University of Agder

Faculty of Engineering and Science

Department of Information and Communication Technology

Study Sprint

Course: IKT205-G 26V

Application Development

Submitted by

Group 18

Christopher Sanden

Fabian Haukedal Johansen

Teodor Salvesen

May 14, 2026

Mandatory group declaration

Each student is responsible for familiarizing themselves with what constitutes permitted aids, guidelines for the use of these, and rules regarding the use of sources. The declaration is intended to make students aware of their responsibility and of the consequences that cheating may entail. A missing declaration does not exempt students from their responsibility.

1.	We hereby declare that our submission is our own work, and that we have not used other sources or received any other assistance than what is mentioned in the submission.	Yes
2.	**We further declare that this submission:** • Has not been used for another examination at another department/university/university college in Norway or abroad. • Does not refer to the work of others without this being stated. • Does not refer to our own previous work without this being stated. • Has all references listed in the bibliography. • Is not a copy, duplicate, or transcript of another person's work or submission.	Yes
3.	We are aware that breaches of the above are to be regarded as cheating and may result in annulment of the examination and exclusion from universities and university colleges in Norway, cf. the Universities and Colleges Act §§4-7 and 4-8 and the Examination Regulations §§ 31.	Yes
4.	We are aware that all submitted assignments may be checked for plagiarism.	Yes
5.	We are aware that the University of Agder will handle all cases where there is suspicion of cheating according to the university college's guidelines for handling cases of cheating.	Yes
6.	We have familiarized ourselves with the rules and guidelines for the use of sources and references on the library's webpages.	Yes
7.	We have, by majority, agreed that the effort within the group is noticeably different and therefore wish to be assessed individually. Ordinarily, all participants in the project are assessed collectively.	No

Publication agreement

Authorization for electronic publication of the assignment The author(s) hold the copyright to the assignment. This means, among other things, the exclusive right to make the work available to the public (Copyright Act. §2).
Assignments that are exempt from public disclosure or are subject to a duty of confidentiality/confidential will not be published.

We hereby grant the University of Agder a royalty-free right to make the assignment available for electronic publication:	No
Is the assignment restricted (confidential)?	No
Is the assignment exempt from public disclosure?	No

Abstract

This report presents the development of Study Sprint, a mobile productivity application designed to help students organise academic work and convert planned tasks into focused study sessions, or sprints. The project addresses common student needs such as reducing procrastination, improving focus, managing assignments and tasks, and making study progress more visible. The application combines a structured hierarchy for subjects, assignments and tasks with timers, breaks, reminders, progress tracking and persistent storage.

The project followed an iterative and scope-conscious development process. Early prototypes were used to develop the core workflow, while later iterations focused on improving navigation, reducing redundancy, improving timer reliability, and lowering friction for first-time users.

The resulting application provides a workflow orientated around students, where users can define what they need to study, start a timed sprint from a specific task, take structured breaks, and review their progress afterwards. The project shows that a smaller and more specialised set of features can be more suitable for this case, rather than a broad productivity platform. While the product meets the main goals of structured planning, task-linked focus sessions, and progress visibility. Given a larger development window, the app would evolve from it's current state by adding whatever could-have requirements are missing from the current build, along with further integration with other third party solutions such as Canvas or native calendar integration.

Contents

1 Introduction

This report presents the development of *Study Sprint*, a mobile productivity application designed to help students organise academic work and turn that structure into focused study sessions. The core idea behind the application is to reduce the gap between intending to study and actually starting. Instead of treating planning, timing and progress as separate concerns, Study Sprint connects subjects, assignments, tasks, timed focus sessions, breaks and visible progress into one lightweight workflow.

The motivation for the project came from a common frustration with existing study and productivity applications. Many such applications are either too broad, too feature-heavy or too abstract for students who simply want to organise what they need to study and begin working with as little friction as possible. In many cases, users are forced through unnecessary setup, presented with too many unrelated features, or given timer tools that are disconnected from the work they are supposed to support. Study Sprint was developed as a response to that problem.

The project therefore focused on a small but coherent feature set. The goal was not to create the most advanced productivity application, but to create one that is easy to understand, fast to use and reliable in practice. This meant prioritising a clear study hierarchy, low-friction navigation, task-linked study sessions, break handling and progress visibility over broader functionality that would have increased the complexity without adding equal value to the intended user.

The conceptual background for the app was also influenced by the Pomodoro technique, where focused work intervals are separated by short breaks. This was relevant because the project aimed to support concentration, reduce study friction, and make it easier for users to turn vague study intentions into concrete work sessions. Recent review literature also suggests that structured Pomodoro-style intervals can improve focus, reduce fatigue, and support sustained task performance in demanding learning contexts [1, 5].

From a development perspective, the project followed an iterative approach. Early prototypes were used to explore structure, interaction and timer behaviour before the application was gradually refined through implementation, testing and revision. Several parts of the application, especially the timer flow and session handling, changed significantly over time as practical issues were discovered and resolved. This process helped move the product from a simple prototype toward a more complete and reliable study tool.

1.1 Product Vision

The complete project vision is included in the appendices folder. The following section summarises the most important parts of that vision and shows how they shaped the direction of the product.

1.2 Target C ustomer

The primary target group for Study Sprint is students in higher education who want a simple and structured way to organise study work, start focused work sessions and keep track of their progress [2].

A secondary target group includes other learners, such as upper secondary students, as well as users who rely on timed work sessions for productivity in non-academic contexts. Even so, the application is mainly designed around student needs, since that is the clearest and most relevant use case for the project.

1.2.1 Customer Needs

Study Sprint is intended to address a small set of practical needs that repeatedly appear in student work:

- better focus during study sessions through timed work periods and breaks,
- simple planning and organisation through subjects, assignments, and tasks,
- motivation through visible progress and recorded study activity,
- low friction, so users can understand the app quickly and begin studying without unnecessary setup.

These needs are closely related. A study timer becomes more useful when it is connected to real tasks, and planning becomes more meaningful when it leads directly into focused work.

1.2.2 Critical Product Attributes

The most important product attribute for Study Sprint is simplicity. The application should be intuitive enough that a user can understand the main study flow almost immediately. A user should be able to move from creating structure to starting a study session without confusion or unnecessary decision-making.

Reliability is also critical. Since the application depends heavily on timed study sessions and progress tracking, it must handle active sessions, completion, cancellation, breaks, and saved progress in a dependable way. If the timer flow feels unstable or if recorded study activity becomes inconsistent, the application quickly loses its usefulness.

A third important attribute is a clear and focused user interface. The design does not need to be visually complex, but it should feel complete, deliberate, and easy to navigate. Since the app is intended to reduce procrastination rather than add friction, fast and understandable interaction is especially important.

Finally, scope control is an important product attribute in itself. A smaller and more polished application is more valuable for this project than a broader solution with too many unfinished or weakly integrated features. A successful version of Study Sprint, therefore, includes structured planning, task-linked study sessions, break handling, progress visibility, and persistence, while avoiding feature expansion that would dilute the product's main purpose.

1.2.3 Unique Selling Points

Study Sprint enters a crowded space of Pomodoro apps, study planners, habit trackers, and general productivity tools such as Forest, Focus To-Do, and Todoist. These applications often provide useful functionality, but many of them are either too broad, too feature-heavy, or too focused on premium features and generalised productivity workflows [2].

Compared to these alternatives, Study Sprint aims to be more focused and more student-orientated. Rather than trying to include every possible productivity feature, it combines only the features most relevant to the intended workflow: subject structure, assignments, tasks, timed focus sessions, breaks, and visible progress.

The main unique selling point of Study Sprint is therefore not feature quantity, but feature connection. The application links planning and studying more directly than a simple timer app, while remaining much lighter and more focused than broader productivity platforms. This makes it easier for a student to decide what to study, begin working quickly, and see meaningful progress afterward.

1.2.4 Target Time-frame and Budget

The target time-frame for Study Sprint was the duration of the project period, ending in early May. Because this work was carried out alongside other courses, assignments, and exams, the project had to be planned with a strong focus on the effort-to-value ratio rather than feature ambition.

For that reason, the team aimed to establish a working prototype early and then use the remaining time on refinement, reliability, usability, and report work. A realistic development path was to first define the product structure and screen flow, then implement the core planning hierarchy, timer functionality, session persistence, break flow, and progress tracking, before focusing on testing and polish [2].

The financial budget for the project was expected to be minimal. The application was developed using free or low-cost tools and frameworks, with time being the main practical constraint. This made it especially important to avoid unnecessary complexity and to concentrate on the parts of the application that contributed most directly to the intended study experience.

2 Method

The development of Study Sprint followed a practical and iterative project method. Rather than treating the application as a fixed specification that could be implemented in one pass, the team worked from a product vision, built early versions of the core functionality, and then refined the solution as weaknesses became clearer through use, testing, and discussion.

This approach fit the nature of the project well. The main challenge was not only to implement isolated features, but to shape a small mobile product that felt coherent, reliable, and easy to use in practice. Because of that, the method had to support both technical implementation and repeated revision of structure, flow, and scope.

2.1 Vision-Driven and Scope-Conscious Development

The project vision functioned as the main strategic starting point for the work. It defined the target users, the most important customer needs, and the product attributes that mattered most for success. In particular, simplicity, reliability, low friction, and realistic scope were treated as guiding constraints throughout the project [2].

This had a direct effect on development priorities. Features that supported the core study flow were given precedence, while ideas that would broaden the product without strengthening its main purpose were de-prioritised or excluded. That is also why the report separates requirements into must-have, should-have, could-have, and will-not-have categories. This prioritisation made it easier to protect the project from feature growth and to keep effort focused on the parts of the application that contributed most directly to the intended user experience.

2.2 Iterative Prototyping and Refinement

The practical development process was iterative. Early work focused on establishing the underlying structure of the app, including navigation, CRUD support for subjects, assignments, and tasks, and the first timer-related flows. These early versions were useful for confirming that the main ideas were technically possible, but they also revealed architectural and usability problems that were not obvious at the planning stage.

One clear example was the relationship between the app's conceptual data model and its screen structure. The intended hierarchy was always *Subject* → *Assignment* → *Task*, but earlier versions exposed assignments and tasks too much as top-level concepts. As the app was used and reviewed, this was recognised as a source of redundancy, weak context, and unnecessary navigation complexity. The architecture was therefore revised so that the interface reflected the actual hierarchy more clearly.

The same pattern appeared in the timer and session logic. Initial timer behaviour was gradually replaced by a more complete session model with focus sessions, breaks, persistence, and safer finalisation rules. Several parts of the timer flow changed more than once as practical issues were discovered, especially around routing, active-session recovery, break handling, and reliability. In that sense, prototyping was not only used to create a first version, but also as a way to expose where the product model was still too weak or too rough.

2.3 Incremental Delivery of Core Features

The implementation was carried out in layers rather than as one large build phase. A basic feature foundation was established first, followed by increasingly product-specific refinement.

At an early stage, the work concentrated on basic entity management and data flow, such as creating, editing, viewing, and deleting subjects, assignments, and tasks. After that, the project moved further toward the product's intended identity by adding progress tracking, local reminders, more structured navigation, and task-linked study sessions. Later work focused more strongly on reducing friction in the main study flow, improving onboarding, and making the timer and break cycle more robust.

This incremental method helped reduce risk. A simple but working baseline could be established first, after which the team could decide which refinements actually improved the product and which ideas were unnecessary within the project's time constraints.

2.4 Testing, Verification, and Revision

Testing in this project was not treated as something that only happened at the end. Instead, verification and revision were interwoven with development. When new functionality was added, the result was checked, weaknesses were identified, and the implementation was adjusted before the project moved further.

The notes show two main forms of verification. The first was manual testing of the app's actual behaviour, especially in flows where user interaction and timing mattered. This was particularly important for the timer, break handling, onboarding flow, routing, and notification behaviour. The second was technical verification through static checks such as linting and TypeScript compilation, which helped detect invalid code, inconsistent types, and integration mistakes before further changes were added.

This combination was necessary because the project contains both logic-heavy and interaction-heavy features. Static checks can confirm that the codebase is structurally valid, but they cannot on their own confirm that a timed session flow feels correct or that screen transitions behave as intended. Manual testing was therefore especially important whenever reliability, usability, or flow clarity were central concerns. For native-dependent functionality such as notifications, a development build was also used instead of relying only on Expo Go, since that gave more realistic behaviour during testing.

2.5 Collaboration and Work Organisation

The work was organised as a collaborative group project with regular meetings, shared responsibility, and ongoing task distribution. According to the project vision and group contract, the group planned to meet weekly to review progress, divide work, and discuss problems as they appeared. Discord functioned as the main communication channel between meetings, making it easier to adjust responsibilities when needed.

Because the project remained relatively limited in scope, with a small team and a fairly small number of central features, the group did not use a heavier Scrum-oriented project management setup with dedicated ticket tools such as Taiga. In practice, many tasks were easier to define, discuss, and re-prioritise through regular meetings and direct communication than through maintaining a formal backlog with only a few active items. This reduced unnecessary administrative overhead and made the work organisation more appropriate for the actual scale of the project.

Version control and shared documentation were important parts of the method. GitHub was used to manage source code and changes over time, while notes and work reports were used to document completed work, encountered problems, and follow-up decisions. This created a lightweight but useful development trail that supported both collaboration and report writing.

2.6 Why This Method Was Appropriate

An important reason for choosing this style of method was that Study Sprint is a product where usability, flow, and reliability matter at least as much as raw feature count. A more rigid process could have described the intended system early, but it would have been less useful for improving the real experience of moving from planning work to starting a study session.

The iterative and scope-conscious method therefore matched the project well. It allowed the team to begin with a workable prototype, identify weaknesses through repeated review and testing, and gradually move the application toward a more coherent and dependable final form without losing control of scope.

3 Architectural Design and Philosophy

3.1 Overview

Study Sprint was designed around a clear hierarchical productivity model:

Subject → Assignment → Task

This hierarchy became the foundation for how navigation and screen-level interaction was handled. The design aimed to make the application intuitive, reduce redundancy, and to make sure the interface reflected the actual the hierarchy in addition to the back-end database structure.

The goal for the design was to create an application that feels calm, minimal, and easy to understand. The architecture emphasises context, structure, and progressive disclosure.

3.2 Architectural Motivation

The initial structure of the application treated several related entities as if they were independent. Subjects, assignments and tasks were all exposed on the top-level. This probably made basic CRUD operations easier to test while developing the back-end. In any case this did not match the product model. This version of the code created several problems;

- tasks could appear without sufficient assignment or subject context,
- assignments felt detached from their parent subjects,
- the interface duplicated information across multiple screens,
- the difference between assignments and tasks are nuanced.

To fix this, the application was restructured so that the architecture of the interface matched the architecture of the data.

3.3 Navigation Architecture

The top-level navigation was simplified into three tabs, and in the final product, two tabs.

- **Dashboard**
- **Subjects**
- **Timer** (for ease of simultaneous development)

This was a deliberate decision. The application in its initial state risked becoming too flat by exposing assignments and tasks as separate top-level tabs. However, assignments and tasks are not standalone concepts.

The new tab structure gives each destination a distinct role:

- **Dashboard** acts as an overview screen for cross-cutting information
- **Subjects** acts as the primary entry point into the study hierarchy

3.4 Hierarchy-Driven Screen Design

The screen flow follows the actual logical structure of the application:

- the **Subjects** screen presents all subjects,
- the **Subject Details** screen acts as the hub for a single subject and its assignments,
- the **Assignment Details** screen acts as the hub for a single assignment and its tasks,
- the **Task Details** screen focuses on one task while preserving its parent context.

This means that users move deeper into the structure through meaningful parent-child relationships rather than through disconnected CRUD pages.

3.5 Hub-Based Detail Screens

A major architectural choice was to turn detail pages into *management hubs* instead of simple read-only detail views.

3.5.1 Subject Details as a Hub

The subject details screen was designed to serve as the central management hub for a subject. It contains:

- subject summary information,
- status and metadata,
- actions such as edit and delete,
- action for creating assignments,
- assignment sections organised by completion state.

This supports a structured flow where the users naturally move from a subject, into an assignment, and then into the tasks that make it actionable.

3.6 Reduction of Redundancy

One of the main architectural design principles was the removal of unnecessary duplication

3.6.1 Removal of Top-Level Assignment and Task Navigation

Assignments and tasks were removed from the tab bar because they did not function as top-level concepts in practice. Their meaning depends on the parent hierarchy. Keeping them as top-level tabs only worsens the contextual clarity and created repeated list views that duplicated information already available deeper in the hierarchy.

3.6.2 Reduction of Inline Management Controls

List screens were simplified so that entity cards primarily function as entry points rather than control panels. For example subject cards no longer contain too many management actions such as inline edit, delete, or progress-heavy UI. Those actions were moved into the corresponding hub screens.

This follows a simple principle: browsing screens should prioritise selection and orientation, while detail screes should prioritise management.

3.7 Reusable Upsert-Based Form Design

Another architectural decision was to reduce duplication in entity creation and editing workflows.

Originally, create and edit flows existed as separate screens even when they used close to identical fields, validation control, and layout. This increased maintenance overhead and produced repeated code.

To improve this, the application moved toward an **upsert-based architecture**. In this pattern, a single screen handles both creation and editing:

- if no id is present, the form works in create mode,
- if an id is present, the form loads existing data and works in edit mode

This was implemented for core entity forms such as subjects, assignments, and tasks. The benefit is that validation, styling, and layout remain consistent while avoiding unnecessary duplication of screen logic.

3.8 Shared Utility Extraction

As the design progressed, common patterns were extracted into globally shared utility modules.

3.8.1 Date Formatting

Date and timestamp rendering was centralised in a shared formatting utility. This replaced raw database timestamps with user-friendly display values and ensured consistency across subject, assignment, and task screens.

3.8.2 Subject colour System

A dedicated subject colour utility was introduced to centralise:

- the available subject colour palette,
- the type definition for valid subject colours,
- the mapping between a subject colour key and its visual values,
- helper logic for retrieving the correct visual accent set.

This avoided repeated colour logic and ensured that subject styling remained consistent across the hierarchy.

3.9 Design Philosophy

The architectural design was strongly influenced by a clear philosophy. The UI was intentionally shaped by the following principles:

3.9.1 Calm Over Visual Noise

The interface was designed to feel calm and discreet. The goal was not to make everything look unique, but rather to make structure intuitive and interaction obvious. This led to a restrained card-based UI, reduced redundancy, and fewer competing visual accents.

3.9.2 Hierarchy Over Flatness

The interface should communicate the real structure of the application. The design preserves parent-child relationships instead of flattening all the entities. This improves orientation and helps the user understand where they are in the study workflow.

3.9.3 Context Over Abstraction

As the user moves deeper into the hierarchy, the more important context becomes. For this reason, assignment screens display subject context, and task screens display both subject and assignment context. This prevents the user from losing track of where a specific item belongs.

3.9.4 Consistency Over Novelty

The design favours repeated, understandable patterns over excessive variation. Pills, cards, spacing, and stable action buttons are reused deliberately so that the application feels predictable. Primary actions continue to use the app-level accent, while subject colours are used for contextual identity.

3.9.5 Identity Through Controlled Colour

Subjects were given user-selectable accent colours from a predefined palette. This creates a stronger sense of identity without overwhelming the interface. The subject colour is inherited by related assignment and task screens, visually reinforcing the branch structure of the hierarchy.

This colour is not used for everything. A key design rule was maintained:

- **app accent colour** = primary actions,
- **subject colour** = contextual identity.

This distinction keeps the interaction language stable while still giving each subject a recognisable visual signature.

3.10 Card-Based Interface Design

The design used a consistent card-based layout. Cards were used to group related information and actions. This supports readability and improves structure.

Each card is designed to;

- present on conceptual unit,
- reduce visual clutter,
- support quick scanning,
- separate context from action.

Subject cards emphasise browsing and selection. Detail cards emphasise context and management. This distinction was important in making the UI feel more intentional.

3.10.1 Progress Representation

Progress indicators were intentionally limited to screens where they carry structural meaning. Rather than showing progress everywhere, progress was only shown on screens such as subject and assignment details.

Progress is represented using both;

- a visual progress bar,
- a numeric completion ratio such as `x / y`,
- remaining assignments/tasks text.

This makes progress easier to interpret compared to a bar alone and avoids ambiguity.

3.11 Outcome

The final architecture now emphasises;

- clear hierarchy,
- reduced redundancy,
- stronger contextual awareness,
- reusable form patterns,
- shared utility logic,
- cleaner, calmer interaction design.

As a result, the interface now accurately reflects the study workflow and provides an overall better foundation for usability and maintainability.

4 Requirements

Requirements are critical for measuring the success of the project. They are both the team's guidelines for the project and determine what users can expect from the product.

Requirements will be divided into two main parts, those being functional and non-functional requirements. Functional requirements define what the system should do, whereas non-functional requirements define how the system should perform [3]. Beyond that, functional/non-functional requirements will both be split into 4 sub-categories. "Must have", "Should have", "Could have" and "Will not have" requirements.

4.1 Functional Requirements

4.1.1 Must Have

FR1: Users must be able to create, edit, delete and view subjects

FR2: Users must be able to create, edit, delete and view assignments

FR3: Users must be able to create, edit, delete and view tasks

FR4: Users must be able to start and cancel timed study sessions

FR5: Users must be prompted to take breaks after timed study sessions

FR6: The app must connect a study session to a specific task

FR7: Subjects, assignments, tasks and study sessions must persist between app instances to avoid losing data

FR8: Tracking of studying progress

4.1.2 Should Have

FR9: Ability to set assignments and tasks as completed/not completed

FR10: Ability to set subjects as active/inactive

FR11: Notifications for important events

FR12: Users should be able to cancel active study session

FR13: Authentication should be set up to allow for cross device persistence and greater safety

4.1.3 Could Have

FR14: Users could be able to customize break and study session durations

FR15: Users could be able to pause study sessions

FR16: Authenticated users could be able to delete their accounts (conditional on FR13)

4.1.4 Will Not Have

FR-W1: Study groups will not be included, as this was deemed too large in scope for the simple design philosophy of the project

FR-W2: Calendar integration will not be included

FR-W4: Advanced analytics will not be included

4.2 Non-Functional Requirements

4.2.1 Must Have

NFR1: The project must have a functional UI where elements do not overlap and do not become unreachable

NFR2: The UI must be easy to understand / low friction

NFR3: Project must be open source on GitHub

NFR4: Simple navigation between screens

NFR5: Timer must be reliable at tracking time during study sessions and breaks

4.2.2 Should Have

NFR6: Loading states between different pages being rendered

NFR7: Page not found / error pages if something were to go wrong during interaction with, for example storage/database

NFR8: A custom UI style

NFR9: App should be prepped and ready for android deployment, and should be published on the Google Play Store if time allows for it

4.2.3 Could Have

NFR10: Display daily motivational quotes to encourage users to study

4.2.4 Will Not Have

NFR-W1: Advanced animations

NFR-W2: Be published on App Store

5 Solution

The final solution developed for Study Sprint is a mobile productivity application that combines lightweight planning, task-based study sessions, breaks, and visible progress into a single workflow. The purpose of the solution was to help students move more easily from deciding what to study to actually starting focused and efficient work.

Rather than functioning only as a timer, or as a planning tool, the application connects the two. A user can create academic structure, chose a concrete task, begin a timed study sprint, continue through short or long breaks, and return to an overview that shows both recent activity and broader progress.

This product direction was influenced by the Pomodoro technique, where work is divided into focused intervals separated by breaks. That background was relevant because the goal of the project was not simply to count time, but to support a more sustainable and actionable study rhythm. Upon reviewing literature on Pomodoro-style study intervals, we found consistent suggestions that structured work-and-break cycles can improve focus, reduce fatigue, and support sustained performance in demanding learning situations [1, 5]. For that reason, the final app was built around the idea that planning and focused study should be tightly connected rather than treated as separate tools.

5.1 Main Components of the Solution

The solution consists of four closely connected parts;

- a planning system for subjects, assignments, and tasks,
- a task-linked timer flow with focus sessions and breaks,
- a dashboard for overview, progress, and quick actions,
- persistent storage and supporting logic for user data, active sessions, and reminders.

These parts are intended to support one continuous study flow. The user should be able to define what to work on, start working without unnecessary setup, continue through breaks when appropriate, and return to a screen that still gives a clear sense of what has been done and what remains.

5.2 Planning and Study Structure

The planning part of the solution gives the user a structured way to organise academic work through subjects, assignments, and tasks. This allows larger and more abstract study obligations to be broken down into smaller, more digestible units. In practice, this matters because focused work sessions are easier to begin when the user is working from a clearly defined task rather than from a vague intention such as "study more" or "work on the course".

This means that the planning system is not only included for book-keeping. It plays a direct and central role in the app's main purpose by making it easier to convert study plans into concrete sprint starts.

5.3 Task-Linked Sprint and Break Flow

The most central part of the solution is the timer flow. In the final app, a sprint is started from a task, not from a detached timer utility. This ties focused work directly to a chosen piece of study content and makes the session feel like part of the planning workflow rather than a separate feature beside it.

The timer supports focus sessions as well as short and long breaks. In the final version of the app, the default timing model follows a familiar Pomodoro-style rhythm;

- a default focus session of 25 minutes,
- a default short break of 5 minutes,
- a default long break of 15 minutes upon completing four focus sessions.

These default values were chosen because they match the common Pomodoro model closely enough to feel familiar, whilst still fitting the project's broader goal of low-friction study support. The point was not to claim that one exact timing is universally optimal, but rather to provide a sensible default that helps the user begin immediately without having to configure the session first.

At the same time, the app was not designed as a rigid timer. The solution still allows for alternative session durations when needed. This was important because flexibility matters in real study situations, but the default path was intentionally kept simple so that customisation would not become a barrier to getting started.

5.4 Dashboard, Progress, and Guided Flow

The dashboard acts as the application's overview and next-action screen. Instead of functioning only as a static home page, it gives the user a way to resume an active session, start a sprint from upcoming tasks, and review recent study activity. This makes the dashboard part of the workflow rather than merely a simple summary screen.

The solution also includes visible progress indicators and recent-activity summaries so that study effort is easier to interpret over time. This supports motivation and orientation by showing not only what the user planned, but also what they have actually completed or spent time on.

To reduce friction further, the app also includes a guided setup and lightweight help flows. These parts of the solution are especially important for first-time users, as they make the main structure and intended study rhythm easier to understand before the app contains much user-created content.

5.5 Persistence and Supporting Functionality

The solution uses Supabase as the main backend for stored user data such as subjects, assignments, tasks, and recorded session information. This provides persistence beyond a single app instance and gives the app a clearer basis for authenticated, user-specific data.

At the same time, local storage is used for state that is more device- and interaction-specific, such as active-session recovery, study-cycle continuity, and some reminder-related information. This division is useful because not all app state serves the same role. Some data belongs to the durable academic structure, while other data belongs to the immediate behaviour of the app on the device.

The solution also includes local assignment reminders through notifications. This extends the application beyond passive planning by helping users notice approaching deadlines without requiring a more complex push-notification infrastructure.

5.6 Solution Aligned with Product Goals

The final solution fits the goals of the project because it remains focused on a small set of connected features rather than trying to become a broad productivity platform. Planning, sprint starts, breaks, reminders, and progress tracking all support the same main objective: helping students begin meaningful study work with less friction and maintain that work through a clearer rhythm of effort and recovery.

In that sense, the main strength of Study Sprint is not the quantity of features, but rather the connection between them. A timer alone does not help the user decide what to study, and a planner alone does not help the user begin. The solution developed here is intended to bridge that gap in a lightweight and student-orientated way.

6 Implementation

The implementation of Study Sprint was carried out as a gradual refinement of both product structure and technical behaviour. Rather than building every feature as an isolated part, the work focused on making the different parts of the application support one coherent study flow. This meant that navigation, CRUD functionality, timer behaviour, persistence, progress tracking, notifications, and onboarding all had to be made to work together as parts of the same product.

The application was implemented using React Native with Expo on the frontend and Supabase as the main backend for authentication and persistent user data. In practice, this gave the project a relatively fast development workflow while still allowing the application to move beyond a purely local prototype. At the same time, some state was intentionally kept local on the device, especially when that state was closely tied to temporary interaction flow rather than long-term user data.

6.1 Core Data and CRUD Implementation

A large early part of the implementation focused on establishing the application's core academic structure through subjects, assignments, and tasks. These three entities form the conceptual hierarchy of the product:

$$\textbf{Subject} \rightarrow \textbf{Assignment} \rightarrow \textbf{Task}$$

This structure was implemented through full CRUD support for each level. Users can create, edit, view, and delete subjects, assignments, and tasks, and the entities are linked together through their parent-child relationships. Subjects act as the top-level organisational unit, assignments belong to subjects, and tasks belong to assignments.

An important implementation decision was to move away from separate create and edit screens wherever the form structure was nearly identical. Instead, the project used upsert-style screens for the main entities. In this pattern, the same screen handles both creation and editing, depending on whether an existing id is passed into the route. This reduced duplicated form logic and made it easier to keep validation, styling, and layout consistent across related flows.

6.2 Navigation and Screen-Structure Implementation

As the project developed, it became clear that the original screen structure did not reflect the intended product model strongly enough. Earlier versions exposed subjects, assignments, and tasks too much as parallel top-level concepts, even though they were not conceptually independent. This created duplicated views, weaker context, and a flatter structure than intended.

To address this, the implementation shifted toward a hierarchy-driven navigation model. Subjects became the main entry point into academic content, while assignments and tasks were accessed through their parent screens instead of being treated as separate top-level browsing areas. Subject details, assignment details, and task details were each implemented more as management hubs than as static detail pages. This made it easier to preserve context and reduced unnecessary navigation noise.

The dashboard remained a cross-cutting overview screen. As development progressed, the timer was eventually integrated into the task flow as originally intended, instead of functioning as a detached utility page. This was an important decision, as the product goal was not merely to provide a timer, but to connect focused work directly to actual study tasks.

6.3 Timer, Session Flow, and Break Handling

The timer implementation developed significantly over time. Earlier versions focused mainly on starting and cancelling sessions, but later iterations expanded the flow into a more complete model with linked tasks, persistent active-session recovery, break handling, and safer finalisation rules.

A major implementation step was to move the timer into the real task workflow. Instead of starting a generic timer in isolation, the user starts a sprint from a specific task. This makes the session part of the study structure rather than a separate feature beside it. The timer was therefore implemented to receive task context, display relevant task information, and preserve that context when moving between focus sessions and breaks.

The final timer flow supports focus sessions, short breaks, and long breaks. Shared session defaults were introduced so that common durations could be reused consistently across the app. This reduced hardcoded duplication and helped align task-start, dashboard-start, and timer-entry behaviour. At the same time, the implementation kept room for alternative durations when needed, so that the timer would not become too rigid in practice. A total time window of 1 - 60 minutes was chosen based on the research material suggesting shorter sessions with frequent breaks being the most beneficial [1, 5].

Another important refinement was the implementation of a small local study-cycle model for break logic. Rather than deciding long breaks based on total historical sessions, the application tracks the current study cycle more narrowly. This makes the break flow behave more like a real continuous work rhythm and avoids misleading long-break prompts based on unrelated earlier activity.

6.4 Persistence and Session Reliability

Persistence in the project was implemented through a combination of Supabase and local device storage. Supabase was used for durable user-specific data such as subjects, assignments, tasks, and recorded session information. Local storage was used for more temporary or interaction-sensitive state, such as active-session recovery, study-cycle continuity, and notification identifiers.

This split was a deliberate implementation decision. Not all state in the application serves the same purpose. Academic structure and recorded session history belong in persistent backend storage, while temporary control state related to the current device session is better handled locally for faster and simpler recovery.

As the timer flow became more central to the app, reliability problems in session handling became more important. Different screens could detect expired, cancelled, or replaced sessions, but earlier implementations risked treating those cases inconsistently. To reduce that risk, session-finalization logic was moved into a shared lifecycle helper. This ensured that active local session state and stored session records were handled more consistently when sessions ended, expired, or were replaced.

This part of the implementation was especially important because the usefulness of the app depends heavily on users being able to trust the timer and recorded study activity. If local UI state and stored session history drift apart, the core study flow becomes less dependable.

6.5 Progress Tracking and Visual Feedback

The application also required implementation work around progress tracking so that planning and studying would feel meaningfully connected. Task completion was used as the main source of truth, and this was then used to calculate assignment-level and subject-level progress.

To support this, related tasks were grouped and evaluated so that progress could be displayed in a more understandable way. This included both visual progress bars and simple completion ratios. The implementation intentionally limited these indicators to screens where they were structurally useful, such as subject and assignment detail screens, instead of placing them aggressively throughout the app.

This choice was partly technical and partly design-orientated. From a technical side, it required consistent progress calculation based on task state. From a product perspective, it reduced visual clutter and kept browsing views calmer while still making progress visible where it mattered most.

6.6 Notifications and Reminder Logic

Another practical implementation area was assignment reminders through local notifications. These were implemented using Expo Notifications and were tied directly to assignment deadlines. The reminder logic schedules notifications for valid upcoming deadlines and updates or cancels them when assignments are edited or deleted.

An important implementation detail here was that notification identifiers were stored locally on the device rather than in the backend. This made reminder clean-up and rescheduling easier while keeping the reminder system lightweight. The project, therefore, avoided a more complex push-notification architecture since that would have added scope without being necessary for the intended use case.

Because notification behaviour depends on native capabilities, this part of the project was tested through an Expo development build rather than relying only on Expo Go. That made the implementation more realistic and gave better confidence in behaviour on Android devices.

6.7 Android Internal Testing and Delivery Preparation

Toward the end of the project, the application was also prepared for internal testing through Google Play Console. This required uploading the Android app bundle, configuring an internal test track, adding at least one tester email address, and publishing the internal test so that testers could gain access to the application. The internal test listing can be found at: https://play.google.com/apps/internaltest/4701094029771195463

After the internal test was published, Google Play Console provided a direct link to the app listing in Google Play. At this stage, the listing used a temporary name and was not publicly searchable. Access was limited to users with the link and the configured tester access. This made it possible to test the distribution flow in a more realistic way without treating the application as a full public release.

This step was part of the practical delivery work rather than the product solution itself. It helped confirm that the application could be packaged, uploaded, and distributed through the expected Android testing channel, which was important for validating the project as more than a local development prototype.

6.8 Onboarding and Low-Friction Flow Improvements

Later implementation work focused more strongly on reducing friction for first-time users and making the main study flow easier to enter. This included guided setup behaviour, clearer help flows, faster sprint-start paths, and more direct routing from dashboard and task screens into the timer.

One specific refinement was to make the default sprint flow faster by presenting a sensible default focus duration immediately instead of forcing the user through a configuration step every time. A custom-duration path was still preserved, but it became an optional side path rather than the default interaction.

The onboarding flow was also adjusted so that incomplete users are routed into setup more consistently, and the first guided sprint was shortened to a small demo session rather than a full, normal-length focus block. This was done to support the product goal that the app should be understandable and usable quickly, without making the first interaction feel unnecessarily heavy.

6.9 Implementation Outcome

In implementation terms, the final application became much more than a basic CRUD prototype. The main technical result was a mobile application where planning structure, task ownership, timed study sessions, breaks, reminders, progress tracking, and persistence all support the same study workflow.

Several of the most important implementation choices were therefore not about adding more features, but about improving coherence between features that already existed. The shift toward hierarchy-driven navigation, task-linked timers, shared lifecycle handling, limited but meaningful progress feedback, and lower-friction session starts all helped move the application closer to its intended product identity.

7 Testing

For the testing section of this project, the group focused mostly on creating tests that verify CRUD behaviour of subjects, assignments and tasks as well as an auth guard test. The reason that these specific tests were chosen is because they are core to the requirements while also being relatively obvious on how they should be set up.

An important note for all of these tests is that the tests should not actually interact with external dependencies like `expo-router`, `AsyncStorage`, `expo-notifications` or APIs like `supabase`. Instead, they should be mocked/simulated using `jest.mock`.

The reason for this is because the tests are not meant to verify whether third party behaviour from libraries or other services work, but rather whether the actual logic of the code works correctly. By mocking the dependencies these tests become more predictable, faster to run and not as dependent on external issues that could occur. There is also an added bonus of not affecting the dependencies, like for example adding useless data to the database.

The basic idea of a test with jest is that certain predefined expectations have to be met. If they are the test passes, if not the test fails.

7.1 Subject Tests

7.1.1 Create Subject Test

For the Create Subject test, the goal was to verify that a new subject can be created through the creation branch of the `handleSubmit` function in `upsertSubject.tsx`. Since this component normally relies on `supabase` and `expo-router`, these have to be mocked.

The expectations for this test are as follows:

- `supabase.from` has been called with `subjects`:
  This confirms that the correct table was targeted.
- `insert` has been called with an object containing subject title and authenticated user id:
  This confirms that the values entered by the user are the values being inserted.
- `select` and `single` have been called:
  This confirms that the subject is returned after being inserted.

- `router.back` has been called:
  This confirms that the user would be routed back after subject creation.

If all of these expectations pass, the test passes.

7.1.2 Update Subject Test

For the Update Subject test, the goal was to verify that an already existing subject can be updated through the update branch of the `handleSubmit` function in `upsertSubject.tsx`. Since this component normally relies on `supabase` and `expo-router`, these have to be mocked using `jest.mock`.

The expectations for this test are as follows:

- `supabase.from` has been called with `subjects`:
  This confirms that the correct table was targeted.
- `select` has been called:
  This confirms that the component attempts to load the subject, before trying to update it.
- `update` has been called with an object containing subject title and authenticated user id:
  This confirms that the changed subject values are submitted for update.
- The update chain `eq` has been called with subject id:
  This confirms that the correct subject is updated.
- `router.back` has been called:
  This confirms that the user would be routed back after a subject is updated.

If all of these expectations pass, the test passes.

7.1.3 Delete Subject Test

For the Delete Subject test, the goal was to verify that an already existing subject can be deleted through the `DeleteSubject` function in `viewDetailsSubject.tsx`. Since this component normally relies on `supabase` and `expo-router`, these have to be mocked using `jest.mock`. It also relies on React Native's `Alert` API, which is spied on using `jest.spyOn`.

The expectations for this test are as follows:

- `alertSpy` has been called with the expected title, message and buttons:
  This confirms that the subject deletion confirmation alert appears.

- The alert buttons are defined:
  This confirms that usable buttons appear on the alert.
- The confirm delete button has an `onPress` function:
  This confirms that subject deletion confirm button has functionality attributed to it.
- `supabase.from` has been called with `subjects`:
  This confirms that the correct table was targeted.
- `delete` has been called:
  This confirms that the subject deletion supabase command has started.
- The delete chain `eq` has been called with subject id:
  This confirms that the correct subject is deleted.
- `router.back` has been called:
  This confirms that the user would be routed back after a subject is deleted.

If all of these expectations pass, the test passes.

7.2 Assignment Tests

7.2.1 Create Assignment Test

For the Create Assignment test, the goal was to verify that a new assignment can be created through the creation branch of the `handleSubmit` function in `upsertAssignment.tsx`. Since this component normally relies on `supabase`, `expo-router`, `AsyncStorage` and `expo-notifications` these have to be mocked.

The expectations for this test are as follows:

- `supabase.from` has been called with `assignments`:
  This confirms that the correct table was targeted.
- `insert` has been called with an object containing assignment title, authenticated user id and its associated subject id:
  This confirms that the values entered by the user are the values being inserted.
- `select` and `single` have been called:
  This confirms that the assignment is returned after being inserted.
- `router.back` has been called:
  This confirms that the user would be routed back after assignment creation.

If all of these expectations pass, the test passes.

7.2.2 Update Assignment Test

For the Update Assignment test, the goal was to verify that an already existing assignment can be updated through the update branch of the `handleSubmit` function in `upsertAssignment.tsx`. Since this component normally relies on `supabase`, `expo-router`, `AsyncStorage` and `expo-notifications` these have to be mocked.

The expectations for this test are as follows:

- `supabase.from` has been called with `assignments`:
  This confirms that the correct table was targeted.
- `select` has been called:
  This confirms that the component attempts to load the assignment, before trying to update it.
- `update` has been called with an object containing assignment title, authenticated user id and its associated subject id:
  This confirms that the changed assignment values are submitted for update.
- The update chain `eq` has been called with assignment id:
  This confirms that the correct assignment is updated.
- The update chain `single` has been called:
  This confirms that the updated assignment is returned.
- `router.back` has been called:
  This confirms that the user would be routed back after an assignment is updated.

If all of these expectations pass, the test passes.

7.2.3 Delete Assignment Test

For the Delete Assignment test, the goal was to verify that an already existing assignment can be deleted through the `DeleteAssignment` function in `viewDetailsAssignment.tsx`. Since this component normally relies on `supabase` and `expo-router`, these have to be mocked using `jest.mock`. It also relies on React Native's `Alert` API, which is spied on using `jest.spyOn`.

The expectations for this test are as follows:

- `alertSpy` has been called with the expected title, message and buttons:
  This confirms that the assignment deletion confirmation alert appears.
- The alert buttons are defined:
  This confirms that usable buttons appear on the alert.

- The confirm delete button has an `onPress` function:
  This confirms that assignment deletion confirm button has functionality attributed to it.
- `supabase.from` has been called with `assignments`:
  This confirms that the correct table was targeted.
- `delete` has been called:
  This confirms that the assignment deletion supabase command has started.
- The delete chain `eq` has been called with assignment id:
  This confirms that the correct assignment is deleted.
- `router.back` has been called:
  This confirms that the user would be routed back after an assignment is deleted.

If all of these expectations pass, the test passes.

7.3 Task Tests

7.3.1 Create Task Test

For the Create Task test, the goal was to verify that a new task can be created through the creation branch of the `handleSubmit` function in `upsertTask.tsx`. Since this component normally relies on `supabase`, `expo-router` and the `CheckAssignmentCompletio` function, these have to be mocked.

The expectations for this test are as follows:

- `supabase.from` has been called with `tasks`:
  This confirms that the correct table was targeted.
- `insert` has been called with an object containing task title, authenticated user id and its associated assignment id:
  This confirms that the values entered by the user are the values being inserted.
- `select` and `single` have been called:
  This confirms that the task is returned after being inserted.
- `CheckAssignmentCompletion` has been called with assignment id:
  This confirms that assignment completion is updated upon a new task being created.
- `router.back` has been called:
  This confirms that the user would be routed back after task creation.

If all of these expectations pass, the test passes.

7.3.2 Update Task Test

For the Update Task test, the goal was to verify that an already existing task can be updated through the update branch of the `handleSubmit` function in `upsertTask.tsx`. Since this component normally relies on `supabase`, `expo-router` and the `CheckAssignmentCompletio` function, these have to be mocked.

The expectations for this test are as follows:

- `supabase.from` has been called with `tasks`:
  This confirms that the correct table was targeted.
- `select` has been called:
  This confirms that the component attempts to load the task, before trying to update it.
- `update` has been called with an object containing task title, authenticated user id and its associated assignment id:
  This confirms that the changed task values are submitted for update.
- The update chain `eq` has been called with task id:
  This confirms that the correct task is updated.
- `CheckAssignmentCompletion` has been called with assignment id:
  This confirms that assignment completion is updated upon a task being updated.
- `router.back` has been called:
  This confirms that the user would be routed back after a task is updated.

If all of these expectations pass, the test passes.

7.3.3 Delete Task Test

For the Delete Task test, the goal was to verify that an already existing task can be deleted through the `DeleteTask` function in `viewDetailsTask.tsx`. Since this component normally relies on `supabase`, `expo-router` and the `CheckAssignmentCompletion` function, these have to be mocked. It also relies on React Native's `Alert` API, which is spied on using `jest.spyOn`.

The expectations for this test are as follows:

- `alertSpy` has been called with the expected title, message and buttons:
  This confirms that the task deletion confirmation alert appears.
- The alert buttons are defined:
  This confirms that usable buttons appear on the alert.

- The confirm delete button has an `onPress` function:
  This confirms that task deletion confirm button has functionality attributed to it.
- `supabase.from` has been called with `tasks`:
  This confirms that the correct table was targeted.
- `delete` has been called:
  This confirms that the task deletion supabase command has started.
- The delete chain `eq` has been called with task id:
  This confirms that the correct task is deleted.
- `CheckAssignmentCompletion` has been called with assignment id:
  This confirms that assignment completion is updated upon a task being deleted.
- `router.back` has been called:
  This confirms that the user would be routed back after a task is deleted.

If all of these expectations pass, the test passes.

7.4 Auth Guard Test

For the Auth Guard test, the goal was to verify that a valid session exists before allowing the user to access `index.tsx` and `subject.tsx`. This is important because accessing those components/pages would mean that the user id would not be defined for CRUD behaviour for subjects, assignments and tasks. This in turn would cause a lot of bugs and undefined behaviour.

Since the `_layout` component that enforces authentication normally relies on `supabase`, `expo-router`, `expo-notifications` and the `getSetupStatus` function, these have to be mocked. There is also a `beforeEach` function that resets important data fields before each test is run.

The first test sets `session: null`. The expectation based on this is that `redirect:/login` is rendered. If this passes it is confirmed that unauthenticated users cannot access the core functionality.

The second test sets session to be a fake authenticated user. The expectation based on this is that user should be redirected to a mocked `TabLayout`. If this passes it is confirmed that authenticated users can access the core functionality.

If both of these tests pass, the auth guard behaviour works as intended and the test suite passes.

8 Discussion

The development of Study Sprint shows both the strengths and limitations of building a small productivity application around one tightly scoped user problem. The project did not aim to compete with broad commercial productivity platforms on feature count. Instead, it aimed to reduce the gap between planning study work and actually starting it. In that sense, the most important question is not whether the app includes many features, but whether the implemented features support one coherent and dependable study flow.

Overall, the final result suggests that the project moved substantially closer to that goal over time. The strongest improvement was not the addition of one isolated feature, but the gradual alignment between planning structure, timer behaviour, session persistence, progress visibility, and onboarding. Earlier versions of the app already contained the core building blocks, but later revisions made those parts reflect the original product direction more clearly and work more convincingly as one product rather than as loosely connected components.

8.1 Strength of the Hierarchy-Driven Product Model

One of the clearest lessons from the project was that the application's underlying hierarchy mattered more than first assumed. The conceptual model of *Subject* → *Assignment* → *Task* was not only a database relationship but also an important usability principle. When assignments and tasks were exposed too much as standalone top-level concepts, the app became flatter, more repetitive, and less clear. Important context was lost, and the user could more easily lose track of what a specific task belonged to.

Restructuring the app around that hierarchy improved both navigation and meaning. Subjects became the natural entry point into academic content, while assignments and tasks became progressively deeper parts of the same flow. This made the product feel more intentional and better aligned with its actual purpose. In discussion terms, this supports the broader argument that architectural clarity in a small application is not only a technical concern but also a user-experience concern.

8.2 Importance of Task-Linked Study Sessions

Another important outcome was that the final implementation came to reflect one of the project's original intentions more accurately: the timer was always meant to support concrete study tasks rather than function as a detached utility. This reflects one of the central ideas behind the project, namely that planning and focused work should support each other directly. A generic timer can measure time, but it does

not necessarily help the user decide what to work on. By contrast, a task-linked sprint flow makes the act of starting a session more concrete and meaningful in the context of study work.

This also strengthens the product's unique value relative to broader productivity tools. Study Sprint does not attempt to replace general-purpose planners or offer advanced analytics. Its strength lies instead in the connection between defining study structure and beginning focused work quickly. From that perspective, the later timer-related revisions should not be understood as a change in product direction, but as a process of bringing the implementation into better alignment with the intended identity of the app.

8.3 Reliability as a Central Product Requirement

Reliability also became one of the most important concerns during development. This was especially visible in the timer and session lifecycle. Once the app began storing active sessions, supporting break cycles, and recovering state across screens, inconsistencies between local state and stored session data became a real product risk. A timer-based application quickly loses credibility if it cannot be trusted to reflect the user's actual study activity.

For that reason, the later work on shared session finalisation, active-session recovery, and break-cycle logic was highly significant. These changes may seem less visible than larger UI changes, but they directly support one of the most important, critical product attributes identified earlier in the report: reliability. In practical terms, this means that some of the most valuable work in the project was not the addition of new features, but the correction of weak interactions between existing features.

8.4 Tradeoffs in Persistence and Scope

The chosen persistence model also represents a clear tradeoff. Supabase was used for durable user data such as subjects, assignments, tasks, and recorded sessions, while local device storage was used for temporary or device-specific state, such as active-session recovery, study-cycle continuity, and notification identifiers. This was a pragmatic and appropriate design for the scope of the project.

The benefit of this split is that it keeps the app practical without introducing unnecessary backend complexity. At the same time, it also creates a boundary that must be handled carefully. When one part of the app depends on local control state and another part depends on backend records, consistency problems can arise if the transition between them is not carefully managed. The later reliability fixes suggest

that this tradeoff was acceptable, but only when paired with more explicit lifecycle handling.

8.5 Usability, Onboarding, and Low-Friction Design

The project also highlights that usability problems in small applications often arise from friction rather than from missing functionality. In this case, several later revisions focused on making the main study flow easier to enter: guided setup was strengthened, default sprint durations were made easier to accept immediately, and help flows were adjusted so that the intended rhythm of focus and breaks became easier to understand.

These changes support the original product vision well. The main goal of the app was not to maximise customisation or feature depth, but to help users move quickly from intention to action. The discussion that follows from this is that a low-friction default path can be more valuable than a more flexible but slower interaction model, especially in a study-orientated application where hesitation and setup overhead directly work against the product's purpose.

The claim that the application has a low-friction and intuitive design can also be supported through Nielsen's usability heuristics. Nielsen's heuristics are broad principles for interaction design and are commonly used to evaluate whether an interface is likely to be understandable, predictable, and usable [4]. Study Sprint was never formally tested through a complete heuristic evaluation with several reviewers, but several of the final design decisions align closely with these principles.

First, the app supports *visibility of system status* by showing the active timer state, study session progress, completion ratios, and recent activity. This helps users understand what is currently happening and what progress has been made. Second, the app follows *match between system and the real world* by organising academic work through familiar student concepts such as subjects, assignments, and tasks. Third, *user control and freedom* are both supported through options such as cancelling active sessions, editing created entities, and ease of navigation.

The design also follows *consistency and standards* by reusing cards, pills, action buttons, spacing, and upsert-based forms across similar screens. *Recognition rather than recall* is supported by keeping the parent context visible as users move through the hierarchy, for example, by showing subject and assignment context on deeper task screens. The default sprint flow also supports *flexibility and efficiency of use* because users can start quickly with a sensible default duration while still having the option to choose alternative durations when needed.

The app follows *aesthetic and minimalist design* by reducing top-level navigation, removing redundant assignment and task tabs, and limiting progress indicators to screens where they provide useful context. Help and documentation are supported through onboarding and lightweight help flows that explain the intended study rhythm without requiring the user to read a large manual. Together, these design choices support the argument that the interface was intentionally shaped to be understandable and low-friction, although a full user test would still be needed to validate this with real users.

8.6 Limitations of the Project

Despite the strengths of the final result, the project also has clear limitations. First, the scope remained relatively narrow. This was intentional and appropriate, but it still means that the product does not attempt to cover many surrounding needs that broader productivity platforms address, such as advanced analytics, collaboration, calendar integration, or richer cross-device notification infrastructure. This area would likely be the projects next development path, given a bigger development window following the implementation of all remaining could-have requirements

Second, the development process remained strongly iterative until late in the project. That was useful for improving the product, but it also meant that some parts of the application changed several times before reaching a more stable form. In report terms, this suggests that the project succeeded more as a process of refinement than as a case of implementing a fully settled design from the start.

Third, the testing picture appears mixed. The report includes structured tests for important CRUD and guard behaviour, together with repeated use of linting, TypeScript checks, manual testing, and development-build verification. Even so, not all important qualities of the app were verified in the same way. Interaction-heavy behaviour such as timer flow, break handling, onboarding transitions, and notification timing depended heavily on manual validation. This is understandable for the project scope, but it still represents a limitation compared to a larger and more thoroughly automated testing strategy.

Fourth, the current state of the project is dependent on Supabase and their free-tier rate limits. One obvious concern is the rate of emails being limited to 2 per hour, meaning that within any given hourly timeframe, only two new accounts can be registered due to the requirement of a confirmed email address before a user can log in. This is also an area of expansion for the project; however, the group decided to forego the cost of a subscription, given that the app, in its current state, is used as a proof-of-concept rather than an actual avenue of business.

8.7 Google Console and Internal Testing

The Android internal testing process also showed that deployment readiness is a separate concern from having a working local application. Preparing the app for Google Play required package identity, signing, app-bundle upload, and tester access to be handled correctly. This did not change the product functionality itself, but it helped validate that the application could be distributed through a realistic Android delivery channel.

8.8 Documentation Lapses

A slight misstep was that the group did not maintain formal time sheets throughout the project. The requirement for time sheets was discovered late, after a vast majority of the product development had been completed. As a result, the group cannot provide a detailed and reliable hour-by-hour record of the work process.

This weakens the documentation of workload distribution and makes it harder to evaluate the exact amount of time spent. However, the group has still documented the development process through the method section, collaboration description, implementation history, testing section, and final product outcome. The work was also supported by regular communication, meetings, source control activity, and shared report writing. Member contributions can be roughly estimated given the git commit history of each member, although this does not provide a realistic estimate of time spent. Group members also wrote daily work summary notes to document the work that had been carried out and the current state of the development branch at the time of writing. This makes it possible to determine workload distribution based on additions to the codebase. Hourly distribution, however, is difficult to estimate given only work notes and commit history.

The group also did not keep any formal ticket list using a Scrum based system. The reasoning was that the app had such limited scope that deploying a full-scale ticket tracking system was deemed to add more complexity to the project than it would alleviate. Given that the planned features for the app were meant to be few but of high quality, the group had no issues staying on top of what each member was working on at any given time, nor did it introduce difficulties in keeping track of future work. With the added benefit of writing work summary notes, the group can also reliably prove the work and contributions of each member. In short; the lack of a full-scale ticket tracking system did not hinder the project in any capacity.

8.9 Overall Evaluation

Taken as a whole, Study Sprint appears successful primarily because the project stayed disciplined about its central problem. The strongest parts of the final application are those where structure, study sessions, breaks, reminders, and progress all reinforce the same user goal. The weaker parts are not signs that the project lacked ambition, but reminders that even a small mobile product becomes complex once reliability, persistence, and user flow are treated seriously.

The most important discussion outcome is therefore that the project demonstrates the value of coherence over breadth. Study Sprint became stronger when the team removed redundancy, tightened the hierarchy, linked sessions to real tasks, and refined the timer lifecycle until it better supported dependable use. That does not mean the application is complete in every sense, but it does mean that the final result reflects its stated product vision more clearly than the earlier prototype versions did.

9 Conclusion

The goal of *Study Sprint* was to create a focused mobile application that helps students organise academic work and move more easily into structured study sessions. The final product addresses this goal by combining subjects, assignments, tasks, timed focus sessions, breaks, progress tracking, reminders, and persistence into one connected workflow.

The project shows that the value of the application lies mainly in how these parts work together. A planner alone would not solve the problem of starting focused work, and a timer alone would not help the user decide what to study. By linking study sessions directly to tasks, the app makes the transition from planning to action more concrete. This supports the original product vision, where simplicity, low friction, reliability, and realistic scope were more important than building a broad productivity platform.

During development, the application also became more coherent through several important refinements. The screen structure was aligned more closely with the hierarchy of *Subject* $\rightarrow$ *Assignment* $\rightarrow$ *Task*, the timer flow was implemented around the task workflow as originally intended, and session reliability was improved through more consistent lifecycle handling. The final design also supports several usability principles, including visible system status, familiar student-oriented concepts, consistent interaction patterns, and a low-friction default path into study sessions.

There are still limitations. The project remained intentionally narrow in scope, and some behaviour, especially around timer interaction, onboarding, notifications, and break handling, depended heavily on manual verification. The product is also still best understood as a proof-of-concept rather than a production service, especially because it depends on Supabase free-tier limits and does not yet include a larger automated testing strategy, advanced analytics, calendar integration, or richer cross-device behaviour. In addition, the lack of formal time sheets and a formal ticket-tracking system weakened the precision of the process documentation, even though the group could still document work through the report, source control, communication, meetings, and development history.

Overall, *Study Sprint* can be considered a successful outcome within the constraints of the project. It delivers a working and coherent study-support application that reflects the original vision more clearly than the early prototype. The final result is not the largest possible productivity app, but it is a focused solution that connects planning, studying, breaks, and progress in a way that is useful for the intended student user.

10 Individual Reports

10.1 Christopher Sanden

My individual contribution to *Study Sprint* covered several central parts of the project, but the largest part of my work was connected to the timer, session flow, onboarding, and final delivery preparation. The commit history attributes 44 commits to me, including the first timer implementation, later task-flow integration, dashboard and session improvements, break-cycle logic, onboarding refinements, signup-confirmation deployment, test updates, and Android delivery preparation. The most relevant work is supported by the timer work notes from 21 April to 5 May and commits such as `d50301c`, `666bdc1`, `c74062c`, `b437643`, `907fa18`, `245b6db`, `2bb2ac6`, `9bb3bb1`, and `419463e`.

My first major contribution was creating the original timer feature. In commit `d50301c`, I added the first functional timer screen with duration selection, start behaviour, running timer state, and animated visual feedback. In the following iterations, I developed the timer into a more complete study-session interaction. This included countdown behaviour, measured layout handling, duration locking while a session was running, a deliberate hold-to-cancel flow, and cleanup of the internal timer structure. The timer eventually relied on several separate kinds of state, including selected duration, running state, countdown text, layout height, progress animation, and cancellation state. Separating these concerns was important because the timer later had to survive routing changes, cancellation, recovery, and integration with persistent session data.

A key part of this work was making the timer maintainable enough to become a core product feature rather than a fragile prototype. In commit `666bdc1`, I refactored the timer code by extracting repeated cleanup logic, grouping refs by responsibility, and making the render structure match the visible screen layers more closely. This was especially important because the timer used intervals, timeouts, animation refs, and cancellation flags. Keeping those responsibilities clear made the later integration work safer and easier to reason about.

The most important product-level change I made was moving the timer into the actual task workflow. In commit `c74062c`, I moved the timer from the tab navigator into the task stack and made it open from a selected task. The timer could then receive a task id, load the matching task from Supabase, display task information during the sprint, and preserve an active sprint locally through an end time. This changed the timer from a generic utility into a study-session feature tied directly to the user's planned work, which better matched the product vision.

I also extended the timer into a more durable session model. In commit `b437643`, I connected active local sprint state to database-backed sprint sessions and added task-level study-time tracking. This meant that completed, expired, or cancelled sessions could contribute elapsed time back to the relevant task instead of only changing temporary UI state. I also handled failure cases around session creation, finalisation, cancellation, and restore behaviour so the local timer state and stored session history would not drift apart.

My work also included heavily contributing to the dashboard and progress experience. I added support for reopening active sprints from the dashboard, showing remaining session time, displaying upcoming deadline tasks with subject and assignment context, and marking tasks as completed from dashboard cards. I also fixed a dashboard issue where upcoming tasks disappeared whenever an active sprint existed. These changes made the dashboard more useful as a daily study overview instead of only being a static landing page.

Later in the project, I worked on the focus-and-break cycle. I introduced shared session defaults in `lib/sessionDefaults.ts` so focus duration, short breaks, long breaks, and long-break frequency could be reused consistently across the timer, task details, and dashboard. I also implemented a local study-cycle model so the app could decide whether the next break should be short or long based on the current continuous study run rather than unrelated historical sessions. This made the session flow closer to a real study rhythm: focus, break, continue the same task, or return to the dashboard.

As the session flow became more connected to the rest of the app, I added a shared lifecycle helper in `lib/sessionLifecycle.ts`. This centralised how active sessions are finalised when they expire, are replaced, or are cleared from other screens. This was one of my more important reliability contributions because it reduced duplicated logic across the timer, dashboard, task details, and setup flow, and lowered the risk of the UI and database recording different versions of the same session.

I also contributed to the first-time-user and account-confirmation flow. In commit `907fa18`, I added a guided setup route that helps new users create their first subject, assignment, and task before starting a sprint. Later, in commit `9bb3bb1`, I added shared setup-completion logic in `lib/setupStatus.ts` so incomplete users are routed into setup more consistently. I also changed the first setup sprint into a short demo timer, making the first interaction easier to test and understand.

Outside the app itself, I improved the signup confirmation loop. I added a small static confirmation page under `deploy/signup-confirmation` in the source direc-

tory, served through `nginx` with Docker Compose and hosted on my VPS behind the existing reverse-proxy setup. I also corrected the Caddy routing target so the page resolved correctly, and customised the Supabase confirmation email into a cleaner HTML email while keeping the required confirmation-link placeholder. This made account creation feel more complete and presentable from signup through email confirmation.

Toward the end of the project, I contributed to verification and delivery preparation. I used TypeScript checks, linting, manual runtime testing, and database inspection during timer and session work, because interaction-heavy behaviour could not be fully verified by static checks alone. I also updated the test setup and several test files in commit `419463e` so they matched the newer logic across auth, subject, assignment, and task flows. Finally, I adjusted the Android package name in `app.json` to match Google Play Console expectations and added the APK artifact for delivery.

Overall, my contribution was broad, but it was tied together by one main goal: making the app feel like a coherent study tool rather than a set of separate screens. I worked on the timer, task integration, session persistence, dashboard visibility, break-cycle behaviour, onboarding, signup confirmation, testing, and Android delivery. Strategically, this strengthened the parts of the project most closely connected to the product vision: helping users move from planning study work to actually starting and tracking focused study sessions.

10.2 Fabian Haukedal Johansen

A large part of my work was designing the application around the hierarchy:

Subject → Assignment → Task

I used this as the foundation for my navigation and screen design. My goal was to make the interface reflect the conceptual model of the application, where subjects serve as the main entry-point, while assignments and tasks are nested entities.

I also helped define the top-level navigation structure:

- Dashboard
- Subjects
- Timer

I designed several of the app's main screens with a focus on clarity, hierarchy and consistency:

- the **Subjects** screen as the main entry point into studies,
- the **Subject Details** screen as the hub for a subject and its assignment,
- the **Assignment Details** screen as the hub for assignment and its tasks,
- the **Task details** screen with context from both the parent assignment and subject,
- all **Upsert** screens for subjects, assignments and tasks.

I also worked on the visual layout of cards, metadata, actions, and progress displays, making screens easier to read and navigate.

I made several design choices throughout the development of the app:

- using card-based layouts to group related information,
- using pills for compact metadata such as subject, deadline, status, and parent information on deeper screens,
- reducing unnecessary visual noise in list screens,
- keeping primary actions visually consistent across the app.

Originally all subjects had the same, calm, color accent. This led to subjects sometimes being hard to distinguish. So with that in mind I introduced subject-specific accent colors with the aim of making subjects easier to distinguish and to provide them some identity. I also made the decision to make this color reused in related

assignment and tasks screens. The purpose for this was to give assignments and tasks additional context.

Throughout the development, my aim was to minimize redundancy, not just in the UI, but also in the code-base. That way I make the UI more relevant and the code-base easier to maintain in the future. That's why I decided to design a more reusable structure for the app's forms. Instead of keeping separate create and edit screens with mostly duplicated logic, I worked with **upsert**-style forms that handles both creation and editing, depending on whether an entity ID exists. This was applied to subjects and extended to assignments and tasks.

I also made shared utility code for things such as:

- date formatting,
- subject color handling,
- context-aware metadata display.

This helped make the implementation more consistent, easier to maintain, and also less redundant.

In addition to my higher-level design decisions, I implemented a large part of the frontend logic and UI behaviour. This included:

- screen layout and styling using NativeWind,
- route structure and navigation updates,
- hierarchical context display between subjects, assignments, and tasks,
- progress display logic,
- creation and editing flows for subjects, assignments and tasks,
- improvements to how dates and timestamps are displayed,
- small parts of the assignment reminder and notification-related flow

Overall my contribution was largely focused on designing and implementing the app's architecture and interface structure. I made the system hierarchy visible in the UI, the navigation more logical, and I built a clear and maintainable structure for the main screens and form flows in the application.

10.3 Teodor Salvesen

For this project my main contributions were the app's early core functionality and the testing infrastructure. At the beginning of the project our vision was very different to what the finished product actually became. At that point the idea was some sort of timer connected to tasks. That idea did make it through to the finished product, but subjects and assignments had not even been thought of yet. This was in fact something I suggested early on and decided to implement in one of my first commits.

The resulting infrastructure that I had implemented after those initial commits ended up being the foundation of the subjects -> assignments -> tasks structure that is the app's core functionality used in the final versions of the app. With these came CRUD for all three and the supabase setups needed to actually make everything work. All three tables were also connected to authentication. Which means that user creation and login also has to be set up around this time. Finally I set up a shared styling and initial tab navigation, both of which would be overwritten in later versions.

Next up I created some of the side functionality. This included notifications, Async-Storage for notifications, the first iteration of assignment progress tracking, and auth guard and major reworking of folder/file structure for the subjects → assignments → tasks flow. My final major contribution to this project were the testing suites. I designed all CRUD tests for subjects, assignment and tasks, and the auth guard test.

Alongside all of these development features that I implemented, I also documented my work through the project notes.

References

[1] Francesco Cirillo. *The Pomodoro Technique: The Acclaimed Time-Management System That Has Transformed How We Work*. Originally self-published in a different form in 2006. New York: Currency, 2018. ISBN: 9781524760700. URL: https://www.penguinrandomhouse.com/books/555557/the-pomodoro-technique-by-francesco-cirillo/.

[2] Fabian Haukedal Johansen and Teodor Salvesen and Christopher Sanden. *Project Vision*. Project document, included as Appendix. 2026.

[3] GeeksforGeeks. *Functional and Non Functional Requirements*. Accessed: 2026-04-28. Apr. 2026.

[4] Heurio. *Nielsen's 10 Usability Heuristics*. Accessed: 2026-05-09. 2026. URL: https://www.heurio.co/nielsens-10-usability-heuristics.

[5] Eren Ogut. "Assessing the efficacy of the Pomodoro technique in enhancing anatomy lesson retention during study sessions: a scoping review". In: *BMC Medical Education* 25 (2025). URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC12532815/.