33	-		
lib/data/models/history/lib/data/models/history	98.1 %	52	51	-		
lib/data/models/lib/data/models	48.1 %	52	25	-		
lib/data/parsers/lib/data/parsers	100.0 %	50	50	-		
lib/data/repositories/cooler_data/lib/data/repositories/cooler_data	93.9 %	181	170	-		
lib/data/repositories/lib/data/repositories	96.1 %	181	174	-		
lib/data/services/lib/data/services	78.7 %	141	111	-		
lib/extensions/lib/extensions	100.0 %	65	65	-		
lib/lib	1.7 %	119	2	-		
lib/ui/commands/lib/ui/commands	78.6 %	14	11	-		
lib/ui/models/lib/ui/models	91.7 %	36	33	-		
lib/ui/pages/lib/ui/pages	90.8 %	444	403	-		
lib/ui/view_models/lib/ui/view_models	88.7 %	284	252	-		
lib/ui/widgets/charts/lib/ui/widgets/charts	97.6 %	287	280	-		
lib/ui/widgets/lib/ui/widgets	93.2 %	366	341	-		

Generated by: [LCOV version 2.0-1](#)

Figure 34: Final app code coverage report

/builds/coolmedtravel/backend		Covered: 232 of 453 (51.21%) 🌙
Path	Coverage	
cli	0 / 0	
config	19 / 45 (42.22%)	
db	108 / 279 (38.71%)	
macros	0 / 0	
web	105 / 129 (81.40%)	

Figure 35: Final backend code coverage report

The NFR has therefore been fulfilled for the app, but not for the backend.

6.2.2 NFR 2: Interaction Capability – Learnability

The Description of NFR 2 “Interaction Capability – Learnability” states that the app must be easily navigable by members of the target audience with varying degrees of technical proficiency.

Two usability tests were conducted to fulfil this NFR. The first was conducted with the utilisation of wireframes, while the second test was conducted with the app prototype.

6.2.2.1 First Usability Test

On the 8 and 9 April 2026, the first usability test was conducted with five participants with the wireframes seen in Section 3.6.1. The average SUS Score was 82, which exceeds the target minimum of 68.

Several issues were raised by multiple participants, primarily that the rapid cooling mode and the trip function were not understood. Therefore, the rapid cooling mode will be removed from the user workflow and activated automatically if applicable. An explanation of the trip function will be added to the onboarding process. Guidance should also be provided on every page through the means of a help button, particularly for those who are less familiar with technology. Adding a video tutorial was also considered beneficial.

The chapters below show the detailed feedback, divided into features that can be easily implemented and changes that fall outside the scope of this project. The latter are categorised under “Possible extensions”.

Implemented Changes

- Using a clearer icon for the statistics menu item
- Adding text labels to menu icons
- Showing battery life in days (>48h) or hours (<48h)
- Warnings should be clear, actionable and consistently positioned
- Adding an indication of how long it takes to charge the battery
- Enabling automatic synchronisation, but adding a manual sync button as well
- Ensuring all key information is visible on the home screen without scrolling

Possible Extensions

- Including icons and images in the initial setup, e.g. for Bluetooth connection
- Adding “min” and “max” labels to the temperature range in the onboarding
- Indicating in the onboarding that the cooler is not on the target temperature yet, with a notification when it is
- Automatically start rapid cooling mode if possible instead of letting the user decide
- Explaining the trip function during onboarding
- Displaying status next to the temperature (or via a subtle line chart in the background)
- Adding a animated icon to the home screen to indicate whether the cooler is active
- Adding the ability to enable and disable the cooler in the app settings
- Creating a video tutorial
- Adding question mark icons on each page linking to help pages and the video tutorial
- Adding battery level and temperature forecasts on the statistics page

6.2.2.2 Second Usability Test – Expert Evaluation

Due to time constraints and the fact that the app had not yet been developed sufficiently to yield meaningful results from a second usability test, a heuristic evaluation by an expert was carried out instead. This evaluation was conducted on 28 May 2026 with Prof. Dr. Markus Stolze, who is a professor of user-experience engineering at OST. Where possible, the questions defined in [Section B.1](#) were asked during the testing. Afterwards, the expert provided some more general feedback on the app and the process.

The expert gave the app a SUS Score of 97.5, exceeding the target minimum of 68 by a significant margin.

The chapters below show the detailed feedback, structured the same way as in [Section 6.2.2.1](#).

Implemented Changes

- Adding a button to connect the app with the cooler
- Checking the app colours for sufficient contrast in accordance with WCAG guidelines [154]
- Making the safe range in the chart more clearly visible
- Warnings should be clear, actionable and consistently positioned
- Ensuring text fields are not covered when the keyboard opens
- Renaming the “battery” infobox to “cooler battery”

Possible Extensions

- Using warning colours only for the message box, while indicating the element that caused the warning in a different way, for example with an icon
- Testing the app with users who have colour blindness
- Adding statistics that show how much time the temperature was inside and outside the safe range
- Clarifying how the chart is displayed when the cooler battery has been empty for a certain amount of time
- Making chart tooltips more discoverable, for example by showing one already opened when the statistics page is opened
- Adding question mark icons on each page linking to help pages
- Adding videos or images explaining how to activate Bluetooth pairing on the cooler
- Taking right-to-left languages into account in language support

Given the score achieved and the improvements that have been made as a result of the testing, the NFR can be considered fulfilled.

The process of the usability tests is outlined in [Section 5.2.5](#), while the detailed protocols of the usability test and the expert evaluation can be found in the appendix in [Section B.2](#).

6.2.3 NFR3: Functional Suitability – Functional Correctness

The [Description of NFR 3 “Functional Suitability – Functional Correctness”](#) states that the integrity of the data from the medication cooler must be ensured.

During the course of this Bachelor’s thesis, it was discussed that the industry partner already validates the data on the cooler before it is sent to the app. Consequently, extensive additional plausibility validation was not implemented in the app itself.

The current architecture provides a solid foundation for future enhancements. Parsing is handled per BLE service in its respective repository, where the validated data is stored in models. Other parts of the app source data from these models, and therefore do not need to conduct additional verifications. Consequently, any additional validation logic can be incorporated into the repositories, eliminating the need for duplicate checks across the application.

A special case is the history data, which is encoded as CBOR rather than sent as a plain data type. Due to the size restrictions of BLE, the CBOR data is split into multiple chunks. The app must collect the individual chunks, reassemble them, and then parse the complete message. This process has been kept simple, with invalid packages (caused by a weak Bluetooth signal, for example) being dropped without the app having a mechanism to request the data again. The industry partner has accepted this limitation for the prototype.

Although the data accessed by the view models is verified, invalid packages are dropped by the CBOR parser and are not re-requested from the cooler. Therefore, NFR3 is considered partially fulfilled.

6.2.4 NFR4: Reliability

The Description of NFR 4 "Reliability" states that all data intended for transmission must be eventually synchronised between the app, medication cooler and backend.

The synchronisation functionality has been implemented and is operational. Historical data can be synchronised from the cooler to the app, where it is stored locally before being synchronised with the backend. Eventual consistency is implemented via a log counter assigned to all values of a given timestamp. The app sends its latest log counter to the cooler, which responds by sending back all values with a greater log count. The same mechanism is also used to send data to the backend.

However, the process of synchronising historical data from the cooler to the app takes longer than the originally defined target of 10 seconds. This is caused by the low data throughput and instability of BLE, especially on iOS. During manual testing, no live or historical data was sent to the iOS app due to the iOS Bluetooth stack being unable to handle BLE operations in quick succession. To make the synchronisation process more reliable, delays have been added between BLE operations. While these delays increase the overall synchronisation duration, they are necessary due to hardware and BLE communication limitations. More about this can be read in Section 5.2.4.

The backend synchronisation has been implemented and is functional. The process is initiated by the app after a successful historical synchronisation where the received data has been stored in the local database. Although the duration of the process has not been benchmarked, the team did not notice significant delays when the app sent hundreds of measurements during manual tests.

Some synchronisation behaviour between the cooler and the app remains unstable due to race conditions in the BLE communication and should be investigated in future work. Optimising the synchronisation time should also be considered in future projects.

Therefore, NFR4 is fulfilled with limitations. Although the synchronisation functions for the prototype, it is unstable and the performance target could not be fully reached due to BLE and hardware constraints.

6.2.5 NFR5: Safety – Hazard Warning

The Description of NFR 5 "Safety – Hazard Warning" states that the app must notify the user if the desired temperature cannot be kept.

The cooler issues warnings, and the app displays these warnings as soon as it is connected to the cooler. The warning messages are clear and actionable, allowing users to understand the problem and react accordingly.

The battery level warnings were implemented correctly. Different notifications are displayed for 20%, 10%, and 5% battery remaining.

Temperature warnings were also implemented, but not exactly as described in the acceptance criteria. As per the initial requirement, a warning is to be displayed if the cooling chamber temperature remains outside the specified range for 30 seconds. In the current implementation, the warning is displayed immediately when the temperature is outside the configured range. This decision was made because the industry partner still needs to define how long the temperature may remain outside the valid range before a warning should be issued. It is yet to be determined whether this warning should be based on live data, which is received every second, or on historical data, which is transmitted once per minute.

In addition, the industry partner still needs to define how the app should behave at the start and end of cooling sessions when the cooler's temperature is not within the target range, and during non-cooling periods. In the current prototype, the app still displays temperature warnings in these states. In later versions, these warnings should either be suppressed or handled differently to avoid misleading the user. The same applies to the recorded data: values obtained during these phases are currently included in the displayed statistics, which may result in maximum and average temperatures that are higher than expected.

Currently, the warnings are only shown displayed within the app. Push notifications could not be implemented due to being removed from the project's scope because of time constraints. Thus, the user cannot know the current cooler status without opening the app.

Therefore, NFR5 is partially fulfilled. The app issues warnings to users regarding critical battery and temperature states. However, the delayed temperature warning and push notifications have not yet been implemented.

6.2.6 NFR6: Maintainability – Analysability

The Description of NFR 6 “Maintainability - Analysability” states that the code needs to pass the rules set by the linters and formatters.

Linters and formatters with strict rulesets were configured for the documentation, the app and the backend in their respective IDEs and are integrated into the CI/CD pipeline. For the app, the pipeline runs the configured Flutter lint sets and Dart formatter with every push. For the backend, the Rust tooling includes formatting and linting through `rustfmt` and `clippy`. Typst code is also formatted in the pipeline with the use of `typstyle`. This is described in more detail in [Section 5.1.7.1](#), [Section 5.1.7.2](#) and [Section 5.3](#). The result is improved maintainability through increased code uniformity.

Therefore, NFR6 is fulfilled.

6.2.7 Summary

The final prototype fulfils the non-functional requirements to varying degrees. Several quality goals were successfully addressed.

However, some requirements were only partially fulfilled because certain acceptance criteria could not be fully implemented or verified within the scope of this Bachelor's thesis. The main limitations concern the incomplete plausibility validation of received data, BLE-related synchronisation delays and warnings that only appear inside the app rather than as push notifications.

Table 25 provides an overview of the fulfilment status of all non-functional requirements.

Requirement	Status	Reason
<u>NFR1</u>	(✓)	Unit tests are implemented and the coverage is at 51.21 percent for the backend and 85.5 percent for the app. Integration tests remain future work.
<u>NFR2</u>	✓	SUS scores exceeded the target minimum of 68 in both usability tests.
<u>NFR3</u>	(✓)	Basic parsing is implemented, but comprehensive plausibility validation remains future work.
<u>NFR4</u>	(✓)	Synchronisation is implemented, but the 10-second target could not be met due to BLE and hardware limitations.
<u>NFR5</u>	(✓)	Battery and temperature warnings are implemented, but delayed temperature warnings and push notifications remain out of scope.
<u>NFR6</u>	✓	Linters and formatters are configured and run in the CI/CD pipeline. The linters are configured with a more stringent set of rules than the default.

Table 25: Summary of non-functional requirement fulfilment

6.3 Conclusion

The objective of this Bachelor's thesis was to design and implement a companion application for a portable medication cooler, as well as a backend for storing and aggregating cooler data. Not all initially planned features could be implemented within the available time, either due to time constraints or scope changes. Nevertheless, the result is a functional prototype that encompasses the fundamental interaction between the cooler, the mobile application and the backend.

The app can connect to the cooler to display live status information and synchronise historical data, as well as visualise temperature history. In addition, telemetry data can be transferred to the backend. This gives the industry partner a solid basis for continuing to develop the system.

A strong focus was placed on code quality and maintainability. The application structure is based on the MVVM architecture and supported by unit tests, strict linting rules, CI/CD pipelines and code documentation. These measures should make it straightforward for the industry partner or future student projects to understand, maintain and extend the codebase.

Overall, the thesis successfully achieved its main goal of providing a functional prototype. Although some features are still to be finalised, the implemented system proves that the planned companion application and backend are technically feasible, providing a solid basis for ongoing development.

6.4 Outlook

The project can be continued in several directions. One important area is optimising the front end. Future work could involve improving the user experience, refining the visual design and adding a first run wizard and help pages to guide users through the use of the application.

Another important task is to improve feature stability. In particular, further testing and improvements are needed to make Bluetooth communication and historical data synchronisation more robust. This is particularly important before the system can be used in a production environment.

Several features that were excluded from the final scope could also be implemented in future iterations. These include the trip feature, push notifications and a more detailed alarm configuration option. The incorporation of such features would enhance the application's utility in real travel scenarios and give users greater control over the cooler.

The possible extension user stories defined in [Section 2.2.1.14](#) also provide a valuable basis for future work. Of these extension features, only the dark mode was implemented during the course of this thesis. The remaining user stories, such as language and temperature unit switching, opting out of usage data tracking, user feedback, notification permission handling, GDPR/nFADP compliance or support for multiple coolers and apps, remain open. Depending on the intended direction of the product, the industry partner could prioritise these features for future projects.

The backend also offers extensive possibilities for extension. For instance, a dashboard could be developed to facilitate the inspection, export and analysis of cooler data by the industry partner. In the long term, the collected telemetry data could be used to create a digital twin of the cooler. This could support simulations, predictive maintenance and further optimisation of the cooling behaviour.

These extensions would turn the current prototype into a more comprehensive and stable product. They would enable users to safely store temperature-sensitive medication and allow the industry partner to improve the cooler based on real-world usage data.

A Project Plan

A.1 Project Management

This project is managed using the RUP (Rational Unified Process) method. This is a method for long-term planning which divides a project into four phases [155]:

- **Inception:** In this phase, the approximate project goals and scope are defined. In addition, a rough schedule is drawn up.
- **Elaboration:** This is where the requirements are identified and the core architecture is implemented. The highest risks should be eliminated and time estimates should become more realistic during this phase.
- **Construction:** During this phase, product functionalities are implemented and the minor risks are eliminated. The deployment is prepared.
- **Transition:** In this phase, the project is completed. The final details are finalised and the product is deployed.

The project is additionally split into two-week sprints for short-term planning.

A.1.1 Project Plan

Figure 36 shows the detailed project plan. The project is divided into three areas of work: documentation, app development and backend development. The chart also shows the division into the four phases according to RUP and the sprints. Since there are no lectures for a week over Easter, sprint 3 will last three weeks instead of two.

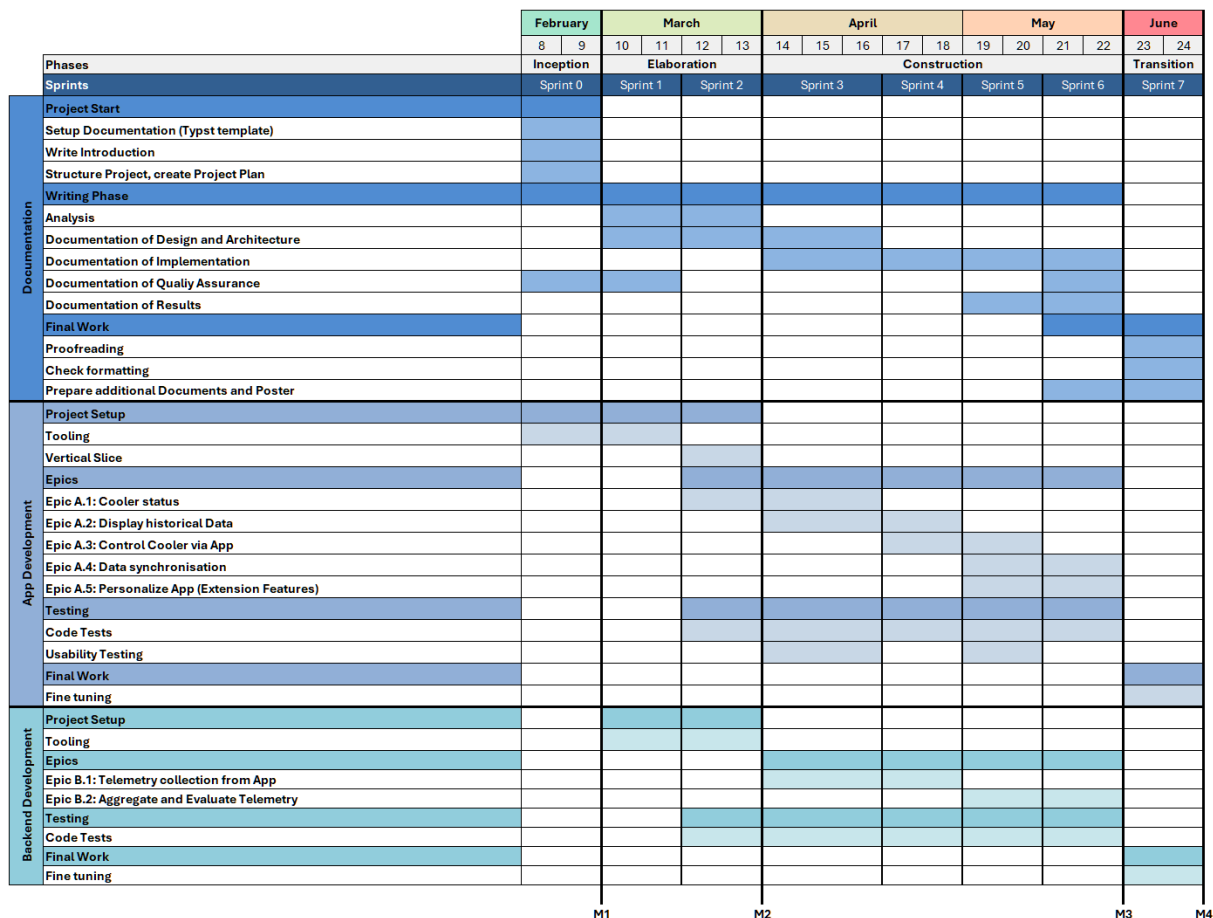


Figure 36: Project plan with work areas, sprints and RUP phases

A.1.1.1 Changes to the Project Plan

- Due to delays on the side of the industry partner during the development of the BLE protocol, the stakeholders agreed at the M2 meeting that all functionality should initially be created using dummy data. The communication aspect of the affected user stories has been transferred to a new Epic A.4. Implementation of these features could start once the BLE simulator has been received in sprint 3.
- In the week 12 meeting, the team announced their intention to drop the implementation of User Story 7 – Trips due to time constraints and poor reception in the first usability testing.
- During the aforementioned meeting, the decision was made to cancel the second round of usability testing in its initially planned form, as the app had not been developed to the stage where significant changes in feedback compared to the wireframes could be anticipated. Instead, an expert evaluation was conducted, as detailed in Section 5.2.5.

A.1.2 Milestones

The following milestones serve as a general guide for when certain work areas should be completed.

M1: End of Inception (CW9, Sprint 0)

- Tooling and project setup are completed
- Introduction is written

M2: End of Elaboration (CW13, Sprint 2)

- Architecture is described in the documentation
- Vertical Slice is completed

M3: End of Construction (CW22, Sprint 6)

- Documentation has been completed, apart from final adjustments and the results
- The App and the Backend are feature-complete

M4: Final Submission (CW24, Sprint 7)

- Documentation and all other documents to be submitted are ready
- The App and the Backend are developed according to the objectives defined at the beginning of the project

A.1.2.1 Changes to the Milestones

While the milestones themselves were not moved during the project, the planned work for each one shifted considerably. At the beginning of the project in particular, the team focused more strongly than initially planned on familiarising themselves with the technologies and implementation work, while parts of the documentation were postponed.

Consequently, the introduction chapter was not completed by M1. Similarly, the architecture description was not fully written by M2. The planned vertical slice was also not completed in its originally intended form, mainly due to the unavailability of the BLE protocol and simulator at that stage. Additionally, a lot of time had to be spent on implementing a proper MVVM architecture and tests for them.

The largest deviation occurred around M3. According to the original milestone plan, the app and backend should have been feature-complete by the end of construction. In practice, however, implementation work continued into the final sprint. This was due to delayed BLE integration on both the app and BLE simulator side, changes in scope and the decision to prioritise the completion of core app and backend functionality. In particular, due to a focus on the app, meaningful work on the backend and the implementation and results chapters of the documentation only started after M3.

A.1.3 Meetings

A meeting with the advisors and the startup is held once every week to review the current status and clarify open questions. Not all individuals involved need to be present at every meeting. The meetings take place in the closed office of Laurent Metzger to ensure that no confidential information is disclosed to the outside world.

If there are no questions, the meeting can be canceled.

No official internal team meetings are planned, as both team members work on site and any open issues can therefore be addressed directly.

A.2 Risk Management

The matrix in [Table 26](#) shows the various risks to the success of the project that have been identified and classified. The probability of occurrence is listed on the left and the impact of the risk is listed above. The individual risks and their mitigations are described in more detail below the matrix. After each sprint, the risks are reevaluated based on their occurrence or applied mitigations and moved accordingly within the matrix.

	Low	Medium	High	Very High	Critical
Impossible					
Improbable	R5	R4, R6			
Possible	R5	R4, R6, R7	R2	R1, R3	
Probable			R2	R1	
Very Probable					

Table 26: Risk matrix of the project

Risk ID	R1
Risk	No app development experience
Description	Neither team member has ever developed an application for Android or iOS. The lack of experience could hinder development, as the team would need to spend time learning the fundamentals of app development.
Mitigations	To mitigate this risk, the team will use the elaboration phase to become familiar with Android Studio, the mobile application development workflow and the fundamental principles of app development such as mobile UX practices.
Changelog	16.03.2026: The team has now been working with Android Studio for the last two weeks and has become familiar with its workflows. Additionally, by preparing for the first UX test, the team has looked into best practices for mobile UI design. Because of that, the risk has been reduced from "Probable" to "Possible".

Table 27: Description of risk R1 "No app development experience"

Risk ID	R2
Risk	No experience with Dart and Flutter
Description	In addition to app development, the team also has no experience with the Flutter UI framework or the Dart language on which it is based. The unfamiliarity with its concepts may also contribute in the application development to take more time than initially planned.
Mitigations	The team will use the elaboration phase to familiarise itself with Flutter and Dart, thereby gaining experience in creating Flutter applications.
Changelog	16.03.2026: The team has completed multiple tutorials on working with Dart and Flutter, and now feels more comfortable when working with both. The risk has been reduced from "Probable" to "Possible".

Table 28: Description of risk R2 "No experience with Dart and Flutter"

Risk ID	R3
Risk	Difficulties with Bluetooth LE communication
Description	There may be difficulties implementing Bluetooth LE communication with the medication cooler with regard to operating system APIs. This is particularly relevant when implementing background synchronisation, as operating system battery optimisation may limit the amount of data that can be transferred when the app is running in the background.
Mitigations	During the elaboration phase, the team will look at pre-existing solutions for handling BLE communication.
Changelog	29.03.2026: The implementation of BLE functionality within the app has been postponed to a subsequent phase of the project due to the challenges encountered by the industry partners in developing the BLE protocol. The team has identified several packages capable of handling BLE communication within Flutter. However, a decision has yet to be made on which package to use for the implementation. 31.05.2026: While there have been some minor difficulties with BLE such as issues with the iOS Bluetooth stack, using the <code>flutter_reactive_ble</code> library has worked out well.

Table 29: Description of risk R3 "Difficulties with Bluetooth LE communication"

Risk ID	R4
Risk	Differing project expectations between the industry partner and the advisors
Description	During development, the industry partner and the advisors may have different expectations regarding what the final product should look like.
Mitigations	All involved stakeholders will agree on the user stories in the beginning of the project, establishing an initial version of the MVP. Later changes to the MVP will have to be discussed and approved by all stakeholders.
Changelog	19.03.2026: The requirements for the MVP are now fixed and agreed on by all involved parties. The risk has been moved from "Possible" to "Improbable".

Table 30: Description of risk R4 "Differing project expectations between the industry partner and the advisors"

Risk ID	R5
Risk	Project management overhead due to communication with the industry partner
Description	The inclusion of the additional stakeholder may lead to more time being spent with project management than desired, leading to slower development and communication fatigue between the stakeholders.
Mitigations	To mitigate this risk, the responsibilities and interaction processes will be clearly defined to minimise management overheads.
Changelog	19.03.2026: The workflow has stabilised. Meeting notes and action items are now documented in a shared OneNote that is accessible to all team members. Urgent questions are communicated via the Teams channel. Weekly meetings remain concise yet productive. The risk has been moved from "Possible" to "Improbable".

Table 31: Description of risk R5 "Project management overhead due to communication with the industry partner"

Risk ID	R6
Risk	Difficulties in creating an intuitive UI for the diverse target audience
Description	Discussions with the industry partner revealed that the target group for the medication cooler ranges from tech enthusiasts to people with little familiarity with technology. It may therefore be difficult to create a UI that can satisfy such a wide range of target demographics.
Mitigations	During the project, UX testing with a group of the target demographic will be conducted. The test subjects will be provided by the industry partner.
Changelog	09.04.2026: Following the initial UX test with subjects from the target demographic and the subsequent adjustment of the UI based on this feedback, the team is now more confident that the UI will be straightforward enough for a diverse user base. The risk has been lowered from "Possible" to "Improbable". 28.05.2026: The second usability test, conducted as an expert evaluation, provided the team with more detailed insights into subtle UI considerations which average user may not consciously notice, such as WCAG compliance or respecting OS screen boundaries. The design now provides a solid foundation that can be refined later on.

Table 32: Description of risk R6 "Difficulties in creating an intuitive UI for the diverse target audience"

Risk ID	R7
Risk	Absence of a team member
Description	A team member may be absent due to illness or an accident, for example.
Mitigations	Clean documentation makes it easier for the other team member to take over a task. If necessary, features can also be removed from the requirements.
Changelog	31.05.2026: There have been several brief periods of absence, such as Jannis' military service and Nina's illness. Despite these absences, progress during those periods was remarkably smooth, with the remaining team member able to build easily on the work already done by the absent person.

Table 33: Description of risk R7 "Absence of a team member"

A.3 Tooling

The tools used to write the code and documentation during the project are listed in this chapter.

A.3.1 Code

The tools described in this chapter were used for writing code. As the app and backend were created with a different tech stack, the tooling has been split into two sections.

A.3.1.1 App

The app was developed with the Flutter framework, which is written in the Dart language. The use of this language and framework was mandated by the industry partner.

- Flutter 3.44 [156]
- Dart 3.12 [157]
- Android Studio (IDE with Android SDK and emulator) [17]
- `dart_style` (Formatter behind `dart format`) [92]
- analyzer with the `flutter_lints` ruleset (Linter behind `flutter analyze`) [80], [83]
- `build_runner` (Code generator for Mock files) [110]

A.3.1.2 Backend

The backend was developed in Rust with the tooling below, as requested by the industry partner.

- Rust toolchain `stable-x86_64-unknown-linux-gnu` 1.94 in the Rust 2024 edition [158], [159]
- `rustfmt` (Formatter) [94]
- `clippy` with the `clippy::pedantic` ruleset (Linter) [90], [91]
- JetBrains RustRover (IDE) [160]

A.3.2 Documentation

The documentation has been written using the typesetting language Typst. The following tools have been used with it:

- Visual Studio Code [161]
- Typst compiler 0.14.2 [162]
- Tinymist (VS Code extension to preview Typst documents) [97]
- `typstyle` (Typst linter, integrated into Tinymist) [96]
- PlantUML for generating the Domain Model figure [163]
- PlantUML-C4 for generating the C4 Model figures [164]
- Figma for creating the wireframes [33]
- Code Spell Checker [165]

A.3.3 Pipeline

Inside the GitLab CI/CD pipelines [126], the following tools were used in at least one repository. Additionally, the linters and formatters named in the previous sections were also integrated in their respective pipelines.

- Docker container of the Flutter SDK [166]
- Docker container of the Rust compiler [137]
- Docker container of the Typst compiler [167]
- `junitreport` (Converts the output of `flutter test` into the JUnit format) [135]
- `cobertura` (Generates a Cobertura report from the Flutter code coverage) [168]
- `cargo-binstall` (Downloads pre-compiled Rust crates and installs them) [138]
- `cargo-tarpaulin` (Generates Rust code coverage report) [143]
- `cargo2junit` (Converts the output of `cargo test` into the JUnit format) [140]
- GitLab Time-Report (Generates charts from GitLab Time-Tracking data) [68]
- OpenSSH and related tools via the `openssh-client-common` and `openssh-client-default` Alpine Linux packages [169], [170]

A.3.4 AI Tools

AI tools were used as part of this Bachelor's thesis to optimise quality. All output from AI tools was manually validated by the team for accuracy and adjusted if necessary. The use of these tools is documented here in accordance with the guidelines for AI usage by OST [171].

The areas of application include:

- Structuring and formulating text in the Bachelor's thesis
- Prototyping features within the App
- Assisting with debugging and optimising program code
- Evaluating and improving architecture, design and framework decisions
- Writing test cases

The team has used the following AI tools:

- OpenAI ChatGPT [172]
- DeepL Write [173]
- JetBrains AI Assistant [174]
- Anthropic Claude [175]
- Google Gemini integrated in Android Studio [176]

B Usability Testing

B.1 Questionnaire

This chapter provides a comprehensive list of the tasks assigned to users and questions asked in the usability tests, as defined in [Section 5.2.5](#). As the testing was conducted in German, this chapter has also been written in German.

The follow-up questions are the questions defined in the original SUS [125], [177] and need to be answered using a 5-point Likert scale ranging from “Strongly Disagree” to “Strongly Agree” [11].

B.1.1 Warm-up Questions

- Wie alt bist du?
- Wie technik-affin würdest du dich auf einer Skala von 1-5 einschätzen?
- Hast du schon Erfahrungen mit Medikamenten-Kühlern gemacht?
- Was sind deine Erwartungen an einen Medikamenten-Kühler?

B.1.2 Task Walkthrough

Ausgangslage

Stell dir vor, du bereitest dich auf eine längere Reise vor. Du hast einen neuen Medikamentenkühler gekauft, der deine Medikamente sicher kühl halten soll. Die App dazu hast du gerade auf deinem Smartphone installiert.

Aufgabe 1 – mit Gerät verbinden

Nun möchtest du die App mit deinem Kühler verbinden. Dazu hast du die App geöffnet.

- Was siehst du nun und wie gehst du vor, um die Verbindung herzustellen?

Aufgabe 2 – Kühlung starten

Du möchtest vor der Abreise den Kühler schnell herunterkühlen. Finde und aktiviere den Schnellkühlmodus.

- Wird dir klar erklärt, dass dabei mehr Energie verbraucht wird?
- Würdest du anhand der angezeigten Informationen entscheiden können, ob und wann dieser Modus für dich sinnvoll ist?

Aufgabe 3 – Aktueller System-Status

Du möchtest sicherstellen, dass deine Medikamente im Kühler aktuell sicher gelagert werden. Verschaff dir deshalb einen Überblick über den aktuellen Status des Kühlers.

- Findest du die aktuelle Temperatur?
- Findest du den aktuellen Batteriestand?
- Findest du die geschätzte verbleibende Laufzeit?
- Wie schätzt du den aktuellen Status des Kühlers ein?
- Fällt dir auf, ob mögliche Probleme vorhanden sind? Falls ja, erkennst du, was das Problem ist?

Anmerkung: Diese Frage wird sowohl mit dem «OK» (grünes Schild), als auch mit dem «Attention» (gelbes Schild) und «Warning»-Status (rotes Schild) gestellt.

Aufgabe 4 – Trip-Funktion

Du möchtest gerne den Temperatur- und Akkuverlauf und die Durchschnittsdaten nur von dieser Reise betrachten können.

- Wie gehst du vor, um dies einzustellen?
- Gib dem Trip einen eigenen Namen.
- Kannst du erkennen, welche Daten während des Trips aufgezeichnet werden?

Aufgabe 5 – Historische Daten

Du bist bereits etwas länger unterwegs und möchtest den Verlauf der Temperatur und des Akkus ansehen.

- Wo siehst du solche Statistiken?
- Erkennst du die minimale und maximale Temperatur, die der Kühler seit dem Start deiner Reise aufgezeichnet hat? Siehst du den Durchschnitt?
- War bisher alles in Ordnung mit dem Zustand des Kühlers?

Aufgabe 6 – Synchronisation vergangener Daten

Stell dir vor, du warst heute unterwegs und hast den Kühler im Hotel gelassen. Nun bist du zurück und möchtest sicherstellen, dass alles immer noch in Ordnung ist.

- Was erwartest du, wenn du die App öffnest?
- Erkennst du den System-Status?
- Siehst du, dass ältere Daten nachgeladen wurden?

Aufgabe 7 – Temperaturbereich einstellen

Stell dir vor, dein Medikament darf maximal 6°C erreichen. Kannst du den Zieltemperaturbereich anpassen?

- Wie verständlich ist der Vorgang?

Aufgabe 8 – Alarme konfigurieren

Du möchtest eine Meldung bekommen, wenn der Akku 50% unterschreitet. Finde die Alarm-Einstellung für die Batterie.

- Kannst du einen eigenen Alarmwert hinzufügen?
- Ist der Ablauf verständlich?

B.1.3 Follow-up Questions

1. Ich werde die App gerne öfter benutzen
2. Ich fand die App unnötig kompliziert
3. Ich fand, dass die App einfach zu bedienen war
4. Ich glaube, ich würde die Unterstützung einer technisch begabteren Person benötigen, um diese App nutzen zu können
5. Ich fand, dass die verschiedenen Funktionen der App gut aufeinander abgestimmt waren
6. Ich fand, dass die App zu viele Unstimmigkeiten aufwies
7. Ich könnte mir vorstellen, dass die meisten Menschen den Umgang mit der App sehr schnell lernen würden
8. Ich fand die Handhabung der App sehr umständlich
9. Ich habe mich bei der Verwendung der App sehr sicher gefühlt
10. Ich musste erst eine Menge lernen, bevor ich mit der Verwendung der App loslegen konnte

B.2 Protocols

B.2.1 First Usability Test

The first usability test was conducted on 8 and 9 April 2026 with five participants and the wireframes depicted in [Section 3.6.1](#). The resulting feedback is listed in this chapter.

Name	Tilman
Age	26
Technical aptitude on a scale of 1 to 5	5
Prior Experiences	Travelling with a large cooling box and relying on hotel room fridges.
Expectations of the product	The cooler should be easy to use and not require constant monitoring. If any problems arise, the user should be notified. Otherwise, the cooler should simply work, so there is no need to worry constantly about the medication.
Feedback	– During the initial setup, it would be helpful to have a note stating that the cooler is not yet cold and that the medication should not be placed inside just yet. – The rapid cooling function could also be explained and possibly enabled directly in the first run wizard. It also needs to be made clearer that this is only useful for the initial cooling phase. – It would be useful to know whether a notification is sent when the cooling temperature is reached. – The cooler status should be visible further up the screen so it is visible at first glance. – A question was raised about connecting multiple devices to one app, as he needs more space for his medication. Alternatively, the participant would like a larger cooler. – The use of the trip function was not understood. – The current temperature on the home screen should link to the historical data. The icon for the statistics page was unclear. – A button to manually synchronise old data would be useful. – For the remaining battery life, it would make sense to display hours rather than days once it reaches around 48 hours. – The participant would like a display showing how long it takes for the battery to fully charge. This could also be shown as a push notification.
Duration	40 minutes
Score	87.5

Table 34: Protocol usability testing 1 with Tilman

Name	Linnéa
Age	29
Technical aptitude on a scale of 1 to 5	3
Prior Experiences	None
Expectations of the product	The device should be compact, sustainable, reusable and practical for travelling with. The app should be user-friendly and clearly indicate whether the cooling function has worked.

Feedback	– It was unclear whether the device needed to be plugged in to start the rapid cooling mode. A notification when cooling is complete would be useful. – The advice provided in the first run wizard was found to be useful. – Warnings and error messages should clearly communicate the problem and its resolution. – The traffic light system for warnings was found to be very intuitive. – The display indicating when the cooler had left the safe temperature range was considered helpful. – The trip function was not understood immediately. – It was expected that there would be a details page accessible from the status infobox, indicating whether the status had changed recently. – The app was considered to be simple with a good amount of features.
Duration	30 minutes
Score	92.5

Table 35: Protocol usability testing 2 with Linnéa

Name	Tanja
Age	30
Technical aptitude on a scale of 1 to 5	4
Prior Experiences	A cooling bag was used to transport the medication from the doctor's office to the home.
Expectations of the product	The app must display the temperature and alert the user in the event of an error. The participant has identified two use cases: transporting the medication home from the doctor's office in its original packaging which may be unwieldy, and transporting the medication while travelling.
Feedback	– The description of the rapid cooling mode is too long and the button has been overlooked. Either the button should be bigger, or it should be coloured. Alternatively, the text could only be displayed only after the button has been tapped on, at least the first time. Another option would be to explain the mode in the onboarding process with an illustration. – It was not clear that there are two different cooling modes and what would happen if the rapid cooling mode was not enabled. – The estimated remaining battery life was not found immediately. – The position of the information and warning boxes should be consistent. – It is unclear how long the status has been as it is currently displayed. – All important information should be visible on the home screen with no need to scroll. – It would be more intuitive if the message "Last updated at x o'clock" said "Last updated x minutes ago". – The purpose of the Trip feature was not entirely clear. It would be helpful to include an explanation during the onboarding process, for example by comparing it to fitness apps. – The difference between the two types of alarms was not clear. – It was considered useful to offer a choice of different time periods in the statistics.
Duration	35 minutes
Score	77.5

Table 36: Protocol usability testing 3 with Tanja

Name	Emma
Age	31
Technical aptitude on a scale of 1 to 5	4
Prior Experiences	Insulation bags have already been used, but nothing electronic.
Expectations of the product	The product should be easy to buy, setup and use. It should provide enough space and function reliably.
Feedback	– During setup, it would be helpful to have icons for Bluetooth, Location, etc. – The text in the notes is too small. – The fields should be labelled when setting the initial temperature range. – An indication when rapid cooling is complete would be useful. – The warnings are unclear and do not specify the required action. – It would be helpful to include a forecast of the battery level and temperature in the statistics. How long will the device remain cool enough via passive cooling once the battery is empty? – The purpose of the "Trip" function was not understood. – It should be communicated clearer when the app is reconnecting. The reconnection should happen automatically. – An icon on the home screen indicating that the cooling system is currently active would be useful. – The participant liked the app because it isn't too cluttered.
Duration	40 minutes
Score	87.5

Table 37: Protocol usability testing 4 with Emma

Name	Joe
Age	62
Technical aptitude on a scale of 1 to 5	2
Prior Experiences	Has already used cooling bags filled with gel. These are always wet, so the handling was considered impractical.
Expectations of the product	Has to be easy to seal and have a rugged build. When the cooler falls down, it shouldn't break.
Feedback	– The temperature range setting was not recognised. The participant wanted to select one of the numbers for his setting. – The rapid cooling mode was not understood. – The information provided in the attention status box was not regarded as a state where action should be taken, unlike the warning status. – The warnings should be tappable to get more information if needed. – The icon for statistics is unclear. – The disconnection state was not recognised as such. – The connection should be restored automatically after disconnection. – It is unclear when old data is being reloaded. – The participant would like a video tutorial.
Duration	1 hour

Score	65
--------------	----

Table 38: Protocol usability testing 5 with Joe

B.2.2 Second Usability Test: Expert Evaluation

The expert evaluation with Markus Stolze on the current implementation of the app was conducted on 28 May 2026. The resulting feedback is listed in this chapter.

App-specific Feedback	– It was unclear where to connect to the cooler. Tapping on the box containing the Bluetooth information was not obvious. It would be helpful to have a button to reconnect, either inside the connection box or below. – The text showing the remaining battery time is too small or too grey, and is probably not WCAG Level AA compliant [154]. – The contrast should generally be improved, especially for colours with low opacity. – Information should not only be communicated through colour. An icon or similar should be present in addition to the colour. – If there is a warning or error message, only the message box should be red, not all elements on the screen. It would be better if the element that caused the error had an error indicator in form of an icon. – It is unclear what action the user should take when a status message appears. – The first run wizard is important and should definitely be included. – The informal “du” wording was questioned. – It took a long time for the safe range area in the chart to be recognised as such. – If the cooler is out of range, the chart indicates this with a red line. This is not sufficient for users with a red-green colour blindness. – A statistic showing how much time the temperature was inside and outside the safe range would be useful. – It should be clarified what the chart looks like when the cooler’s battery has been empty for a certain amount of time and therefore no data is present in that time frame. – The tooltip in the chart is not discoverable. When the chart is opened for the first time, one tooltip should be visible to inform the user of this feature. – Including a guide as described in the industry partners mockups is a good idea. – The “last updated” timestamp should show the exact time rather than “2 minutes ago”. – It would be useful to have a video or images explaining how to activate Bluetooth pairing on the cooler. – Language support should take right-to-left languages into account. – The app should respect OS boundaries, such as the iOS Dynamic Island. – For text fields it is important to ensure that the keyboard does not obscure the input field. – If it is possible for multiple statuses to overlap (e.g. “fast cooling” and “charging”), it should be clearly defined which status takes precedence. – To avoid any confusion, the “battery” infobox should be renamed “cooler battery”. – If the cooler is not connected, the battery percentage symbol could be omitted.
General Feedback	– In usability tests, the task descriptions should not contain terms that appear directly in the UI. – If multiple usability tests are conducted at different stages of the development process, new participants should be used each time.

	– It would be useful to discuss with end users after what amount of time they start to worry when their medication is no longer cooled, and whether there are any other issues they would like to be informed about. – In-app user feedback to the manufacturer is very important. Otherwise, users may leave negative reviews on the App's store pages.
Duration	45 minutes
Score	97.5

Table 39: Protocol usability test expert evaluation

C External Code Reviews

This chapter contains the full content of the code reviews carried out by external experts. All code reviews were conducted via email in German; consequently, this chapter is also written in German.

C.1 Code Review by Clemens Meier

The Code Review by Clemens Meier was conducted at the end of April after the initial MVVM implementation, with the aim of gathering feedback on adherence to MVVM principles.

C.1.1 Initial Response, 20.04.2026

Ich schaue morgen mal rein. Habe selbst eher wenig Ahnung von Flutter, aber Architekturmässig kann ich vielleicht ein paar Inputs geben.

Zu den Fragen:

Wenn wir den Mittelwert von verschiedenen Sensoren berechnen müssen, gehört das ins ViewModel oder ins Repository?

Ganz streng genommen wäre ein Service Layer noch dazwischen und das würde dort hereingehören. Grundsätzlich ist das ViewModel sozusagen ein Headless UI -> es ist also nur UI State und ruft Services auf. Daher ist das Repository wahrscheinlich die bessere Lösung als das ViewModel, obwohl dann der Name Repository nicht mehr ganz gerechtfertigt ist. Manchmal macht man auch Service Klassen und erlaubt trotzdem auch direkt auf die Repos zu gehen und sogar direkt Entities anzuzeigen, sofern keine Logik dazwischen liegt. Das nennt man Durchgriff und ist streng genommen nicht mehr echtes MVVM. Ist aber schon ein bisschen eine Architekturverletzung. Aber ich muss es mir mal anschauen, was ihr genau gemacht habt.

Ist das Coupling zwischen dem live telemetry repository und dem connection repository in Ordnung?

Normalerweise sollten die Repositories ja unabhängig voneinander sein, aber im connection repository wird der State der Bluetooth-Verbindung verwaltet, deshalb muss das live telemetry repository fast davon abhängen. Ja, Unabhängigkeit von Repos ist schon einigermaßen wichtig. Aber vielleicht macht es das dort auch überkompliziert. Oder ihr könnt vielleicht auch dort einen Service dazwischen schalten.

C.1.2 Code Review, 22.04.2026

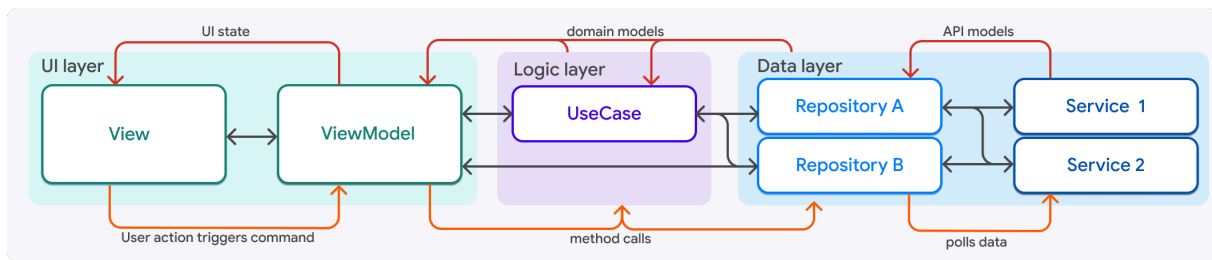
Habe mir das Repo angeschaut und mich nicht nur auf MVVM sondern auch auf die sonstige Architektur konzentriert.

Allgemein zur Umsetzung von MVVM. Der Anfang ist nicht schlecht. Auch Flutter empfiehlt, dass man einen Data Layer (Repos und Services) trennt vom UI Layer für einfache Projekte. Es wäre aber schön, wenn ihr noch einen Schritt weiter geht.

Flutter schreibt aber auch dies:

Apps with complex logic might also have a logic layer that sits in between the UI layer and data layer. This logic layer is commonly called the domain layer. The domain layer contains additional components often called interactors or use-cases. The domain layer is covered later in this guide.

— [34]



Euer Projekt ist zwar überschaubar, und ein vollständiger Domain Layer wäre streng genommen Over-engineering, aber für eine Bachelorarbeit ist es absolut sinnvoll, das zu üben. Ich würde dafür in der Bewertung nur Pluspunkte geben, sofern ihr in der Dokumentation begründet, dass ihr das bewusst gemacht habt, um saubere Architektur zu lernen. Interactors und UseCases machen Apps ausserdem deutlich besser testbar.

Das Pendant zu Interactors sind in .NET MAUI übrigens mit Commands implementiert, also Objekte, die ans ViewModel gebunden werden, aber gleichzeitig keine Abhängigkeit vom UI Framework haben. Das entkoppelt das ViewModel von den Services darunter und gleichzeitig das ViewModel vom View Framework.

also anstatt (Ich hoffe ich konnte Flutter genug gut lesen...):

```
1 onTap: () async {
2   await vm.connectCommand.execute(device);
3   if (context.mounted) Navigator.pop(context);
4 }
```

eher:

```
1 TextButton(
2   onPressed: vm.stopScanCommand.canExecute
3     ? () => vm.stopScanCommand.execute(null)
4     : null,
5   child: vm.stopScanCommand.isRunning
6     ? const CircularProgressIndicator()
7     : Text(context.l10n.cancel),
8 )
```

Aber ich habe auf die Schnelle nicht herausgefunden, ob Flutter so ein sauberes Konzept hat wie ICommand in WPF/MAUI.

Liste für mögliche Verbesserungen (müssen nicht umgesetzt werden, aber machen den Code «Enterprise Ready» und für Bonuspunkte solltet ihr in der Doku wenigstens erwähnen, dass man das hätte tun können → Begründung warum nicht wäre z.B. Aufwand zu gross und Zeitknappheit).

- In main.dart könntet ihr mit dem DI Container GetIt arbeiten und stärker gegen Interfaces implementieren. Das ermöglicht automatische Auflösung von Abhängigkeiten ausserhalb des Widget Trees und vereinfacht das Mocken in Tests erheblich.
- Domain Layer mit UseCases
Mögliche Implementation von UseCases: → ScanForDevicesUseCase, ConnectToCoolerUseCase, DisconnectFromCoolerUseCase, MonitorCoolerStatusUseCase
- \lib\ui\models\user_message.dart
UserMessage (UI Layer) importiert CoolerStatus (Domain/Data Layer) und definiert einen coolerStatus-Getter – die Abhängigkeit zeigt in die falsche Richtung. Das Mapping von CoolerStatus → UserMessage gehört ins ViewModel. UserMessage selbst sollte keine Abhängigkeit auf den Data Layer haben.

- CoolerStatusRepository - kein echtes Repository
CoolerStatusRepository kapselt keine eigenständige Datenquelle – es leitet State ab, der bereits in ConnectionRepository und LiveTelemetryRepository vorhanden ist. Die Ableitung von CoolerStatus gehört in einen MonitorCoolerStatusUseCase, der beide Repositories kombiniert.
- IBleService -> Services nach Clean Architecture sind zustandslos (reiner Wrapper um eine externe API). IBleService verwaltet jedoch StreamController, StreamSubscription und gecachte Daten, das ist per Definition ein Repository. Die Klasse sollte in data/repositories/ liegen und entsprechend umbenannt werden.
- Bei HomeViewModel State Transformation ausbauen.
Das ViewModel sollte der View fertig aufbereiteten UI-State liefern, anstatt rohe Domain-Typen wie CoolerStatus zu exponieren, die der View dann selbst per switch interpretieren muss. Beispiel: `UserMessage get statusMessage` statt `CoolerStatus get status`.
- Das Widget SettingsPage hat nach kurzem Überfliegen keinen State und müsste wahrscheinlich nicht Stateful sein.
- HomePage: `MaterialPageRoute<void>(builder: (_) => const HistoryPage())`, Navigation ist in der View -> sollte eher im ViewModel sein (dort über eine Abstraktion aufgerufen werden!), damit das ViewModel alleine bedient werden kann. Dies wird aber nicht immer strikt umgesetzt und ist nur eine sehr leichte MVVM Verletzung. Allerstriktes MVVM kennt ihr aus der UIP-Vorlesung. Ein Kommandozeilenprogramm könnte mit dem gleichen ViewModel funktionieren ;P
- Errorhandling bei Verbindungsproblemen könnte verbessert werden.
- Die aktuelle Struktur vermischt Verantwortlichkeiten
Model und Repository nicht vermischen, Mapper Logik in eigene Bereiche platzieren
- Resource Lifecycle beachten (dispose) -> LiveTelemetryRepository, CoolerStatus -> Stream wird nicht geschlossen

Vorschlag:

Data	Domain	UI
– data/services (stateless, externe apis) – data/repositories (zustandbehaftet) – data/models (reine datenobjekte /dtos)	– domain/use_cases (Interactors) – domain/models (fachliche domain-objekte) -> Bsp TelemetryData, CoolerStatus	– ui/pages (views) – ui/viewmodels – ui/widgets (wiederverwendbare UI-Komponenten) – ui/mappers (z.b. Mapping auf UserMessage)

Allgemeine Bemerkung: Macht ein bisschen mehr Entwicklerdoku in den .md Files. ☑

C.1.3 Clarification Response, 23.04.2026

Use Cases: Sind sicher sinnvoll, wir haben aber beschlossen, dies aufgrund Zeitmangels nicht mehr umzusetzen. Bei einer App von unserem Umfang dürfte es auch reichen, wenn wir auf diese verzichten. → Das ist gut nachzuvollziehen. Schreibt es aber auch in die Doku. Das kann Pluspunkte geben, weil ihr damit zeigt, dass ihr wisst was ihr tut. ☑

Dagegen dürften wohl Commands ein sinnvolles Mittelding zu Use Cases darstellen. Da müssen wir uns aber auch nochmal genauer einlesen, weil wir aus der Doku auch nicht ganz schlau geworden sind <https://docs.flutter.dev/app-architecture/design-patterns/command>

→ Ist sicher eine gute Massnahme um MVVM konsequenter umzusetzen, ja. Commands helfen, MVVM sauber umzusetzen, weil sie die Interaktionslogik kapseln. MVVM = Keine Logik in der View ausser Darstellung und Binding; das ViewModel enthält Zustand und Interaktion, ggf. über Commands, und erhält seine Abhängigkeiten per DI; Domain- und Datenzugriffslogik liegen in Model, Repository oder Use Case.

Was wäre der Vorteil eines DI Containers gegenüber der Provider-Lösung? Diese wird so von Flutter empfohlen <https://docs.flutter.dev/app-architecture/case-study/dependency-injection>

→ Ich müsste mir Flutter und die Provider Lösung genauer anschauen, um das besser zu beantworten. Ich habe das geschrieben, weil ich vermute, dass die Auflösung beim Flutter Provider stark an UI Widgets gekoppelt ist und sich wahrscheinlich weniger um den Life Cycle kümmert als das DI Container normalerweise tun. Aber du hast Recht, dieser Nachteil hat bei euch nicht viel Relevanz.

CoolerStatusRepository basiert momentan hauptsächlich auf den Verbindungsstatus des Kühlers, wird aber später noch von anderen Faktoren beeinflusst, beispielsweise ob der Kühler im vom User angegebenen Temperaturbereich ist. Wie handhabt man so einen globalen State ohne Use Cases?

→ Hier läuft ihr eben gerade in die Architekturentscheidung rein. Bewusst Stateful Services erlauben (Namesgebung etwa StatusCoordinator oder AppStateManager – Manager und Coordinator sind zwar recht generisch, implizieren aber State oder Logik was hilft zu begreifen was hier los ist), State als UI-State behandeln (nur wenn es wirklich flüchtiger UI-State ist!) oder UseCases einführen (sauberste Lösung).

IBleService ist kein Service im MVVM-Kontext, sondern bezieht sich auf die vom Bluetooth Low Energy (BLE) angebotenen «Services» (interne Gruppierung von Daten). Es soll ein Mapper-Interface zwischen den BLE-Daten-Repositories (LiveTelemetryRepository) und den entsprechenden Model-Klassen (TelemetryData) darstellen. Ebenfalls bietet es Default-Implementationen für die BLE-Daten-Repositories an. Aber es wäre wahrscheinlich sinnvoller, es von «models» nach «repositories» zu verschieben.

→ OK, das klingt sinnvoll. Ich würde nur die Benennung und Platzierung (in repositories ist ganz ok) korrigieren, um Architektur-Missverständnisse zu vermeiden.

Sollte in der View auch auf simple Logik wie beispielsweise null checks verzichtet werden? Das würde aber ja das ViewModel «unnötig» aufblähen, nicht?

→ Simple Darstellungslogik ist vollkommen ok. (null checks, if, switch, conditional layouting, animation, usw)

Wie reicht man Fehler in einem Service (z.B. Bluetooth-Verbindungs-Fehler im BluetoothService) am besten bis zur View durch?

→ Eine gute Lösung ist oft ein Mediator Pattern oder ein Message Bus für Messages die angezeigt werden sollen. Ich habe oft eine ObservableCollection (in .NET) hinter ein einfaches addMessage interface gekapselt, diese Collection direkt bei einem ViewModel referenziert und auf die View gebunden. Natürlich könnte man auch mehrere Collections dort rein packen und diese an unterschiedlichen Orten anzeigen lassen. Unten durch kennen die Komponenten einfach nur das Interface addMessage (vielleicht auch removeMessage) und sind damit unabhängig vom UI Framework. Im UI kann man ganz einfach bestimmen, unter welchen Bedingungen die Messages wieder weg sollen. Ihr müsst natürlich immer noch entscheiden, was denn bei welcher Exception sonst noch passieren soll in den unteren Schichten.

D Glossary and List of Abbreviations

D.1 Glossary

This chapter lists project-specific terms and their definitions.

APK	File format used by Android to package apps for distribution and installation. It is in essence a ZIP archive with a set structure, containing the app code and its metadata.
Artifact	The output of a GitLab CI/CD job that should be stored, such as build output or test reports. Subsequent jobs can download artifacts from earlier jobs and utilise them.
Crate	All Rust code is grouped into a crate. A crate is the smallest unit that the Rust compiler takes into account. It is compiled into either a binary or a library. External libraries are often also referred to as crates. Publicly released crates are often hosted on crates.io, the official repository by the Rust Foundation.
Characteristic	In BLE, data is stored in characteristics which provide the actual value and associated metadata. This includes information such as manual polling permissions and whether the characteristics can be subscribed to in notify mode (sends acknowledgments of data) or indicate mode (does not send acknowledgments).
CRUD	The workflows of creating, reading, updating and deleting entries. Most often used in a relational database context.
Dependency Injection	Object-oriented design pattern in which elements external to the object are passed to it at runtime instead of instantiated in the object directly. The pattern promotes loose coupling and improved code testability.
Diff	Short for difference. A diff showcases the content changes in one or multiple files from a "original" state to a "changed" state. While diffs can be used to track arbitrary changes, they are most often used in version control to show changes from one point in time to another (e.g. changes of a single commit, differences between branches).
Dirty State	In object-oriented programming, certain classes may have class invariants, i.e. implicit rules for how the class should behave. For example, a list with a <code>length</code> property should always return the number of elements in the list. If there is a way to modify the number of elements in the list without affecting the <code>length</code> property, the class invariant is violated, resulting in a dirty state of the list object.
Epic	A larger requirement for the system (for example, a larger feature) that can be divided into several user stories.
Milestone	A defined point in time in a project at which certain goals or results are to be achieved.
Mocking	The process of providing a fake variant of a dependency for testing purposes. These mocks can then provide set results when their functions are called and test cases can verify how often a certain function of a mock was called. Often closely tied to the Dependency Injection pattern.
MVP	An initial version of the product with just enough functionality to be useful to customers.
Push	The process of sending the changes from a local Git repository to a remote repository. It usually refers to sending commits created on a branch to a corresponding branch of a remote repository hosted on a code forge, such as GitHub or GitLab. A push may trigger further actions such as executing a CI/CD pipeline.

Toolchain	In the Rust context, this refers to an installation of the Rust compiler and additional tooling like the Cargo packet manager for a certain processor architecture and operating system.
User Story	A requirement for the system from the user's perspective. Describes what users want to do and how they benefit from it.

D.2 List of Abbreviations

The acronyms used in the thesis are written out in full in this chapter.

APK	Android Package
BLE	Bluetooth Low Energy
C4	Context, Containers, Components and Code model
CBOR	Concise Binary Object Representation
CI/CD	Continuous Integration / Continuous Delivery
CLI	Command Line Interface
CRUD	Create, Read, Update, Delete
CW	Calender week
DI	Dependency Injection
ECTS	European Credit Transfer System
FR	Functional requirements
GDPR	General Data Protection Regulation of the European Union
IDE	Integrated Development Environment
M	Milestone
MR	Merge request
MVP	Minimum viable product
MVVM	Model-View-ViewModel
nFADP	New Federal Act on Data Protection in Switzerland
NFR	Non-functional requirements
OST	Eastern Switzerland University of Applied Sciences
RUP	Rational Unified Process
SSH	Secure Shell
SCP	Secure Copy
SUS	System Usability Scale
UI	User Interface
US	User Story
UUID	Universally Unique Identifier
UX	User Experience

E List of Tables

Table 1	Key findings from the market analysis	4
Table 2	List of user stories mapped to Epics	9
Table 3	Description of NFR 1 "Maintainability – Testability"	10
Table 4	Description of NFR 2 "Interaction Capability – Learnability"	10
Table 5	Description of NFR 3 "Functional Suitability – Functional Correctness"	11
Table 6	Description of NFR 4 "Reliability"	11
Table 7	Description of NFR 5 "Safety – Hazard Warning"	12
Table 8	Description of NFR 6 "Maintainability - Analysability"	12
Table 9	Overview of the pre-defined and custom BLE services offered by the cooler	14
Table 10	Implemented feedback from external code review	70
Table 11	Not implemented feedback from external code review	71
Table 12	Status of the acceptance criteria for user story 1	72
Table 13	Status of the acceptance criteria for user story 2	73
Table 14	Status of the acceptance criteria for user story 3	74
Table 15	Status of the acceptance criteria for user story 4	74
Table 16	Status of the acceptance criteria for user story 5	75
Table 17	Status of the acceptance criteria for user story 6	76
Table 18	Status of the acceptance criteria for user story 7	77
Table 19	Status of the acceptance criteria for user story 8	78
Table 20	Status of the acceptance criteria for user story 9	78
Table 21	Status of the acceptance criteria for user story 10	79
Table 22	Status of the acceptance criteria for user story 11	80
Table 23	Status of the acceptance criteria for user story 12	80
Table 24	Summary of the main functional requirement fulfilment	82
Table 25	Summary of non-functional requirement fulfilment	88
Table 26	Risk matrix of the project	92
Table 27	Description of risk R1 "No app development experience"	92
Table 28	Description of risk R2 "No experience with Dart and Flutter"	93
Table 29	Description of risk R3 "Difficulties with Bluetooth LE communication"	93
Table 30	Description of risk R4 "Differing project expectations between the industry partner and the advisors"	93
Table 31	Description of risk R5 "Project management overhead due to communication with the industry partner"	94
Table 32	Description of risk R6 "Difficulties in creating an intuitive UI for the diverse target audience"	94
Table 33	Description of risk R7 "Absence of a team member"	94
Table 34	Protocol usability testing 1 with Tilmann	99
Table 35	Protocol usability testing 2 with Linnéa	99
Table 36	Protocol usability testing 3 with Tanja	100
Table 37	Protocol usability testing 4 with Emma	101
Table 38	Protocol usability testing 5 with Joe	101
Table 39	Protocol usability test expert evaluation	102

F List of Figures

Figure 1	First run wizard of the OHealth app, guiding the user through connecting with their wearable	2
Figure 2	The four main screens in the OHealth app	3
Figure 3	Detail page with statistics in the OHealth app	3
Figure 4	Chart in OHealth in landscape mode	4
Figure 5	Domain model of the problem domain	16
Figure 6	System Context Diagram showing the main systems and its interactions with external entities	17
Figure 7	Container Context Diagram showing the different containers within the systems	18
Figure 8	Component Context Diagram for the backend	19
Figure 9	Backend database schema	19
Figure 10	Local database schema	23
Figure 11	Screenshots of the design specification pages	25
Figure 12	Wireframes of the first run wizard, guiding the user through setting up the app	25
Figure 13	Home page with different statuses: Connected, OK, Warning, Attention, Not connected .	26
Figure 14	Trip page in different statuses, history page and settings	26
Figure 15	The MVVM architecture as described by Flutter. Source: [34]	27
Figure 16	The widgets created in Listing 4 after three clicks	27
Figure 17	Flutter MVVM with optional Domain/Logic Layer. Source: [38]	28
Figure 18	Overview of the Bluetooth connect/disconnect functionality, showing the separation into MVVM layers	30
Figure 19	App requesting permissions and asking the user to switch on Bluetooth	30
Figure 20	App scanning for Bluetooth devices, showing a cooler and other non-cooler devices	31
Figure 21	Overview of the Live Data processing implementation, showing the separation into MVVM layers	34
Figure 22	Home Screen in its different states	36
Figure 23	Overview of the historical synchronisation implementation, showing the separation into MVVM layers	37
Figure 24	Overview of the settings implementation, showing the separation into MVVM layers	41
Figure 25	Overview of the backend telemetry implementation, showing the separation into MVVM layers	44
Figure 26	Image of the app running on an iPhone XR	62
Figure 27	Connecting to a cooler	72
Figure 28	Home screen in different states: disconnected, connected with charging and discharging battery, battery warnings, battery errors and temperature errors.	73
Figure 29	History synchronisation process	74
Figure 30	History page with a line chart showing temperatures within and outside the target range, including a tooltip view, and a bar chart shown both with and without a tooltip.	76
Figure 31	The trips page shows a coming soon text	77
Figure 32	Settings page displaying error and success messages, as well as the updated interface after a successful temperature range change.	78
Figure 33	The pages of the app in dark mode	81
Figure 34	Final app code coverage report	83
Figure 35	Final backend code coverage report	83
Figure 36	Project plan with work areas, sprints and RUP phases	90

G List of Listings

Listing 1	Example JSON value stored in the telemetry readings table	20
-----------	---	----

Listing 2	Example query for exporting telemetry readings of one cooler within a time range	21
Listing 3	Example query for exporting aggregated telemetry data across all coolers within a time range	21
Listing 4	Simplified widget example: A <code>Column</code> contains a <code>Text</code> and an <code>ElevatedButton</code> . When the button is pressed, the counter displayed in the <code>Text</code> widget increases. Note that the code for state management has been omitted.	27
Listing 5	The directory structure used in the app	28
Listing 6	Flutter dependency injection example: <code>ConnectionRepository</code> reads the already initialized <code>BluetoothService</code> from the <code>BuildContext</code>	29
Listing 7	Example of a <code>BleCharacteristicsId</code> implementation	31
Listing 8	Connect to a BLE device in <code>ConnectionRepository</code>	32
Listing 9	Signature of <code>updateModel</code> in the <code>IBleData</code> interface	34
Listing 10		34
Listing 11		35
Listing 12	Parsing of the BLE Battery Service's Battery Level Status characteristic	35
Listing 13	Subscription handling for live data updates in the <code>HomeViewModel</code>	36
Listing 14	Starting historical synchronisation by sending the latest stored log counter and triggering the sync control point	38
Listing 15	Reassembling chunked BLE packets into a complete CBOR payload	39
Listing 16	Decoding a CBOR payload and mapping integer keys to a <code>HistoricalLogPoint</code>	40
Listing 17	Saving the temperature range by first writing it to the cooler and then updating the local app settings	42
Listing 18	Validating the temperature range before executing the save command	43
Listing 19	Backend telemetry synchronisation using the latest backend log counter and locally stored readings	45
Listing 20	Overview of the Rust crates generated by Gerust	46
Listing 21	Changeset and insert function for a <code>TelemetryReading</code>	48
Listing 22	Demonstrating static and dynamic dispatch in Rust	53
Listing 23	Difference in error output when using a raw value versus a dedicated matcher in Flutter tests	57
Listing 24	Unit test from <code>ConnectionRepository</code> showcasing Mockito mocks and stubs	57
Listing 25	Example Widget Test of the <code>HomePage</code> widget	59
Listing 26	An API test of the <code>/latest_counter</code> route in the backend	60
Listing 27	The changeset for the <code>TelemetryReadings</code> database table with test data generator and validation attributes	61
Listing 28	Example request for retrieving filtered telemetry readings of a specific cooler	79
Listing 29	Example response containing filtered telemetry readings and cooler metadata	80

H References

- [1] The Flutter team, 'Flutter - Build apps for any screen'. Accessed: Mar. 26, 2026. [Online]. Available: <https://flutter.dev/>
- [2] The tokio Team, *Axum: HTTP routing and request-handling library for Rust that focuses on ergonomics and modularity*. Accessed: Mar. 29, 2026. [Online]. Available: <https://github.com/tokio-rs/axum>
- [3] Gin-Gonic, *Gin: Gin is a high-performance HTTP web framework written in Go. It provides a Martini-like API but with significantly better performance—up to 40 times faster—thanks to httprouter. Gin is designed for building REST APIs, web applications, and microservices*. Accessed: Mar. 29, 2026. [Online]. Available: <https://github.com/gin-gonic/gin>
- [4] 'Modulbeschreibung Bachelor-Arbeit Informatik'. OST - Ostschweizer Fachhochschule, Nov. 27, 2025. Accessed: Feb. 26, 2026. [Online]. Available: https://studien.ost.ch/allModules/41626_M_BAI21.html
- [5] HeyTap, 'OHealth -- Apps on Google Play'. Accessed: July 04, 2026. [Online]. Available: https://play.google.com/store/apps/details?id=com.heytap.health.international&hl=en_GB
- [6] 'HeyTap - Official Website'. Accessed: July 04, 2026. [Online]. Available: <https://brand.heytap.com/en/index.html>
- [7] Agile Alliance, 'What is User Story Template?'. Accessed: Mar. 02, 2026. [Online]. Available: <https://agilealliance.org/glossary/user-story-template/>
- [8] ISO Committee, 'Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model', *ISO/IEC*, no. 25010:2023. Nov. 2023. Accessed: Feb. 27, 2026. [Online]. Available: <https://www.iso.org/standard/78176.html>
- [9] arc42.org, 'Update on ISO 25010, version 2023'. Accessed: Feb. 27, 2026. [Online]. Available: <https://quality.arc42.org/articles/iso-25010-update-2023>
- [10] Scott O. Bradner, 'Key words for use in RFCs to Indicate Requirement Levels', *Request for Comments*, no. 2119. RFC Editor, Mar. 1997. doi: [10.17487/RFC2119](https://doi.org/10.17487/RFC2119).
- [11] UXArmy, 'System Usability Scale (SUS): Measure Usability with Confidence'. Accessed: Mar. 02, 2026. [Online]. Available: <https://uxarmy.com/blog/system-usability-scale-sus/>
- [12] Nordic Semiconductors Developer Academy, 'Services and Characteristics -- Bluetooth Low Energy Fundamentals'. Accessed: June 10, 2026. [Online]. Available: <https://academy.nordicsemi.com/courses/bluetooth-low-energy-fundamentals/lessons/lesson-4-bluetooth-le-data-exchange/topic/services-and-characteristics/>
- [13] Nordic Semiconductors Developer Academy, 'GATT operations -- Bluetooth Low Energy Fundamentals'. Accessed: June 10, 2026. [Online]. Available: <https://academy.nordicsemi.com/courses/bluetooth-low-energy-fundamentals/lessons/lesson-4-bluetooth-le-data-exchange/topic/gatt-operations/>
- [14] Bluetooth Special Interest Group, 'GAP service and characteristics for GATT Server -- Generic Access Profile'. Accessed: June 10, 2026. [Online]. Available: <https://www.bluetooth.com/wp-content/uploads/Files/Specification/HTML/Core-54/out/en/host/generic-access-profile.html#UUID-8942f34d-80a7-1f7f-8f92-c8ee39aed116>
- [15] 'Device Information Service 1.2'. in Bluetooth Service specification. Accessed: June 10, 2026. [Online]. Available: <https://www.bluetooth.com/specifications/specs/dis-1-2/>
- [16] 'Battery Service 1.1', *Bluetooth Service specification*. Accessed: June 10, 2026. [Online]. Available: <https://www.bluetooth.com/specifications/specs/bas-1-1/>

- [17] Google for Developers, 'Android Studio'. Accessed: Mar. 26, 2026. [Online]. Available: <https://developer.android.com/studio?hl=en>
- [18] Android Developers, 'Advanced emulator usage | Android Studio'. Accessed: May 20, 2026. [Online]. Available: <https://developer.android.com/studio/run/advanced-emulator-usage?hl=en>
- [19] 'Build and release an iOS app', *Flutter docs*. The Flutter team. Accessed: May 04, 2026. [Online]. Available: <https://docs.flutter.dev/deployment/ios>
- [20] F. Mehta, 'SE Practices 1: 02 - Domain Modelling', OST - Ostschweizer Fachhochschule, Sept. 2024.
- [21] S. Brown, 'The C4 Model for Software Architecture', June 25, 2018, *InfoQ*. Accessed: Mar. 24, 2026. [Online]. Available: <https://www.infoq.com/articles/C4-architecture-model/>
- [22] 'System context diagram', *C4Model.com*. Accessed: May 01, 2026. [Online]. Available: <https://c4model.com/diagrams/system-context>
- [23] 'Software system', *C4Model.com*. Accessed: May 01, 2026. [Online]. Available: <https://c4model.com/abstractions/software-system>
- [24] 'Container diagram', *C4Model.com*. Accessed: May 01, 2026. [Online]. Available: <https://c4model.com/diagrams/container>
- [25] 'Container', *C4Model.com*. Accessed: May 01, 2026. [Online]. Available: <https://c4model.com/abstractions/container>
- [26] 'Component diagram', *C4Model.com*. Accessed: June 12, 2026. [Online]. Available: <https://c4model.com/diagrams/component>
- [27] 'Architecture - Main Choices'. Accessed: May 06, 2026. [Online]. Available: <https://gerust.rs/docs/architecture/#main-choices>
- [28] 'PostgreSQL: The World's Most Advanced Open Source Relational Database'. Accessed: May 06, 2026. [Online]. Available: <https://www.postgresql.org/>
- [29] 'JSON Types', *PostgreSQL Documentation*. Accessed: May 06, 2026. [Online]. Available: <https://www.postgresql.org/docs/current/datatype-json.html>
- [30] Tiger Data, *TimescaleDB: A time-series database for high-performance real-time analytics packaged as a Postgres extension*. Accessed: Apr. 15, 2026. [Online]. Available: <https://github.com/timescale/timescaledb>
- [31] 'Flutter persistence cookbook'. in *Flutter docs*. The Flutter team. Accessed: May 09, 2026. [Online]. Available: <https://docs.flutter.dev/cookbook/persistence>
- [32] 'Persist data with SQLite'. in *Flutter docs*. The Flutter team. Accessed: May 09, 2026. [Online]. Available: <https://docs.flutter.dev/cookbook/persistence/sqlite>
- [33] Figma, 'Figma: The Collaborative Interface Design Tool'. Accessed: May 12, 2026. [Online]. Available: <https://www.figma.com/>
- [34] 'Guide to app architecture: The recommended way to architect a Flutter app.', *Flutter docs*. The Flutter team. Accessed: Mar. 30, 2026. [Online]. Available: <https://docs.flutter.dev/app-architecture/guide>
- [35] 'Layouts in Flutter', *Flutter docs*. The Flutter team. Accessed: May 17, 2026. [Online]. Available: <https://docs.flutter.dev/ui/layout>
- [36] 'Differentiate between ephemeral state and app state', *Flutter docs*. The Flutter team. Accessed: May 18, 2026. [Online]. Available: <https://docs.flutter.dev/data-and-backend/state-mgmt/ephemeral-vs-app>

- [37] 'Communicating between layers', *Flutter docs*. The Flutter team. Accessed: May 11, 2026. [Online]. Available: <https://docs.flutter.dev/app-architecture/case-study/dependency-injection>
- [38] 'Optional: Domain Layer - Guide to app architecture', *Flutter docs*. The Flutter team. Accessed: May 26, 2026. [Online]. Available: <https://docs.flutter.dev/app-architecture/guide#optional-domain-layer>
- [39] 'Package structure -- Architecture case study', *Flutter docs*. The Flutter team. Accessed: June 10, 2026. [Online]. Available: <https://docs.flutter.dev/app-architecture/case-study#package-structure>
- [40] 'Architecture recommendations and resources', *Flutter docs*. The Flutter team. Accessed: June 10, 2026. [Online]. Available: <https://docs.flutter.dev/app-architecture/recommendations>
- [41] rrousselGit, *provider: A wrapper around InheritedWidget to make them easier to use and more reusable*. pub.dev. Accessed: June 09, 2026. [Online]. Available: <https://pub.dev/packages/provider>
- [42] Philips Hue, *flutter_reactive_ble: Reactive Bluetooth Low Energy (BLE) plugin that can communicate with multiple devices*. pub.dev. Accessed: June 09, 2026. [Online]. Available: https://pub.dev/packages/flutter_reactive_ble
- [43] tekartik, *sqflite: Flutter plugin for SQLite, a self-contained, high-reliability, embedded, SQL database engine*. pub.dev. Accessed: June 09, 2026. [Online]. Available: <https://pub.dev/packages/sqflite>
- [44] The Dart Team, *http: A composable, multi-platform, Future-based API for HTTP requests*. pub.dev. Accessed: June 09, 2026. [Online]. Available: <https://pub.dev/packages/http>
- [45] flutter-it, *command_it: command_it is a way to manage your state based on `ValueListenable` and the `Command` design pattern. It is a rebranding of flutter_command*. pub.dev. Accessed: June 09, 2026. [Online]. Available: https://pub.dev/packages/command_it
- [46] shamblett, *cbor: A CBOR library for Dart. An RFC8949 compliant encoding/decoding CBOR implementation*. pub.dev. Accessed: June 09, 2026. [Online]. Available: <https://pub.dev/packages/cbor>
- [47] imaNNeo, *FL Chart: A highly customizable Flutter chart library that supports Line Chart, Bar Chart, Pie Chart, Scatter Chart, and Radar Chart*. pub.dev. Accessed: June 09, 2026. [Online]. Available: https://pub.dev/packages/fl_chart
- [48] The Flutter Team, *shared_preferences: Flutter plugin for reading and writing simple key-value pairs. Wraps NSUserDefaults on iOS and SharedPreferences on Android*. pub.dev. Accessed: Sept. 10, 2026. [Online]. Available: https://pub.dev/packages/shared_preferences
- [49] 'Consumer class -- provider library'. Accessed: June 10, 2026. [Online]. Available: <https://pub.dev/documentation/provider/latest/provider/Consumer-class.html>
- [50] 'Selector class -- provider library'. Accessed: June 10, 2026. [Online]. Available: <https://pub.dev/documentation/provider/latest/provider/Selector-class.html>
- [51] Nordic Semiconductor DevAcademy, 'Pairing process - Bluetooth Low Energy Fundamentals'. Accessed: June 10, 2026. [Online]. Available: <https://academy.nordicsemi.com/courses/bluetooth-low-energy-fundamentals/lessons/lesson-5-bluetooth-le-security-fundamentals/topic/pairing-process/>
- [52] 'ChangeNotifier -- Simple app state management', *Flutter docs*. The Flutter team. Accessed: June 11, 2026. [Online]. Available: <https://docs.flutter.dev/data-and-backend/state-mgmt/simple#changenotifier>

- [53] Bluetooth Special Interest Group, 'Volume 6: Low Energy Controller -- 1.2 Bit ordering', *Bluetooth Core Specification 6.0*. Accessed: June 11, 2026. [Online]. Available: <https://www.bluetooth.com/wp-content/uploads/Files/Specification/HTML/Core-60/out/en/low-energy-controller/link-layer-specification.html#UUID-17792dae-6036-b3f4-4e04-f244c12d387c>
- [54] 'Battery Level Status - Battery Service 1.1', *Bluetooth Service specification*. Accessed: June 11, 2026. [Online]. Available: https://www.bluetooth.com/wp-content/uploads/Files/Specification/HTML/BAS_v1.1/out/en/index-en.html#UUID-37af6df3-3115-0c29-7fe3-468d607a3bff
- [55] Espressif, *esp_bas.h, Definition of Battery Level Status characteristic, Line 153 -- Espressif IoT Library*. Accessed: Sept. 11, 2026. [Online]. Available: https://github.com/espressif/esp-iot-solution/blob/master/components/bluetooth/ble_services/bas/include/esp_bas.h#L153
- [56] Argenox, 'Bluetooth 6 Speed: Guide to Maximizing BLE Data Throughput'. Accessed: June 11, 2026. [Online]. Available: <https://argenox.com/blog/bluetooth-le-throughput-max-performance>
- [57] 'ValueNotifier<T> class', *Flutter API reference documentation*. Accessed: June 11, 2026. [Online]. Available: <https://api.flutter.dev/flutter/foundation/ValueNotifier-class.html>
- [58] 'SnackBar class', *Flutter API reference documentation*. Accessed: June 11, 2026. [Online]. Available: <https://api.flutter.dev/flutter/material/SnackBar-class.html>
- [59] 'Serializing JSON inside model classes -- JSON and serialization', *Flutter docs*. The Flutter team. [Online]. Available: <https://docs.flutter.dev/data-and-backend/serialization/json#serializing-json-inside-model-classes>
- [60] Mainmatter, *Gerust: Gerust is a project generator for Rust backend projects. It takes care of the accidental complexity so you can stay focused on what matters*.
- [61] Transact.rs, *SQLx: The Rust SQL Toolkit. An async, pure Rust SQL crate featuring compile-time checked queries without a DSL. Supports PostgreSQL, MySQL, and SQLite*. Accessed: June 11, 2026. [Online]. Available: <https://github.com/transact-rs/sqlx>
- [62] Keats, *validator: Simple validation for Rust structs*. Accessed: June 07, 2026. [Online]. Available: <https://github.com/Keats/validator>
- [63] J. Tschan, 'Pull Requests by TheJanzap -- mainmatter/gerust'. Accessed: June 12, 2026. [Online]. Available: <https://github.com/mainmatter/gerust/pulls?q=is%3Apr+author%3A%40TheJanzap>
- [64] Conventional Commits contributors, *Conventional Commits: A specification for adding human and machine readable meaning to commit messages*, v1.0.0 edn. Accessed: Feb. 19, 2026. [Online]. Available: <https://www.conventionalcommits.org/en/v1.0.0/>
- [65] 'Git feature branch workflow', *Atlassian*. Accessed: Feb. 19, 2026. [Online]. Available: <https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow>
- [66] 'Gitflow workflow', *Atlassian*. Accessed: Feb. 19, 2026. [Online]. Available: <https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>
- [67] 'Time Tracking', *GitLab Docs*. GitLab. Accessed: Mar. 27, 2026. [Online]. Available: https://docs.gitlab.com/user/project/time_tracking/
- [68] Nina Grässli and Jannis Tschan, *GitLab Time-Report: GitLab Time-Report generates statistics and graphs from your GitLab Time Tracking data*. Accessed: Feb. 19, 2026. [Online]. Available: <https://gitlab.com/gitlab-time-report/gitlab-time-report>
- [69] Robert C. Martin, 'Design Principles and Design Patterns', *objectmentor.com*, 2000. Accessed: Mar. 24, 2026. [Online]. Available: https://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

- [70] 'Create Widgets: Learn more about stateless widgets and how to build your own.', *Flutter docs*. The Flutter team. Accessed: Mar. 30, 2026. [Online]. Available: <https://docs.flutter.dev/learn/pathway/tutorial/widget-fundamentals>
- [71] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. 1994, pp. 19–20.
- [72] 'Interface, Class Modifiers', *Dart documentation*. The Dart team. Accessed: Apr. 26, 2026. [Online]. Available: <https://dart.dev/language/class-modifiers#interface>
- [73] 'Making Structs and Enums Public', in *The Rust Programming Language*, 3rd edn, 2026. Accessed: Apr. 12, 2026. [Online]. Available: <https://doc.rust-lang.org/book/ch07-03-paths-for-referring-to-an-item-in-the-module-tree.html#making-structs-and-enums-public>
- [74] 'Keyword dyn', *Rust API documentation - The Rust Standard Library*. The Rust Project Developers. Accessed: Apr. 08, 2026. [Online]. Available: <https://doc.rust-lang.org/std/keyword.dyn.html>
- [75] 'Performing Dynamic Dispatch', in *The Rust Programming Language*, 3rd edn, 2026. Accessed: Apr. 08, 2026. [Online]. Available: <https://doc.rust-lang.org/book/ch18-02-trait-objects.html#performing-dynamic-dispatch>
- [76] 'Performance of Code Using Generics', in *The Rust Programming Language*, 3rd edn, 2026. Accessed: Apr. 12, 2026. [Online]. Available: <https://doc.rust-lang.org/book/ch10-01-syntax.html#performance-of-code-using-generics>
- [77] 'Inheritance as a Type System and as Code Sharing', in *The Rust Programming Language*, 3rd edn, 2026. Accessed: Apr. 12, 2026. [Online]. Available: <https://doc.rust-lang.org/book/ch18-01-what-is-oo.html#inheritance-as-a-type-system-and-as-code-sharing>
- [78] Monday Morning Haskell, 'Obey the (Type) Laws!'. Accessed: Apr. 08, 2026. [Online]. Available: <https://mmhaskell.com/blog/2017/3/13/obey-the-type-laws>
- [79] 'Macro unimplemented in std', *Rust API documentation - The Rust Standard Library*. The Rust Project Developers. Accessed: Apr. 08, 2026. [Online]. Available: <https://doc.rust-lang.org/std/macro.unimplemented.html>
- [80] Dart team, *analyzer: This package provides a library that performs static analysis of Dart code*. pub.dev. Accessed: Mar. 26, 2026. [Online]. Available: <https://pub.dev/packages/analyzer>
- [81] 'Enabling Dart team recommended linter rules - Customizing static analysis.', *Dart documentation*. The Dart team. Accessed: Mar. 29, 2026. [Online]. Available: <https://dart.dev/tools/analysis#lints>
- [82] Dart team, *lints: Official Dart lint rules. Defines the 'core' and 'recommended' set of lints suggested by the Dart team*. pub.dev. Accessed: Mar. 29, 2026. [Online]. Available: <https://pub.dev/packages/lints>
- [83] The Flutter team, *flutter_lints: Recommended lints for Flutter apps, packages, and plugins to encourage good coding practices*. pub.dev. Accessed: Mar. 26, 2026. [Online]. Available: https://pub.dev/packages/flutter_lints
- [84] 'Rules - Linter Rules.', *Dart documentation*. The Dart team. Accessed: Mar. 29, 2026. [Online]. Available: <https://dart.dev/tools/linter-rules#rules>
- [85] 'Excluded Rules', *very_good_analysis: Lint rules for Dart and Flutter used internally at Very Good Ventures*. Accessed: Mar. 29, 2026. [Online]. Available: https://pub.dev/packages/very_good_analysis#excluded-rules

- [86] Very Good Ventures, 'very_good_analysis: Lint rules for Dart and Flutter used internally at Very Good Ventures.'. pub.dev. Accessed: Mar. 29, 2026. [Online]. Available: https://pub.dev/packages/very_good_analysis
- [87] 'always_use_package_imports: Linter Rules', *Dart documentation*. The Dart team. Accessed: Mar. 29, 2026. [Online]. Available: https://dart.dev/tools/linter-rules/always_use_package_imports
- [88] 'public_member_api_docs: Linter Rules', *Dart documentation*. The Dart team. Accessed: Mar. 29, 2026. [Online]. Available: https://dart.dev/tools/linter-rules/public_member_api_docs
- [89] 'lines_longer_than_80_chars: Linter Rules', *Dart documentation*. The Dart team. Accessed: May 09, 2026. [Online]. Available: https://dart.dev/tools/linter-rules/lines_longer_than_80_chars
- [90] The Rust Project Developers, *clippy: A bunch of lints to catch common mistakes and improve your Rust code*. Accessed: Feb. 23, 2026. [Online]. Available: <https://github.com/rust-lang/rust-clippy>
- [91] The Rust Project Developers, 'Pedantic: Clippy's Lints - Clippy's Documentation'. Accessed: Mar. 26, 2026. [Online]. Available: <https://doc.rust-lang.org/stable/clippy/lints.html#pedantic>
- [92] Dart team, *dart_style: An opinionated formatter/linter for Dart code*. pub.dev. Accessed: Mar. 26, 2026. [Online]. Available: https://pub.dev/packages/dart_style
- [93] 'Effective Dart: Style. Formatting and naming rules for consistent, readable code.', *Dart documentation*. The Dart team. Accessed: Mar. 29, 2026. [Online]. Available: <https://dart.dev/effective-dart/style>
- [94] rustfmt contributors, *rustfmt: Format Rust code*. Accessed: Feb. 23, 2026. [Online]. Available: <https://github.com/rust-lang/rustfmt>
- [95] The Rust Project Developers, 'The Rust Style Guide'. Accessed: Mar. 29, 2026. [Online]. Available: <https://doc.rust-lang.org/nightly/style-guide/>
- [96] typstyle contributors, *typstyle: Beautiful and reliable typst code formatter*. Accessed: Feb. 23, 2026. [Online]. Available: <https://github.com/typstyle-rs/typstyle>
- [97] Myriad-Dreamin et al., *tinymist: Tinymist [taɪni mɪst] is an integrated language service for Typst [taɪpst]*. Accessed: Feb. 23, 2026. [Online]. Available: <https://github.com/Myriad-Dreamin/tinymist>
- [98] maikol-solis, '[Feature]: Add one sentence per line as an alternative to hardline breaks. · Issue #405 · typstyle-rs/typstyle'. Accessed: Feb. 23, 2026. [Online]. Available: <https://github.com/typstyle-rs/typstyle/issues/405>
- [99] T. Erhart and C. Furrer, 'Orientierungshilfe für die Dokumentation einer Studien- oder Bachelorarbeit', OST - Ostschweizer Fachhochschule, Apr. 2025. Accessed: Apr. 21, 2026. [Online]. Available: <https://ostch.sharepoint.com/:b:r/teams/TS-StudiengangInformatik/Freigegebene%20Dokumente/Studieninformationen/Studien-%20und%20Bachelorarbeiten/Informationen/Orientierungshilfe%20SA%20BA%20Dokumentation.pdf?csf=1&web=1&e=7Dneix>
- [100] S. Richter and M. Stocker, 'Leitfaden für Bachelor- und Studienarbeiten', OST - Ostschweizer Fachhochschule, Dec. 2025. Accessed: Apr. 21, 2026. [Online]. Available: [https://ostch.sharepoint.com/:b:r/teams/TS-StudiengangInformatik/Freigegebene%20Dokumente/Studieninformationen/Studien-%20und%20Bachelorarbeiten/Leitfaden%20BA%20SA%20v1.4%20\(ab%20FS%2026\).pdf?csf=1&web=1&e=TbYhAO](https://ostch.sharepoint.com/:b:r/teams/TS-StudiengangInformatik/Freigegebene%20Dokumente/Studieninformationen/Studien-%20und%20Bachelorarbeiten/Leitfaden%20BA%20SA%20v1.4%20(ab%20FS%2026).pdf?csf=1&web=1&e=TbYhAO)

- [101] F. D. Mehta and T. Kählin, *SE Project LaTeX Documentation Template*. OST - Ostschweizer Fachhochschule. Accessed: Apr. 21, 2026. [Online]. Available: <https://gitlab.ost.ch/SEProj/latex-documentation-template>
- [102] D. Glasl, *Arbeiten-Vorlage-Typst: Vorlage für Arbeiten an der Fachhochschule OST mit Typst*. Accessed: Apr. 21, 2026. [Online]. Available: <https://github.com/darioglasl/Arbeiten-Vorlage-Typst>
- [103] 'Label type', *Typst documentation*. Accessed: Apr. 21, 2026. [Online]. Available: <https://typst.app/docs/reference/foundations/label/>
- [104] 'Reference function', *Typst documentation*. Accessed: Apr. 21, 2026. [Online]. Available: <https://typst.app/docs/reference/model/ref/>
- [105] 'Letter case § Use within programming languages', *Wikipedia*. Accessed: Apr. 21, 2026. [Online]. Available: https://en.wikipedia.org/wiki/Letter_case#Use_within_programming_languages
- [106] 'flutter_test library', *Flutter API reference documentation*. Accessed: June 03, 2026. [Online]. Available: https://api.flutter.dev/flutter/flutter_test/
- [107] 'An introduction to unit testing', *Flutter docs*. The Flutter team. Accessed: June 03, 2026. [Online]. Available: <https://docs.flutter.dev/cookbook/testing/unit/introduction>
- [108] *Mockito: A mock framework inspired by Mockito with APIs for Fakes, Mocks, behavior verification, and stubbing*. pub.dev. Accessed: June 03, 2026. [Online]. Available: <https://pub.dev/packages/mockito>
- [109] The Dart team, 'Nice mocks vs classic mocks -- Mockito'. Accessed: June 03, 2026. [Online]. Available: <https://pub.dev/packages/mockito#nice-mocks-vs-classic-mocks>
- [110] The Dart Team, *build_runner: A build system for Dart code generation and modular compilation*. pub.dev. Accessed: June 04, 2026. [Online]. Available: https://pub.dev/packages/build_runner
- [111] 'Unit Tests', in *The Rust Programming Language*, 3rd edn, 2026. Accessed: June 07, 2026. [Online]. Available: <https://doc.rust-lang.org/book/ch11-03-test-organization.html#unit-tests>
- [112] 'Private Function Tests', in *The Rust Programming Language*, 3rd edn, 2026. Accessed: June 07, 2026. [Online]. Available: <https://doc.rust-lang.org/book/ch11-03-test-organization.html#private-function-tests>
- [113] 'Macro assert in std', *Rust API documentation - The Rust Standard Library*. The Rust Project Developers. Accessed: June 07, 2026. [Online]. Available: <https://doc.rust-lang.org/stable/std/macro.assert.html>
- [114] 'Macro assert_eq in std', *Rust API documentation - The Rust Standard Library*. The Rust Project Developers. Accessed: June 07, 2026. [Online]. Available: https://doc.rust-lang.org/stable/std/macro.assert_eq.html
- [115] 'Macro assert_ne in std', *Rust API documentation - The Rust Standard Library*. The Rust Project Developers. Accessed: June 07, 2026. [Online]. Available: https://doc.rust-lang.org/stable/std/macro.assert_ne.html
- [116] 'Testing Flutter apps', *Flutter docs*. The Flutter team. Accessed: June 03, 2026. [Online]. Available: <https://docs.flutter.dev/testing/overview>
- [117] 'An introduction to widget testing', *Flutter docs*. The Flutter team. Accessed: June 03, 2026. [Online]. Available: <https://docs.flutter.dev/cookbook/testing/widget/introduction>
- [118] Google, 'Firebase Test Lab'. Accessed: June 11, 2026. [Online]. Available: <https://firebase.google.com/docs/test-lab?hl=en>

- [119] 'Run integration tests - Check app functionality with an integration test', *Flutter docs*. The Flutter team. Accessed: May 20, 2026. [Online]. Available: <https://docs.flutter.dev/testing/integration-tests#run-integration-tests>
- [120] 'Integration Tests', in *The Rust Programming Language*, 3rd edn, 2026. Accessed: June 07, 2026. [Online]. Available: <https://doc.rust-lang.org/book/ch11-03-test-organization.html#integration-tests>
- [121] cksac, *fake-rs: A library for generating fake data in Rust*. Accessed: June 07, 2026. [Online]. Available: <https://github.com/cksac/fake-rs>
- [122] 'DB crate README'. Accessed: June 07, 2026. [Online]. Available: <https://github.com/mainmatter/gerust/blob/main/blueprint/db/README.md>
- [123] The CocoaPods team, *CocoaPods: The Cocoa Dependency Manager*. Accessed: June 11, 2026. [Online]. Available: <https://github.com/CocoaPods/CocoaPods>
- [124] Dovetail editorial team, 'A step-by-step guide to writing an effective usability test script'. Accessed: Mar. 15, 2026. [Online]. Available: <https://dovetail.com/ux/how-to-write-a-usability-testing-script/>
- [125] A. Bangor, P. T. Kortum, and J. T. Miller, 'An Empirical Evaluation of the System Usability Scale', 2008, *International Journal of Human-Computer Interaction*. doi: [10.1080/10447310802205776](https://doi.org/10.1080/10447310802205776).
- [126] 'CI/CD pipelines', *GitLab Docs*. GitLab. Accessed: Feb. 23, 2026. [Online]. Available: <https://docs.gitlab.com/ci/pipelines/>
- [127] 'Hide a job', *GitLab Docs*. GitLab. Accessed: May 04, 2026. [Online]. Available: <https://docs.gitlab.com/ci/jobs/#hide-a-job>
- [128] Cimbali, *pympress: Pympress is a simple yet powerful PDF reader designed for dual-screen presentations*. Accessed: Apr. 28, 2026. [Online]. Available: <https://github.com/Cimbali/pympress>
- [129] Cirrus Labs, 'Flutter container 3.41.6'. Mar. 26, 2026. Accessed: Mar. 26, 2026. [Online]. Available: <https://github.com/cirruslabs/docker-images-flutter/pkgs/container/flutter/760495787?tag=3.41.6>
- [130] Cirrus Labs, *docker-images-flutter README.md*. Accessed: May 04, 2026. [Online]. Available: <https://github.com/cirruslabs/docker-images-flutter?tab=readme-ov-file>
- [131] tmaczukin, 'Reply to: Can not append values to PATH environment variable', *GitLab Runner: Issue #1249*. [Online]. Available: https://gitlab.com/gitlab-org/gitlab-runner/-/work_items/1249#note_4957993
- [132] 'Set up macOS runners', *GitLab Docs*. GitLab. Accessed: May 04, 2026. [Online]. Available: https://docs.gitlab.com/runner/configuration/macos_setup
- [133] sickcodes, *Docker-OSX: Run macOS VM in a Docker! Run near native OSX-KVM in Docker! X11 Forwarding! CI/CD for OS X Security Research! Docker mac Containers*. Accessed: May 04, 2026. [Online]. Available: <https://github.com/sickcodes/Docker-OSX>
- [134] 'dart format: Command-line tool for formatting Dart source code.', *Dart documentation*. The Dart team. Accessed: Mar. 22, 2026. [Online]. Available: <https://dart.dev/tools/dart-format>
- [135] TOPdesk, *junitreport: Generate JUnit XML reports from dart test runs. Transforms the output of dart or flutter tests to JUnit style XML*. pub.dev. Accessed: Feb. 23, 2026. [Online]. Available: <https://pub.dev/packages/junitreport>
- [136] *full_coverage: A package script for allowing coverage test tool to see all Dart files*. pub.dev. Accessed: May 04, 2026. [Online]. Available: https://pub.dev/packages/full_coverage

- [137] The Rust Project Developers, 'Image Layer Details - rust:1.94 | Docker Hub'. Accessed: Mar. 20, 2026. [Online]. Available: <https://hub.docker.com/layers/library/rust/1.94/images>
- [138] cargo-bins organization, *cargo-binstall: Binary installation for rust projects*. Accessed: Feb. 23, 2026. [Online]. Available: <https://github.com/cargo-bins/cargo-binstall>
- [139] 'cargo db prepare - cli README'. Accessed: Mar. 27, 2026. [Online]. Available: <https://github.com/mainmatter/gerust/blob/main/blueprint/cli/README.md.liquid#L42>
- [140] johnterickson, *cargo2junit: Converts cargo's json output (from stdin) to JUnit XML (to stdout)*. Accessed: Feb. 23, 2026. [Online]. Available: <https://github.com/johnterickson/cargo2junit>
- [141] 'Code coverage', *GitLab Docs*. GitLab. Accessed: May 14, 2026. [Online]. Available: https://docs.gitlab.com/ci/testing/code_coverage/
- [142] The Rust Unstable Book contributors, 'RUSTC_BOOTSTRAP - The Rust Unstable Book'. Accessed: Mar. 20, 2026. [Online]. Available: https://doc.rust-lang.org/beta/unstable-book/compiler-environment-variables/RUSTC_BOOTSTRAP.html
- [143] xd009642 et al., *cargo-tarpaulin: A code coverage tool for Rust projects*. Accessed: Feb. 23, 2026. [Online]. Available: <https://github.com/xd009642/tarpaulin>
- [144] 'GitLab Container Registry', *GitLab Docs*. GitLab. Accessed: May 07, 2026. [Online]. Available: https://docs.gitlab.com/user/packages/container_registry/
- [145] 'SSH keys when using the Docker executor - Using SSH keys with GitLab CI/CD', *GitLab Docs*. GitLab. Accessed: May 14, 2026. [Online]. Available: https://docs.gitlab.com/ci/jobs/ssh_keys/#ssh-keys-when-using-the-docker-executor
- [146] The OpenBSD team, 'ssh-agent(1)'. Arch Linux manual pages. Accessed: May 14, 2026. [Online]. Available: <https://man.archlinux.org/man/ssh-agent.1>
- [147] The OpenBSD team, 'StrictHostKeyChecking - ssh_config(5)'. Arch Linux manual pages. Accessed: May 14, 2026. [Online]. Available: https://man.archlinux.org/man/core/openssh/ssh_config.5.en#StrictHostKeyChecking
- [148] 'Verify your SSH connection - Using SSH keys with GitLab CI/CD', *GitLab Docs*. GitLab. Accessed: May 14, 2026. [Online]. Available: <https://docs.gitlab.com/user/ssh/#verify-your-ssh-connection>
- [149] Mozilla, *sccache: Sccache is a ccache-like tool. It is used as a compiler wrapper and avoids compilation when possible. Sccache has the capability to utilize caching in remote storage environments, including various cloud storage options, or alternatively, in local storage*. Accessed: May 09, 2026. [Online]. Available: <https://github.com/mozilla/sccache>
- [150] 'Architecture', *sccache Documentation*. Accessed: May 09, 2026. [Online]. Available: <https://github.com/mozilla/sccache/blob/main/docs/Architecture.md>
- [151] J. Tschan, 'ci: Use sccache for caching Rust builds (!12) · Merge requests · CoolMedTravel / Backend · GitLab'. Accessed: May 09, 2026. [Online]. Available: https://gitlab.ost.ch/coolmedtravel/backend/-/merge_requests/12
- [152] 'GitLab Pages', *GitLab Docs*. GitLab. Accessed: Apr. 28, 2026. [Online]. Available: <https://docs.gitlab.com/user/project/pages/>
- [153] 'Internationalizing Flutter apps'. in Flutter docs. The Flutter team. Accessed: June 08, 2026. [Online]. Available: <https://docs.flutter.dev/ui/internationalization>
- [154] 'Success Criterion 1.4.3 Contrast', in *Web Content Accessibility Guidelines (WCAG)*, 2.2 edn. Accessed: May 29, 2026. [Online]. Available: <https://www.w3.org/TR/WCAG22/#contrast-minimum>

- [155] S. Kapferer and T. Kälin, 'SE Practices 2: 01 - Project Planning', OST - Ostschweizer Fachhochschule, Feb. 2025.
- [156] K. Nguyen, 'What's new in Flutter 3.44'. Accessed: June 12, 2026. [Online]. Available: <https://blog.flutter.dev/whats-new-in-flutter-3-44-b0cc1ad3c527>
- [157] C. Ooi, 'Announcing Dart 3.12'. Accessed: June 12, 2026. [Online]. Available: <https://dart.dev/blog/announcing-dart-3-12>
- [158] The rustc Book contributors, 'The rustc Book: Platform Support'. Accessed: Feb. 23, 2026. [Online]. Available: <https://doc.rust-lang.org/stable/rustc/platform-support.html>
- [159] The Rust Edition Guide contributors, 'The Rust Edition Guide: What are Editions?'. Accessed: Feb. 23, 2026. [Online]. Available: <https://doc.rust-lang.org/edition-guide/editions/index.html>
- [160] JetBrains, 'RustRover: Rust IDE by JetBrains'. Accessed: Feb. 23, 2026. [Online]. Available: <https://www.jetbrains.com/rust/>
- [161] Microsoft, 'Visual Studio Code - Code Editing, Redefined.'. Accessed: Feb. 23, 2026. [Online]. Available: <https://code.visualstudio.com/>
- [162] Typst contributors, *Typst: A new markup-based typesetting system that is powerful and easy to learn. - Version 0.14.1 (December 3, 2025).* (Dec. 12, 2025). Accessed: Feb. 23, 2026. [Online]. Available: <https://github.com/typst/typst/releases/tag/v0.14.2>
- [163] PlantUML contributors, *PlantUML: Generate diagrams from textual description.* Accessed: May 09, 2026. [Online]. Available: <https://github.com/plantuml/plantuml>
- [164] PlantUML stdlib contributors, *C4-PlantUML: C4-PlantUML combines the benefits of PlantUML and the C4 model for providing a simple way of describing and communicate software architectures.* Accessed: May 09, 2026. [Online]. Available: <https://github.com/plantuml-stdlib/C4-PlantUML>
- [165] Street Side Software, *vscode-spell-checker: A simple source code spell checker for code.* Accessed: Feb. 23, 2026. [Online]. Available: <https://github.com/streetsidesoftware/vscode-spell-checker>
- [166] adrianjagielak, 'Flutter container 3.44.0'. Accessed: June 12, 2026. [Online]. Available: <https://github.com/adrianjagielak/docker-images-flutter/pkgs/container/flutter/873832577?tag=3.44.0>
- [167] Typst contributors, 'Typst Container 0.14.2'. Accessed: Feb. 23, 2026. [Online]. Available: <https://github.com/typst/typst/pkgs/container/typst/609612362?tag=0.14.2>
- [168] VannaDii, *cobertura: Converts LCOV to Cobertura format; the ideal format for GitHub and Azure DevOps.* pub.dev. Accessed: Feb. 23, 2026. [Online]. Available: <https://pub.dev/packages/cobertura>
- [169] *openssh-client-common.* Alpine Linux Packages. Accessed: May 14, 2026. [Online]. Available: https://pkgs.alpinelinux.org/package/edge/main/x86_64/openssh-client-common
- [170] *openssh-client-default.* Alpine Linux Packages. Accessed: May 14, 2026. [Online]. Available: https://pkgs.alpinelinux.org/package/edge/main/x86_64/openssh-client-default
- [171] OST Kommission Lehre und OST Kommission Weiterbildung, 'Leitlinie zum Umgang mit KI-basierten Hilfsmitteln in Lehre und Weiterbildung', Sept. 2024. Accessed: Feb. 23, 2026. [Online]. Available: <https://ostch-my.sharepoint.com/shared?listurl=https%3A%2F%2Fostch%2Fsharepoint%2Ecom%2Fteams%2FTS%2DStudiengangInformatik%2FFreigegebene%20Dokumente&viewid=eaede7da%2D2ef7%2D421c%2Ddaad%2D6a54e117731&id=%2Fteams%2FTS%2DStudiengangInformatik%2FFreigegebene%20Dokumente%2FStudieninformationen%2FLeitlinie%20KI%2DEinsatz%20in%20Lehre%20und%20Weiterbildung%2Epdf&parent=%2>

- [172] OpenAI, 'ChatGPT'. Accessed: Feb. 23, 2026. [Online]. Available: <https://chatgpt.com/>
- [173] DeepL, 'DeepL Write'. Accessed: Feb. 23, 2026. [Online]. Available: <https://www.deepl.com/de/write>
- [174] JetBrains, 'AI Assistant features'. Accessed: Feb. 23, 2026. [Online]. Available: <https://www.jetbrains.com/ai-assistant/>
- [175] Anthropic, 'Claude AI'. Accessed: Feb. 23, 2026. [Online]. Available: <https://claude.ai/>
- [176] Android Developers, 'Gemini in Android Studio'. Accessed: May 04, 2026. [Online]. Available: <https://developer.android.com/gemini-in-android>
- [177] J. Brooke, *Usability Evaluation In Industry*. Taylor & Francis, 1996, pp. 189–194. doi: [10.1201/9781498710411](https://doi.org/10.1201/9781498710411).