

Development of a Social Ski Trip Application

Intelligent Matching, Production Engineering, and an Agentic Development Workflow

Ben Aissa Amen Allah

Academic Year:	2025–2026
Company:	Tanit Consulting AB (FrostLink)
Internship Duration:	July 1, 2026 – August 31, 2026
Internship Supervisor:	Issam Ben Taieb



Submitted in partial fulfillment of the requirements
for the internship in Computer Science Engineering

September 2026

Dedications

To my family, for their support throughout these two months, and to the entire FrostLink team.

Acknowledgments

I would like to thank Issam Ben Taieb, my supervisor at FrostLink, for his guidance and for trusting me with production code from the first week. I also thank my teammates Abdennour and Mariam, whose reviews, test sessions, and feedback shaped most of the work described in this report, and the rest of the FrostLink team for the environment they created. Working in a small team that shipped continuously taught me more than any single project I had built before.

Contents

List of Abbreviations viii

General Introduction 1

1 Project Definition and Management 2
- 1.1 Introduction . . . 2
- 1.2 General Context . . . 2
- 1.3 Host Organization . . . 3
- 1.4 Social Trip Matching and Community Platforms . . . 3
- 1.5 Problem Statement . . . 4
- 1.6 Objectives and Scope . . . 5
- 1.7 Market Study and Existing Solutions . . . 5
- 1.8 Requirements Analysis . . . 6
- 1.9 Development Methodology . . . 7
- 1.10 Project Planning and Timeline . . . 8
- 1.11 Conclusion . . . 9

2 System Design and Implementation 10
- 2.1 Introduction . . . 10
- 2.2 System Architecture . . . 10
- 2.3 Technology Justification . . . 11
- 2.4 System Design . . . 12
- 2.5 Database Design . . . 13
- 2.6 User Interface Design . . . 15
- 2.7 Backend Implementation . . . 17
  - 2.7.1 Cloud Functions . . . 17
  - 2.7.2 Firestore Security Rules . . . 18
  - 2.7.3 Push Notifications and Moderation . . . 19
- 2.8 Frontend Implementation . . . 19
  - 2.8.1 Core Screens and Features . . . 19
  - 2.8.2 Build Pipeline and Releases . . . 20
- 2.9 Conclusion . . . 20

3 Beyond Code: Engineering Intelligence, Modern Workflows, and Production Reality 21
- 3.1 Introduction . . . 21
- 3.2 Intelligent Trip Matching . . . 21
  - 3.2.1 The Matching Challenge . . . 21
  - 3.2.2 Algorithm Inputs . . . 21

3.2.3 Scoring Algorithm Design . . . 22
3.2.4 Broadening and the Score Floor . . . 23
3.2.5 Iterative Refinement . . . 24
3.2.6 Limitations and Future Work . . . 25
3.3 Building an Agentic Development Workflow (Piste) . . . 25
3.3.1 Motivation . . . 25
3.3.2 Architecture . . . 26
3.3.3 Human-in-the-Loop Design . . . 27
3.3.4 Adversarial Verification . . . 27
3.3.5 Use on FrostLink . . . 27
3.3.6 Limitations and Lessons . . . 27
3.4 Production Engineering: Theory vs Reality . . . 28
3.4.1 Firebase Rules Reconciliation . . . 28
3.4.2 Build and Release Engineering . . . 28
3.4.3 Account Deletion and Auth Edge Cases . . . 29
3.4.4 Cross-Platform Fragmentation . . . 29
3.5 Evaluation and Results . . . 30
3.5.1 Security Rules and Functions . . . 30
3.5.2 Matching Validation . . . 30
3.5.3 Performance and Release Health . . . 31
3.6 Conclusion and Reflections . . . 31

General Conclusion **32**

Bibliography **33**

A Matching Algorithm Excerpts **35**

B Firestore Security Rules Excerpts **38**

C Additional Screenshots **40**

D Piste: Commands, Agents, and Skills **42**

List of Figures

1.1 The FrostLink application icon. 3
1.2 The Scrum framework: the Product Backlog feeds Sprint Planning, the Sprint produces an Increment, and the Sprint Review and Retrospective close the loop. 8
1.3 The development workflow: every change moved from a ticket through a plan, implementation, device testing, and review before it reached main. . . 8
1.4 Project phases and releases. Each column is one calendar week; column 1 is the week of June 29, 2026, and column 12 is the week of September 14, 2026. 9

2.1 The system architecture. The client reads and writes through callables and listeners, calls Google Maps and Places directly, and the functions call the hosted moderation API and hand notifications to the Expo push service. . 11
2.2 Roles and the main actions attached to them. The generalization arrows show that trip admins, network admins, and the superadmin are members with additional rights, and a member who creates a trip becomes its trip admin. 12
2.3 The join flow for a public trip or an event. The function reads the trip, the membership, and the block lists, applies the join checks, and commits the membership and the counter in one transaction before it returns a result or a failure code. 13
2.4 Core entities and their identifiers. A membership document joins a user to a trip, both chat kinds hold messages in a subcollection, and a report points back to a message. A user carries an account role and a map of network roles, and a trip carries the network and criteria that scope an event. 15
2.5 The three-step trip creation wizard; step three is shown across two scrolled screenshots. 16
2.6 Match hub results with the score, the contributing reasons, and the broadening label. 16

3.1 The broadening ladder. The search stops once at least five results sit at or above the 35 percent score floor; results below the floor are still returned, behind the recommended ones. 24
3.2 The Piste flow. Planning and building run in the top row, review and delivery in the bottom. Each shaded gate is a point where the workflow stops and waits for me, and the reviewer and verifier never edit the code they examine. 26

C.1 Trip details screen for a trip the viewer has not joined, with the join action and the member count. 40
C.2 The unified chat screen with trip rooms and direct messages. 41
C.3 The profile screen with the gear and riding preferences. 41

List of Tables

1.1 Existing solutions and how they connect riders. 6
1.2 Functional requirements summary. 7

2.1 Firestore collections behind the application. 14
2.2 Cloud Functions and their triggers. 18

3.1 Matching dimensions and their weights. The network weight is added outside the denominator. 23
3.2 Broadening passes in order. Trip filters are the age range and the gender list. 24
3.3 Matching iterations during the internship. Dates are merge dates. 25
3.4 Piste components. Both subagents are read-only, and each declares its own command allowlist. 26
3.5 What each area was verified against. 30

List of Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
APK	Android Package Kit
CI	Continuous Integration
EAS	Expo Application Services
HTTP	Hypertext Transfer Protocol
i18n	Internationalization
JSON	JavaScript Object Notation
OAuth	Open Authorization
PR	Pull Request
QA	Quality Assurance
SDK	Software Development Kit

General Introduction

Planning a ski trip usually starts in a group chat. One person proposes a resort and a week, the group argues about dates, and the details scatter across messages: who is driving, who needs rental gear, who is still deciding. The same pattern repeats every season, and the tools available today help very little. General social platforms can host a group but know nothing about resorts, skill levels, or the constraints that make a trip work. Forums and booking sites each solve one piece of the problem, and none of them compares a rider's profile against the requirements of a trip.

FrostLink addresses that problem directly. It is a cross-platform mobile application for skiers and snowboarders, built around trip creation, weighted group matching, and community chat. A user creates a profile with riding preferences, languages, budget, and skill level, and the matching engine scores every open trip against those attributes and explains the result. Trips have dedicated chat rooms, visibility settings, and push notifications, while opt-in networks let clubs and groups organize privately. The application runs on iOS and Android from a single React Native codebase, with Firebase providing authentication, the Firestore database, Cloud Functions, and push delivery.

I joined FrostLink as a software engineering intern from July 1 to August 31, 2026, and worked across the stack. On the backend I rebuilt the matching pipeline, hardened the Firestore security rules, and worked on account deletion and moderation. On the frontend I worked on trip creation, the Match hub, chat, onboarding, and notifications. I also designed and built Piste, an agentic development workflow that I used to plan, review, and describe pull requests. By mid-September 48 pull requests had passed through it, 41 of them during the internship window. Several beta releases shipped over the two months, from version 1.1.2 in early August to 1.1.4 at the end, with the 1.2.x line following in early September.

The report is organized into three chapters. Chapter 1 defines the project: the context, the host organization, the market, the requirements, and the development methodology. Chapter 2 covers system design and implementation. Chapter 3 examines three areas in depth: the matching algorithm, the agentic workflow built around Piste, and the distance between how production behaved and how it looked on paper.

Chapter 1

Project Definition and Management

1.1 Introduction

This chapter sets the project up: the environment FrostLink was built in, the company that develops it, the domain the application addresses, the concrete problems the first release had to solve, and the way the team organized the two months.

The internship described in this report covered two months of the product's development, from July 1 to August 31, 2026. A part-time continuation followed in September, which is why the Piste statistics and the 1.2.x releases discussed in Chapter 3 extend past the internship window. Chapter 2 covers the architecture, the data model, and the implementation, and Chapter 3 examines the matching algorithm, the workflow I built around Piste, and the production problems the design alone did not predict.

1.2 General Context

Skiing and snowboarding are social activities. Most people do not travel to a resort alone; they go with friends, with family, or with a group assembled for the occasion. Booking the trip itself is rarely the difficult part, since resorts publish their dates, rental shops and ski schools have websites, and lift passes can be bought online. The difficult part is the coordination around the trip, and most of that coordination has no dedicated tool.

In practice, ski trips are planned in group chats. Someone suggests a resort and a week, and the rest of the conversation happens across dozens of messages: who is available on which dates, who owns gear and who needs to rent, who is willing to drive, what everyone can afford, and whether the group is large enough to split an apartment. Details that belong together end up spread across different conversations, and anyone who joins late has to reconstruct the state of the plan from the message history.

Groups on large social networks and Internet forums cover parts of this process. Social network groups are good at gathering people around a sport, and forums are good at keeping long-lived discussions. Neither expresses the structure of a trip: dates, resort, skill level, budget, gear, and the number of places left. A post can describe these things in free text, but no system can compare them against the preferences of a reader or tell a skier which announced trips would suit them. Organizers face the same problem from the other side, and keeping the plan current falls on one person.

Dedicated applications have appeared in the same space, and Section 1.7 compares them with FrostLink in detail. The first release targets riders in Europe, where the company is based and where ski culture is long established.

Seasonality makes the problem sharper. Resorts sell accommodation and passes for specific weeks, prices rise as those weeks approach, and a group that decides late may find little available. The planning window is short and the cost of a late decision is high.

1.3 Host Organization

FrostLink is developed and operated by Tanit Consulting AB, a company based in the Stockholm area of Sweden. The application connects skiers and snowboarders through trip planning, preference profiles, and community features.

One codebase, written in React Native with Expo and TypeScript, ships to both stores. Firebase provides authentication, the Firestore database, and Cloud Functions, and notifications leave through the Expo push service. The interface ships in five languages: English, French, German, Italian, and Swedish. The application covers trip creation, group matching, chat, and notifications, and the minimum registration age is 17.

Profiles carry riding preferences, gear, skill level, budget, and languages, and the matching engine uses them to score trips for the Match hub. A trip is public, private, or an admin-created event, and chat covers trip rooms and direct conversations between members.



Figure 1.1: The FrostLink application icon.

The team I joined was small. Issam Ben Taieb supervised my internship, Abdennour and Mariam worked with me as the other developers, and the three of us reviewed and tested each other's pull requests. The repository was hosted on GitHub, and bugs and tasks were tracked in Nuclino and Trello. Everyone touched several parts of the stack. During the two months of the internship I worked on the matching backend, the Firestore security rules, account deletion, trip creation, the Match hub, the unified chat, onboarding, and notifications, and I designed and built Piste, the agentic development workflow described in Chapter 3.

The stack is serverless as well: Firebase removes the need to operate servers, which lets a small team ship releases every few weeks.

1.4 Social Trip Matching and Community Platforms

Matching people to ski trips is a distinct problem because the attributes that decide whether a trip works are structured and specific to the sport. The most important of these is skill level. A group of beginners and a group of advanced skiers can share a resort and a set of dates and still have incompatible days, because they use different slopes and lifts. Skill also interacts with gear: a beginner is more likely to need rental equipment, while an experienced rider travels with their own skis or board.

Gear ownership is a matching attribute of its own. A trip that expects every member to bring equipment is a poor fit for someone who rents. Budget belongs in the same category: lift passes, accommodation, transport, and rental add up to a total that varies widely, and a mismatch is usually discovered late, after the destination has been agreed.

Resort preference and distance add a geographic dimension. Skiers in the same city may prefer different areas, and a resort two hours away differs from one that requires a flight. Dates are constrained by the season, work schedules, and snow conditions, and the useful window for a given trip can be a single week. Group size matters as well, since apartments, cars, and lessons are priced per group, and every trip carries a capacity that stops being a detail when the group is one short or one over. Language adds a practical constraint in an international sport: two riders can share a resort and still not share a comfortable way to plan an evening.

Beyond these hard attributes, compatibility includes riding style and personality: groomed runs versus park or off-piste, and how a group handles decisions that the plan did not cover. Recurring groups and clubs solve part of the trust problem with a stable membership and a shared history, which is why an application that supports opt-in networks can represent the social context of a trip. A short profile has to carry enough of these signals for a stranger to judge whether the group is likely to work.

This is the structure the matching engine was designed around. Every attribute above becomes a field in the user profile or in the trip record, and the score compares the two sides. General community platforms keep the same information as free text, and a text field cannot be compared across users.

1.5 Problem Statement

The previous sections describe a sport with structured planning needs and general tools that do not model them. FrostLink was built around four problems.

The first problem is fragmented communication. Group chats mix ordinary conversation with the information that matters, so dates, costs, equipment, and commitments are spread through a scrollback history, with no single record of what the group has agreed. When a member leaves or a trip fills up, the change is a message like any other, and the people who miss it plan around outdated details.

Centralized trip discovery is the second problem. Trips are announced in whatever group their organizer happens to belong to, and there is no catalog that a skier can browse. Someone who wants to join a trip in February sees only the announcements that reach the groups they already belong to. Discovery depends on being in the right place at the right time.

Compatibility is the third problem. A person who considers joining a trip sees a short description and a list of participants, and has no reliable way to judge whether the group matches their skill level, budget, language, or expectations. An organizer receiving a join request faces the same uncertainty in the other direction. Mismatches surface during the trip, when changing plans is expensive, so trust between strangers is a practical concern for the product.

Notifications close the list. Trip details change, seats fill, and messages arrive at the moments when decisions are made, and a missed notification can cost a member a place or send them to the wrong resort with the wrong equipment. The failures are also platform-specific: an issue that breaks delivery on Android can leave iOS unaffected, and

the reverse. Testing every flow on both platforms, which the team did repeatedly during the internship, was how these gaps were found.

Section 1.6 turns these problems into the objectives and scope of the first release.

1.6 Objectives and Scope

The first release of FrostLink was defined around six objectives. Trip creation came first: a member describes a resort, a date window, a member limit, and optional age and gender filters, and that record becomes the unit the rest of the application attaches to. Matching was the second objective: every open trip is scored by comparing the rider's profile against the trip's current members, and the result is shown with an explanation. Chat and notifications formed the third and fourth objectives, so that planning a trip and changing its details stay inside the application, and profiles and opt-in networks completed the set by giving matching something to compare and clubs a way to organize privately.

The objectives depend on each other. Matching has nothing to compare without profiles, discovery needs trips to list, and chat needs a trip before it has a room. This is why the matching work came after the first stability pass: the attributes it consumes had to be defined and populated first.

The scope was limited on purpose. Subscriptions and private trip invitations are out of scope for this report. The matching engine used static weights instead of learning from behavior. Static weights were a deliberate choice for a first version, and the reasons for that choice appear in Chapter 3.

The market scope was Europe. The company is based in Sweden, the interface language set covers the main Alpine and Nordic markets, and the distances that the matching engine broadens over, from 50 to 400 kilometres, describe a continent where resorts are reachable by car or a short flight. No market statistic is claimed here; the first release targets Europe, following the geography of the sport.

The two months of the internship produced three releases inside the window, listed in Section 1.10, and the 1.2.x line followed in early September. By the end of the internship every objective above was in the application.

1.7 Market Study and Existing Solutions

People who want to join a ski trip today choose between general platforms and a small set of dedicated applications. The general platforms are where most coordination happens. Meetup [8] organizes interest groups and events, Facebook groups host communities around a resort, a region, or a discipline, and forums keep long-running discussions about conditions, gear, and resorts. Booking sites complete the picture by selling the trip itself, with no social layer at all. Each of these tools works for what it was built for, and none represents a trip as a record whose dates, capacity, and group profile can be compared.

Dedicated applications address parts of the gap. SkiBuds [13] connects riders for après-ski, carpooling, and days on the mountain, and adds conditions and lift status. Broski [1] matches riders by availability, resort, and riding style. buddies.ski [2] places travelers in resort chat groups and matches them by dates. Snoji [14] combines group posts for carpools and lodging with a secondhand gear market and course listings. Table 1.1 summarizes the comparison.

Other names in the space include Snowflake [15], an early location-based buddy application, SkiBuddy [12], which connects skiers, snowboarders, hikers, and bikers, and Skibuddies [11], a community network for LGBTQIA+ riders. A pattern is visible across all of them: most products connect riders for a day on the mountain or gather an audience around a resort. Fewer treat a multi-day trip with a cost, a capacity, and a group whose skill and budget can be compared as the object to be matched.

FrostLink was built around that object. A trip is a record with a resort, a date window, a capacity, and filters; matching compares a profile against the members of each trip across several weighted dimensions and explains the score, and the result is a ranked list with reasons attached. Opt-in networks add the group context that recurring clubs already provide elsewhere, and the five-language interface targets the European market described in Section 1.6. Table 1.1 reflects public product descriptions at the time of writing, in September 2026; features in this market change quickly.

Table 1.1: Existing solutions and how they connect riders.

Platform	Focus	How it connects people
Meetup	General interest events	Group membership and event sign-ups
Facebook groups	Community discussion	Posts and comments
Ski forums	Long-lived discussion	Threads and replies
SkiBuds	Ride partners, carpool, conditions	Community posts by location
Broski	Same-day riding partners	Availability, resort, and riding style filters
buddies.ski	Resort groups and travel companions	Resort chat groups and date matching
Snoji	Trips, lodging, and gear	Group posts with carpool and lodging
FrostLink	Trips and groups	Weighted profile-to-trip scoring with explanations

1.8 Requirements Analysis

FrostLink has one base role and three privileged roles layered on it. A member is any registered user: they create trips, match and join public trips, join events, chat, and manage their profile. A member who creates a trip becomes its trip admin and manages that trip's details, status, and members; trips carry their own admin list instead of inheriting authority from a global role. A network admin holds the admin role inside one network and creates events scoped to it, while a superadmin creates application-wide events, manages application-level settings including the moderation switches described in Chapter 2, and reads the moderation inbox. The separation matters in the security rules, where no role grants authority beyond its scope.

Table 1.2 summarizes the functional requirements. They came from the problems in Section 1.5 and from testing the application against real use: the bug bash in the first two

weeks of the internship turned most of them into concrete cases, and the list was refined as the data model changed.

Table 1.2: Functional requirements summary.

Area	Requirement
Trips	Create, edit, and cancel a trip with a resort, a date window, a capacity, and optional age and gender filters; public, private, or event type
Events	Network admins create events scoped to a network they administer; superadmins create application-wide events; visibility follows network membership and subgroup criteria
Membership	Join and leave through server-side functions, with capacity and overlap checks
Matching	Score open trips against a profile across weighted dimensions, explain the result, and broaden when results are thin
Chat	A room per trip and direct conversations, with reactions and message reporting
Notifications	Push delivery for new messages and trip changes, with suppression while the relevant screen is open
Profiles	Riding preferences, gear, skill level, budget, and languages
Networks	Opt-in groups joined by invite code; membership gates network events
Moderation	Message reports, under-18 flagging, and per-user rate limiting
Administration	Trip admins manage their members; network admins manage their network and its events; superadmins manage application settings and reports
Localization	Interface available in English, French, German, Italian, and Swedish

The non-functional requirements are summarized by five constraints. The application is cross-platform mobile: one codebase serves iOS and Android, and every flow is tested on both. Data is real-time, with Firestore listeners keeping chats, rosters, and trip details current. Offline behavior is handled through cached data, and it was tested explicitly during the internship. Security is enforced twice, in the Firestore rules and in Cloud Functions for the operations that rules cannot express, so a client cannot bypass the checks; Chapter 3 records the one join check that the shipped version does not repeat. Performance work is described through request patterns rather than read counts. The N+1 reads removed from the Your Trips screen and the parallelized Events feed queries are the clearest examples.

1.9 Development Methodology

The team followed an agile approach [16] within the Scrum framework [18]. Work was organized in sprints, standups ran over Meet or Slack, and planning and grooming sessions marked phase boundaries. The team did not designate formal Scrum roles, and the three developers planned and prioritized together. The internship opened with a bug bash,

which filled the first two-week sprint and closed with a demo and a retrospective. A ticket in Trello was the entry point for almost every piece of work, and failures found in testing went to a shared Nuclino bug list.

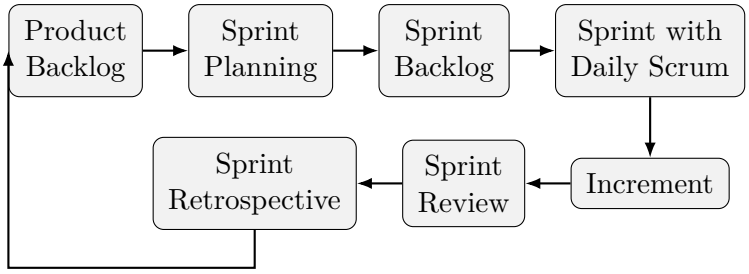


Figure 1.2: The Scrum framework: the Product Backlog feeds Sprint Planning, the Sprint produces an Increment, and the Sprint Review and Retrospective close the loop.

Feature work started with a written plan, and code was written only after the plan was agreed. The same discipline runs through Piste: its `/work` command produces a plan and stops, and it never edits files. Planning before coding was not a formality. The schema changed often enough that a plan was the cheapest place to discover that a feature touched a collection whose shape was about to change.

Pull request review was the team standard throughout. Every change reached main through a pull request that at least one other developer reviewed and tested on a device, and I reviewed and tested the pull requests from Abdennour and Mariam as part of my normal workload.

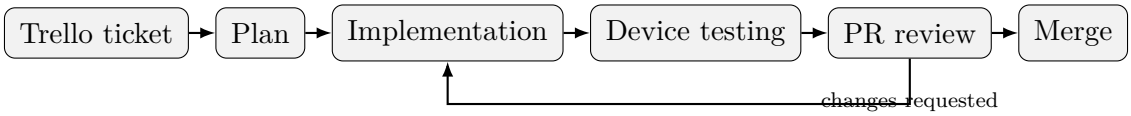


Figure 1.3: The development workflow: every change moved from a ticket through a plan, implementation, device testing, and review before it reached main.

Feature work and review work were interleaved, which is how matching v2, trip types, networks, moderation, and onboarding all shipped inside two months. Testing happened on real devices early and often, because emulators and the web preview did not reproduce notification and map behavior. The project allowed and encouraged AI tools. Piste, described in Section 3.3, was my own use of that permission.

1.10 Project Planning and Timeline

The internship ran for nine weeks of active work, from July 1 to August 31, with a part-time continuation into early September that produced the 1.2 releases. Figure 1.4 shows the phases across that period. The bug bash occupied the first two weeks. Matching v2 and the trip types rework followed, with the trip types change touching the data model, the Cloud Functions, and the Match hub at once. The scoring refinements ran through August, from the score floor and reasons to the opt-in network dimension and the null-score fix. Security rules and the emulator tooling ran in parallel with chat, networks, and moderation, and continued into the final weeks because the schema kept moving. The

last phase covered onboarding, trip visibility, and performance work, and the 1.2.x line carried the remaining fixes into September.

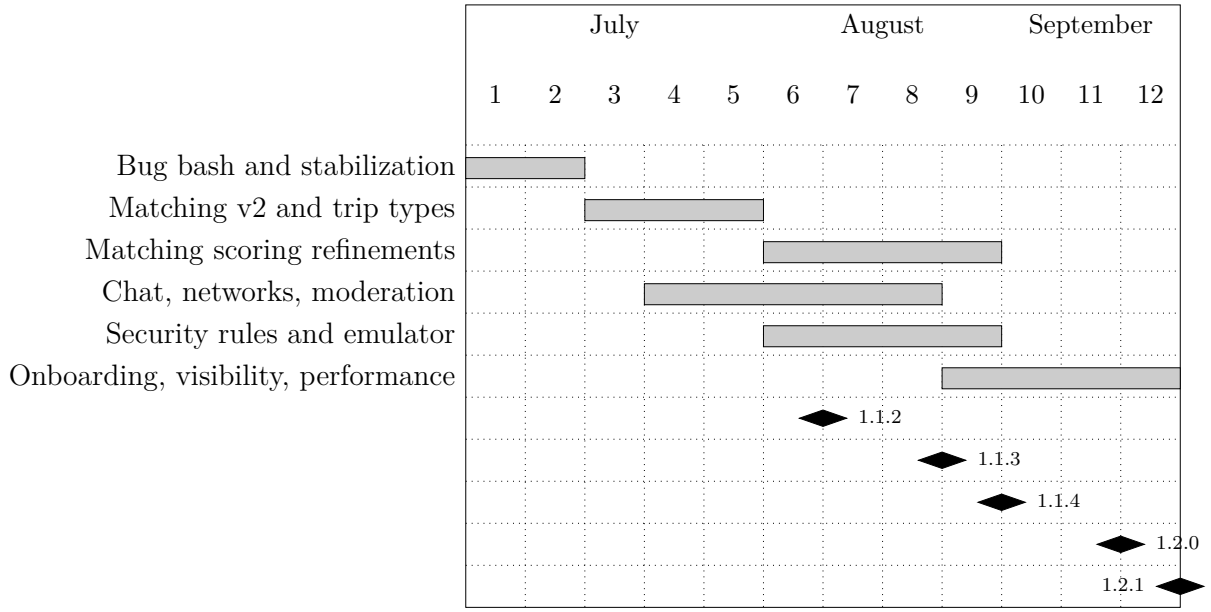


Figure 1.4: Project phases and releases. Each column is one calendar week; column 1 is the week of June 29, 2026, and column 12 is the week of September 14, 2026.

Two properties of the plan are worth stating. The work streams overlapped by design: the rules, for example, could not be finished before the data model stopped changing, so the rules work was scheduled to run alongside the features that changed it. And the estimates were not always right. The security rules took three weeks instead of one, and Chapter 3 explains what that cost and what the test suite saved.

Releases marked the phases. Version 1.1.2 shipped on August 6 with event creation and the first matching changes, 1.1.3 on August 21 and 1.1.4 on August 29 with the accumulated fixes from the chat, rules, and platform work, and 1.2.0 on September 9 with the Expo SDK 57 upgrade and the iOS TestFlight fixes. Version 1.2.1 followed on September 14.

1.11 Conclusion

This chapter covered the ground the rest of the report builds on, from the context of the sport to the requirements, the method, and the timeline.

Two themes carry forward. The product adds structure where the alternatives keep free text: trips as records, profiles as comparable attributes, scores with explanations, and rules that enforce the same constraints on the server. And the plan and the execution diverged in places that only production revealed, from a schema that kept moving under the security rules to platform differences that only device testing exposed.

Chapter 2

System Design and Implementation

2.1 Introduction

The system is best understood from the outside in: architecture and stack choices first, then the roles, the data model, and the interface, and finally the backend, the frontend, and the release pipeline. The matching endpoint is taken apart in a section of its own later.

2.2 System Architecture

FrostLink runs on two sides that Firebase joins together. The client is a React Native application written in TypeScript and built with Expo SDK 57, and it ships as one codebase to iOS and Android [10]. Screens and components carry the interface, hooks hold screen logic, contexts share state such as connectivity and block lists, and React Navigation keeps a root stack for detail screens behind a bottom tab bar. The five interface languages run through i18next. The backend is Firebase [6]: authentication through the email, Google, and Apple providers; Cloud Firestore for the documents and the real-time listeners; and Cloud Functions for the server-side logic on Node 22. Notifications are handed to the Expo push service, which reaches Android through Firebase Cloud Messaging and iOS through the Apple push service. Three further services complete the picture. Google Places answers location searches, Google Maps draws the resort map, and a hosted moderation API reviews message text when the pipeline calls it.

The design is server-authoritative. Clients never create trips, memberships, or messages directly; those writes arrive through Cloud Functions, which verify the caller, apply the business rules, and commit the result in Firestore transactions and batches. Trip admins edit their trip documents from the client, under rules that admit no one else. This follows from the problems in Section 1.5: capacity, overlap, and block checks have to hold for every client version, so they live on the server and not in the screen that happens to submit the request. Figure 2.1 shows the layers and the traffic between them.

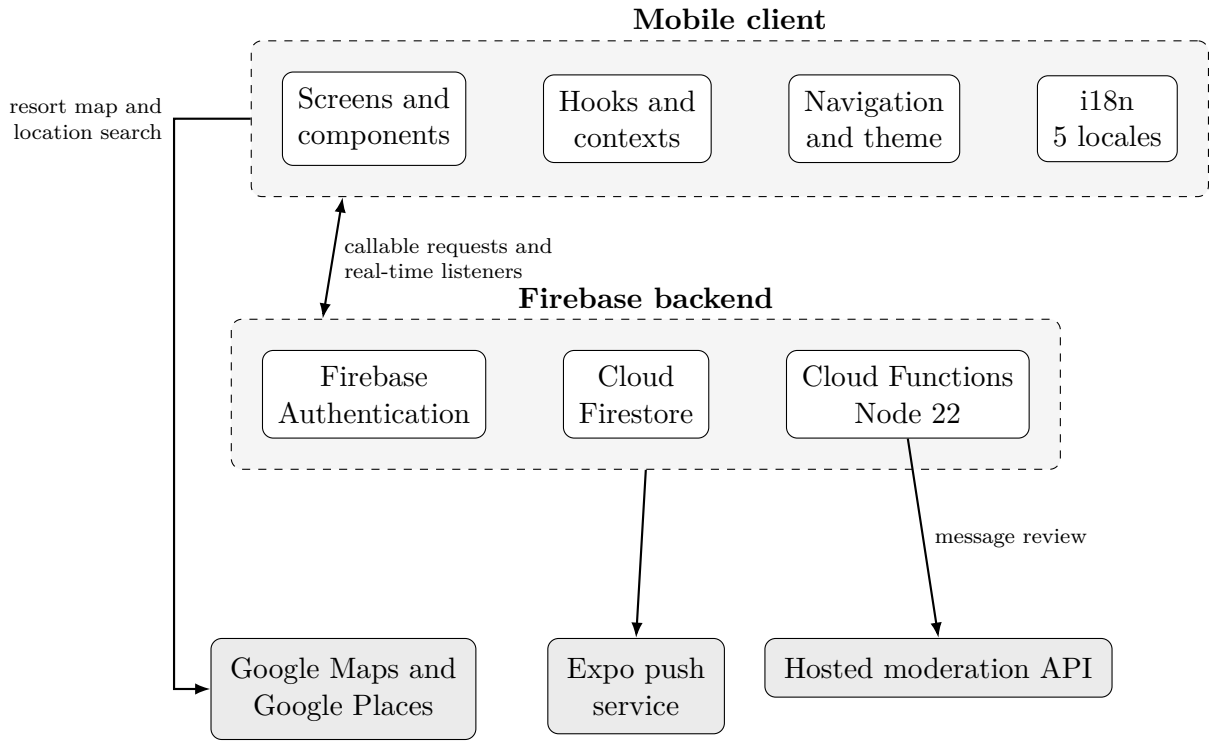


Figure 2.1: The system architecture. The client reads and writes through callables and listeners, calls Google Maps and Places directly, and the functions call the hosted moderation API and hand notifications to the Expo push service.

2.3 Technology Justification

The stack was chosen for a small team shipping a mobile product over two months. React Native gives one codebase for both stores, and the application needs native capabilities in several places: maps, push notifications, date and time pickers, haptics, and location. Writing the product twice was not an option for a team of three developers.

Expo SDK 57 provides the build layer [5]. EAS Build compiles the application in the cloud, so no one needs a Mac or a configured Android toolchain to produce a test build, and the development profile installs a development client that loads the JavaScript bundle from the developer machine. The cost is a dependency on the Expo release cadence and a native project that is regenerated rather than maintained by hand. TypeScript runs on both sides of the network. The client and the functions share type definitions for trips, memberships, and messages, so a change to the trip document shape breaks the build in both places instead of failing at runtime.

Firebase suited the product for four reasons. It provides the three sign-in providers the team wanted, a document database whose real-time listeners keep open screens current, security rules that can be tested against an emulator, and functions that remove the need to operate a server. The application is coupled to the platform, and leaving it would mean rewriting the data layer. Firestore has no joins, so reads are shaped around denormalized documents, which Section 2.5 describes, and queries are limited to combinations that have an index. The security rules grew into a layer of their own, described in Section 2.7.2. The team accepted those costs in exchange for not running infrastructure; Chapter 3 reports where they showed up.

2.4 System Design

The roles were defined in Section 1.8: a member, plus the trip admin, network admin, and superadmin rights a member can hold. Whoever creates a trip is its trip admin; a network admin creates events scoped to one network; a superadmin owns application-wide events, settings, and the moderation queue. Figure 2.2 attaches the actions to each role.

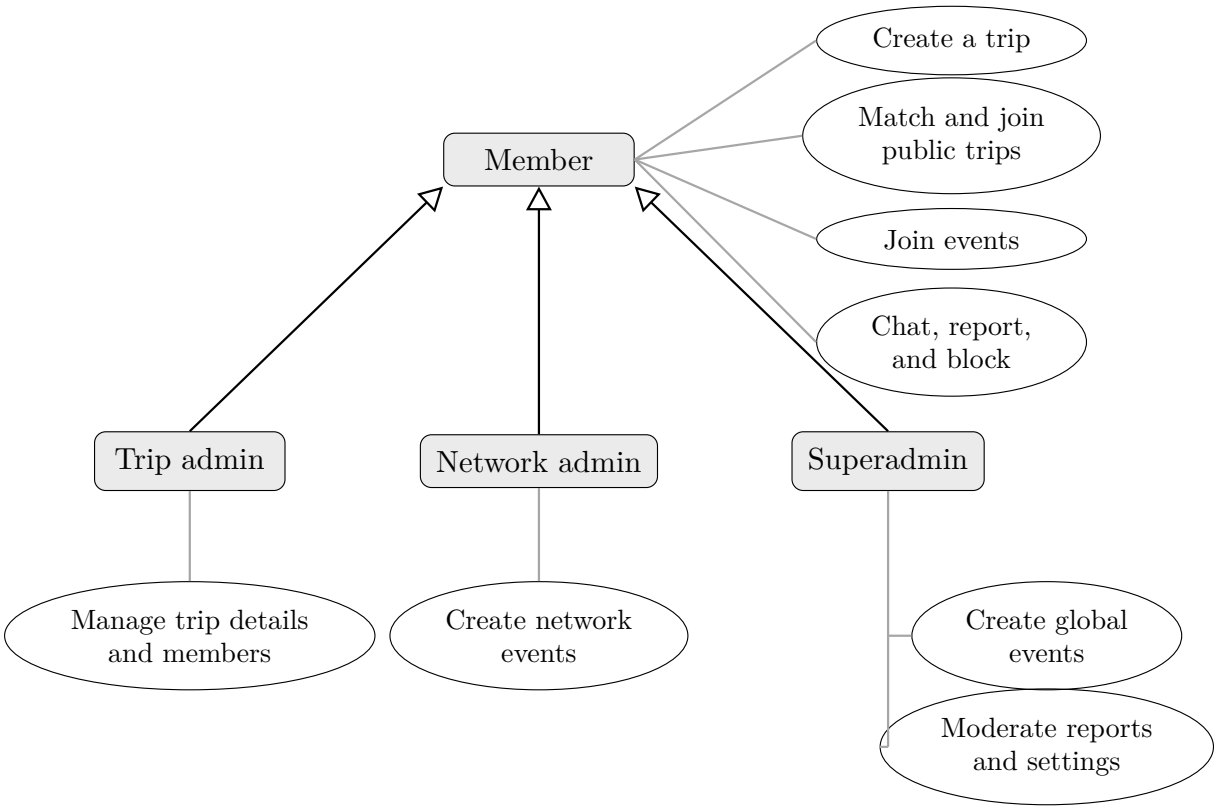


Figure 2.2: Roles and the main actions attached to them. The generalization arrows show that trip admins, network admins, and the superadmin are members with additional rights, and a member who creates a trip becomes its trip admin.

Flows that change shared state run through Cloud Functions, which re-check the caller and the current state before writing. The join flow in Figure 2.3 is the clearest example. The client sends the trip identifier to the `joinTrip` function; the function reads the trip, the membership, and the block lists, then applies the checks: an admin cannot join their own trip, a full trip rejects the request, a duplicate join is refused, a block on either side stops it, and a confirmed date overlap blocks the join while a provisional overlap only warns. The membership and the counter are committed in one transaction, and a full public trip flips to private in the same write. Self-serve joins are limited to public trips and events. The client receives codes such as `trip_full` and `trip_overlap`, each mapped to a message.

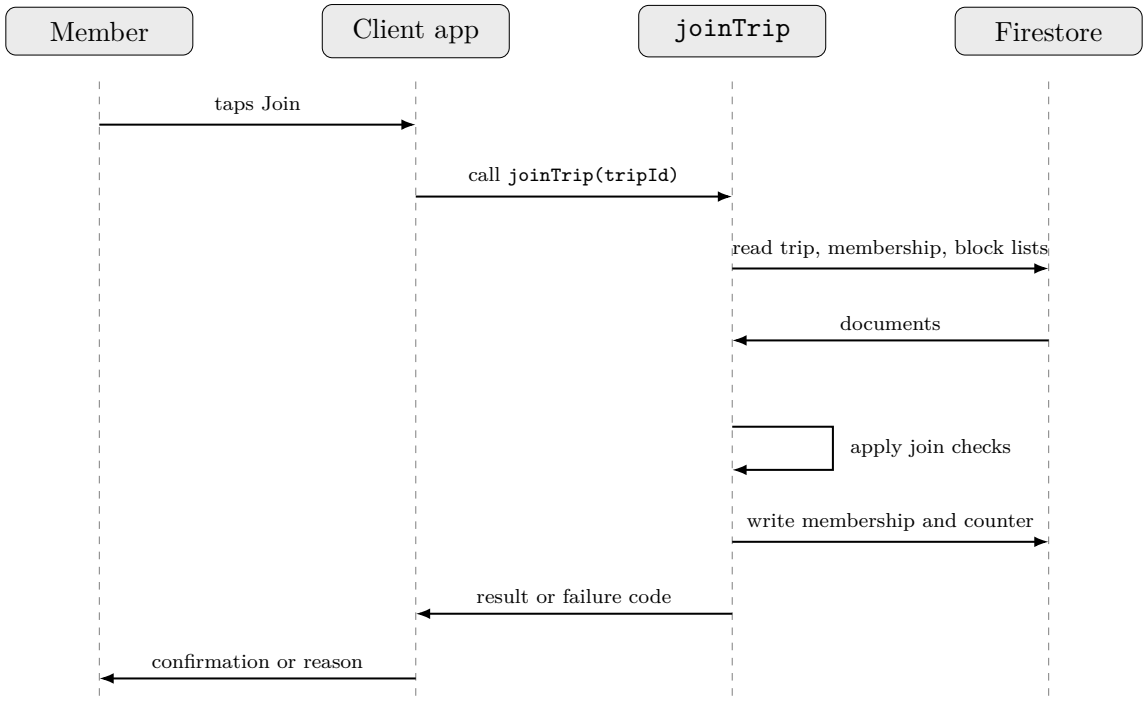


Figure 2.3: The join flow for a public trip or an event. The function reads the trip, the membership, and the block lists, applies the join checks, and commits the membership and the counter in one transaction before it returns a result or a failure code.

Trip creation and message sending follow the same shape. A callable validates a small payload and writes the related documents in one batch or transaction. Event creation carries the only creation gate: a network admin scopes the event to a network they administer, and a superadmin may omit the network, which makes the event application-wide. Matching is the exception in transport only. It is served by an HTTP endpoint authenticated with the caller's identity token because it returns a ranked list rather than a state change, and it scores public trips only; events reach members through the Events feed. The internals of the score are the subject of Chapter 3.

2.5 Database Design

Firestore stores documents in collections, and a document can hold subcollections [3]. The trips collection holds the trip record, and tripMembers joins users to trips: the document identifier is the composite string `{tripId}_{uid}`, so a membership is fetched by identifier and an existence check needs no query. Messages are subcollections of their chat parent: `tripsChat/{tripId}/messages` for a trip and `privateChats/{chatId}/messages` for a direct conversation, whose identifier is the two user identifiers in sorted order. Both chat kinds share the same message shape, which lets one screen and one send function serve them. Table 2.1 lists the collections, their identifiers, and their contents. Networks are the exception: the catalog is static JSON shared by the client and the functions, and memberships are a map on the profile document.

Table 2.1: Firestore collections behind the application.

Collection	Document ID	Contents
`users`	`uid`	Profile, preferences, languages, account and network roles, push tokens, moderation and deletion fields
`users` subcollections `blocked`, `blockedBy`	User identifier	Mirrored block records, one pair per block
`trips`	Automatic	Type, status, dates, location, capacity, counters, admin list, network and criteria for events
`tripMembers`	`{tripId}_{uid}`	Role, status, join time
`tripsChat`	Trip identifier	Chat parent, last message preview, typing state, admin list
`tripsChat` subcollections `messages`, `userStatus`	Automatic, user identifier	Messages with reactions and read state; last read time per user
`privateChats`	Sorted uid pair	Direct chat parent, participants, last message preview
`privateChats` subcollections `messages`, `userStatus`	Automatic, user identifier	Same message shape as trip chats
`messageReports`	Automatic	Report, moderation status and labels, message text snapshot
`moderationActions`	Automatic	Audit record for each moderator action
`appSettings`	`reports`	Moderation toggles
`moderateRateLimits`, `reportRateLimits`	User identifier	Sliding windows for the limits, removed after expiry

Firestore has no joins, so the model duplicates data where a read would otherwise fan out. The trip document carries the counters and the admin list, each chat parent carries the last message preview, a reply stores the quoted text and its sender, and a report copies the message text and the moderation outcome so it survives deletion. Each copy is written by the function that writes the source, in the same batch or transaction. A repair script rebuilt the counters and removed memberships whose user document no longer existed.

Indexes are declared in `firestore.indexes.json` rather than in the console. Four composite indexes carry the main queries: the events feed orders trips by type, environment, and creation time; the notification fan-out reads trip members by status; the moderation fan-out finds reports by message and chat path; and the suspension sweep reads users by state and expiry. Schema changes ran through versioned scripts: v2 for network and visibility fields, v3 for the matching criteria, v4 for the trip type taxonomy, v5 for preference normalization, and a repair script for the member counters. Figure 2.4 shows the core entities and the fields that are duplicated on purpose.

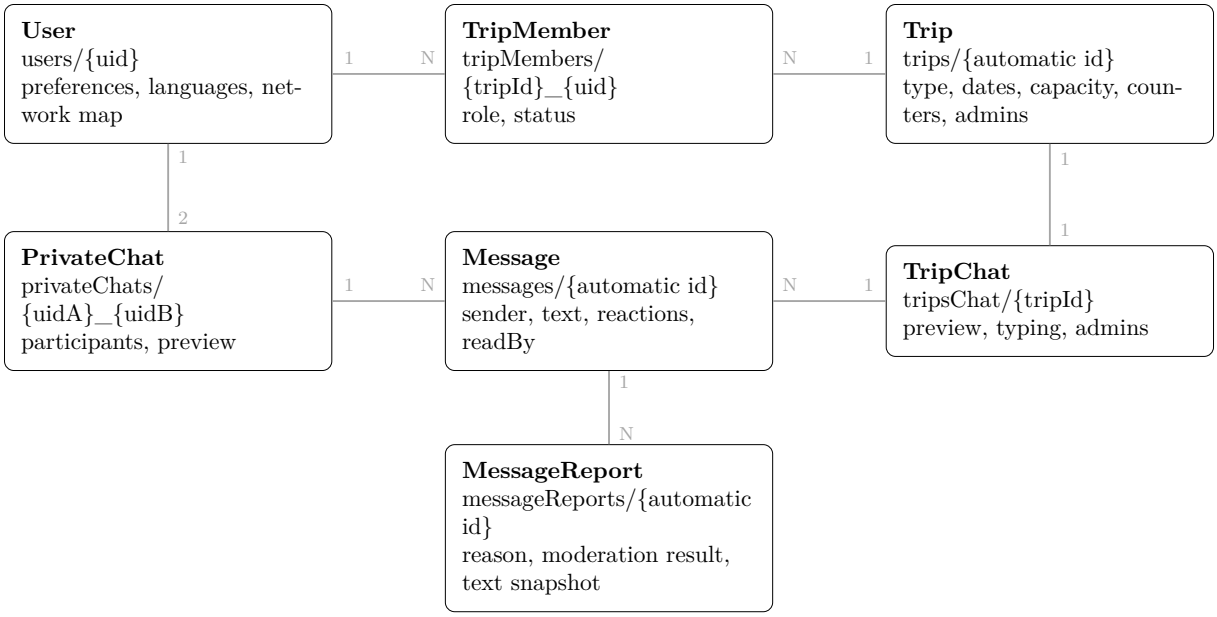


Figure 2.4: Core entities and their identifiers. A membership document joins a user to a trip, both chat kinds hold messages in a subcollection, and a report points back to a message. A user carries an account role and a map of network roles, and a trip carries the network and criteria that scope an event.

2.6 User Interface Design

The application opens on a landing screen for signed-out visitors, then moves through registration, email verification, and three onboarding steps: the profile, the sports profile with gear and riding preferences, and the notification permission. After onboarding, the user reaches five tabs: Events, Match, Your Trips, Messages, and Profile. Detail screens such as trip creation, trip details, and chat push over the tab bar from a root stack, which keeps the tab state alive through a flow. The Events tab lists the events the user can see: network-scoped events, application-wide events, and events whose subgroup criteria their profile satisfies.

The trip creation wizard (Figure 2.5) spreads a long form over three steps. Step one sets the name, the description, and whether the trip is committed or provisional. Step two sets the dates and the location: a fixed range and a resort for a committed trip, or an earliest and a latest date with a resort, a region, or a broader area for a provisional one. Step three sets visibility, the member limit, accommodation, transport, and the age and gender filters. Each step validates itself and the wizard refuses to advance until it passes, so a date error appears on the date field instead of at the end of a long form.

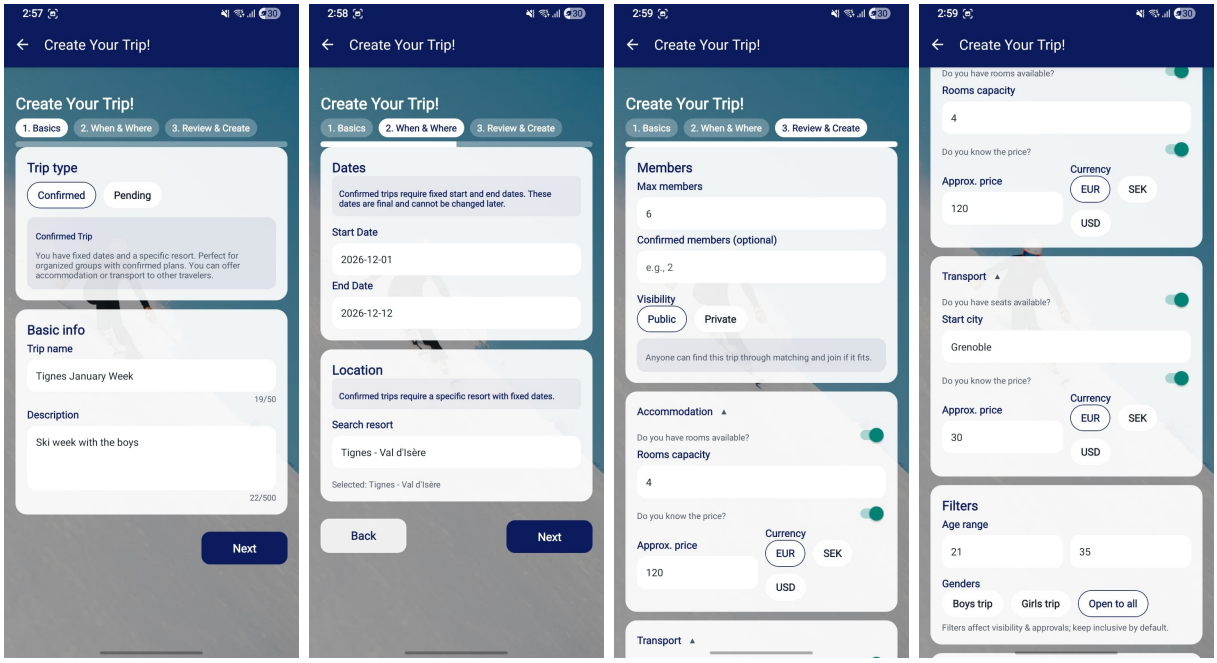


Figure 2.5: The three-step trip creation wizard; step three is shown across two scrolled screenshots.

The Match tab (Figure 2.6) also runs in three stages: preferences, search, and results. Each result card carries the score, up to three reasons from its strongest dimensions, and a label when the engine broadened the search. The reasons matter for the problem stated in Section 1.5: a user can see why a trip was recommended instead of trusting an unexplained number.

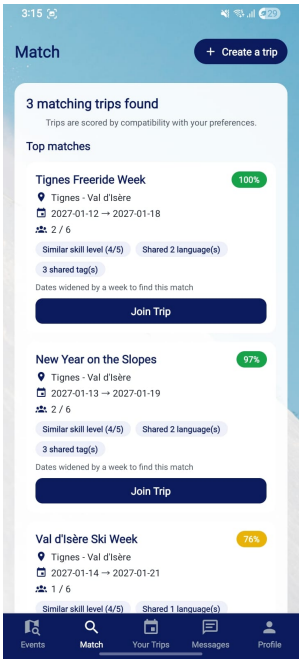


Figure 2.6: Match hub results with the score, the contributing reasons, and the broadening label.

Appendix C collects three more screens: trip details, the unified chat, and the profile.

2.7 Backend Implementation

The backend is where the product's rules live. Cloud Functions runs on Node 22, and the functions divide into four families: callables for user actions, event triggers for work that follows a document, one scheduled job, and one HTTP endpoint for matching. A callable is an HTTPS function that the Firebase client SDK invokes with the caller's identity attached, so the function can trust the user identifier without reading it from the request body. Table 2.2 lists the functions, and the subsections below cover the patterns they share.

2.7.1 Cloud Functions

The functions reuse a small set of patterns. Callables validate their input and return business failures as data with codes, so a client that tries to join a full trip receives `trip_full`, which the screen turns into a message. The trip functions are transactional where two writers could collide: `joinTrip` re-reads the trip inside the transaction before it writes the membership and the counter, and `leaveTrip` wraps the same transaction in a retry loop so a double leave decrements once. Event triggers handle work that follows a document, which is how message and reaction notifications fan out and how a report reaches the moderator. One scheduled job runs every twenty-four hours and re-enables accounts whose suspension has expired.

`createTrip` sets the trust-sensitive fields itself. The creator becomes the only admin, the counters start at one, the environment is copied from the profile, and the schema version is stamped. It writes the trip, the chat parent, and the creator's membership in one batch. `sendMessage` is the only path that creates messages. It checks that the caller is a member of the trip chat or a participant in the direct chat, rejects messages between blocked users, caps the text at four thousand characters, and writes the message and the chat preview in one transaction. Network membership runs through three callables against the static catalog. Account deletion tombstones the profile before it cleans up memberships, chats, and the authentication record, so a failed cleanup leaves a visible record.

Table 2.2: Cloud Functions and their triggers.

Function	Trigger	Purpose
`createTrip`	Callable	Validates the payload and batch-writes the trip, the chat parent, and the creator's membership; event creation is admin-only
`joinTrip`	Callable	Accepts public and event trips, applies the capacity, overlap, and block checks, and commits the membership and the counter in one transaction
`leaveTrip`	Callable	Removes the membership, promotes an admin when needed, and deletes an empty trip with its chat
`confirmTripMembership`, `setTripStatus`, `setTripVisibility`	Callable	Moves a trip between provisional and confirmed states and flips public or private visibility
`sendMessage`	Callable	The only message write path; validates the sender and updates the parent preview
`matchTrip`	HTTP request	Verifies the identity token, queries the eligible trips, and scores them for the Match hub
`joinNetworkByInviteCode`, `leaveNetwork`, `kickUserFromNetwork`	Callable	Network membership against the static catalog
`blockUser`, `unblockUser`	Callable	Mirrored block records, capped at 500 per user
`deleteUserAccount`	Callable	Tombstones the profile, cleans up memberships and chats, then deletes the authentication user
`onPrivateMessage`, `onTripMessage`, message reaction triggers	Firestore trigger	Push fan-out for new messages and reactions
`onMessageReported`	Firestore trigger	Runs the moderation model, applies strikes, and suspends at the threshold
`takeModerationAction`	Callable	Superadmin action on a report, with an audit record
`liftExpiredSuspensions`	Scheduled	Re-enables accounts whose thirty-day suspension expired
`onModerationToggle` `Backfill`	Firestore trigger	Re-checks reports created while moderation was off

2.7.2 Firestore Security Rules

The rules file is 387 lines and starts from deny by default [7]. Every collection the client touches has an explicit match block; anything without one, including the rate-limit and audit collections, is reachable only through the Admin SDK. Writes that could forge state are closed to clients: trip creation, memberships, messages, blocks, and application settings are written through functions, and a trip document accepts updates only from its admins. The exceptions are specific: the client creates a report document in a shape the rules verify, a participant may edit only the reactions and read-by fields of a message, a user may update the profile fields that are not trust-sensitive, and a superadmin may flip the moderation toggles in the exact document shape the rules expect.

A trip membership is proven by the existence of a document whose identifier is the composite `{tripId}_{uid}`. Trip admin authority comes from the admin list on the trip document, with a second variant for transactions, where a `get` call would fail. The superadmin check reads the account role from the profile, and the rules protect the fields the moderation pipeline owns, so a client cannot promote itself or reset its strikes.

The rules filter queries instead of trusting the client to filter them: the trip list rule allows only the public and event queries the application issues, and the single-document

rule accepts a missing document so the client sees a not-found result. Deleting a missing document is allowed in several places for double-leave races. The emulator suite runs roughly 150 checks across nine areas. The work behind it ran longer than planned; Chapter 3 covers what that cost.

2.7.3 Push Notifications and Moderation

Push delivery starts with a token, which the client registers after the permission step and stores on the profile. Four Firestore triggers send notifications: two for new messages and two for reactions. Trip recipients are the confirmed and pending members; a mentioned member receives a mention instead. Reaction notifications go only to the sender, for a new or changed reaction.

The payload carries the notification type, the chat or trip identifiers, and the target screen. The client routes a tap to the matching chat, and a cold-start tap is held until navigation is ready. While the user is inside the relevant chat, the foreground handler suppresses the banner. Users on either side of a block or with notifications off are skipped, and unregistered tokens are removed on the next send.

Moderation begins with a report from the chat. The client writes a report document, the rules verify its shape, and a trigger handles it. Each reporter may file ten reports per hour, with limiter documents that expire. The function sends the text to the hosted moderation model over a streaming API, which returns a label set. The two most severe label groups apply a strike, and a third strike disables the account and records a thirty-day suspension. The superadmin console applies a dismissal, a removal, or a user action, and each appends an audit record. The client marks messages from under-18 members with a badge and a banner in trip and direct chats.

2.8 Frontend Implementation

The client holds no state management library. Screens fetch what they need and share state through contexts and hooks, which kept the dependency list short and the data flow readable. As a result, some screens talk to Firestore directly while others go through hooks, and Chapter 3 discusses the reads that this produced.

2.8.1 Core Screens and Features

Chat is the largest piece of shared code. One hook and one presentational screen serve trip chats and direct chats, and the two wrapper screens supply the header, the data source, and the chat-specific behavior. Messages support reactions, replies with a quoted snapshot, a typing indicator, and mentions with autocomplete over the trip roster. Read state has two layers: a per-message read list draws the receipt inside a bubble, and a per-chat last-read timestamp on the chat parent feeds the unread badges in the inbox. The inbox listens to the user's direct chats, the user's memberships, and the parent documents of the trip chats, and it recomputes the badges behind a debounce. When the connection drops, composed messages queue in local storage under their collection path, each carries a client identifier, and the queue flushes on reconnect; the identifier also stops the interface from drawing an optimistic message twice.

Onboarding ends at the notification permission, and the language chosen at any point is stored on the profile and reapplied on the next sign-in, with the device language as

a fallback. The Match tab saves the preference form through a callable, then sends the saved search identifier to the matching endpoint with the identity token in the request header.

2.8.2 Build Pipeline and Releases

Every build is produced by EAS [4]. The development profile installs a development client on a tester device, the preview profile produces an APK that can be sideloaded without the Play Store, the production profile produces an Android App Bundle for the store, and a fourth profile produces a production APK for testers. iOS builds reach TestFlight through EAS Submit. The build environment decides the data lane: the production profiles set the environment flag to production and the other profiles use the QA lane, so a test build cannot write into real trips.

Versions are bumped together in the application configuration, the package manifest, and the changelog, while the native build number for the release profiles is incremented remotely by EAS. A script stamps the branch, the commit, and the version into a generated file the settings screen reads, so a tester can report the exact build. Section 1.10 lists the release dates. From 1.1.2 in early August to 1.2.1 in mid-September the pipeline changed twice: the Android profile was split into a store build and a production APK for testers, and the SDK 57 upgrade rebuilt the native directories and produced the iOS launch crash that 1.2.0 fixed. There is no automated CI. Each release followed a checklist of rules tests, function tests, a device pass on both platforms, a build, and a submission.

2.9 Conclusion

This chapter described the system the first chapter set out to build: the two-tier architecture, the Firestore model with its composite identifiers and deliberate copies, and the rules, notification, and moderation pipelines.

One principle held throughout: the server decides. Chapter 3 examines where that principle was hardest to keep, from the matching algorithm and its iterations to Piste and the production problems that appeared after a release.

Chapter 3

Beyond Code: Engineering Intelligence, Modern Workflows, and Production Reality

3.1 Introduction

Chapter 2 described what the system is. This chapter is about what building it required beyond the design. Three subjects carry it: the matching engine, whose first scoring mistake cost several rounds of rework; Piste, the development workflow I built and used alone; and the production problems that surfaced only on real devices and in the stores. The evaluation closes by stating where the verification stopped.

3.2 Intelligent Trip Matching

3.2.1 The Matching Challenge

Matching compares the profile described in Section 1.4 against the trip record and returns a ranked list that a person can trust. The rider side is a small set of structured attributes, and the trip side is a resort, a date window, a capacity, and the members already signed up. The engine has to combine them into a single percentage that a user can act on.

The dimensions do not share a unit, the data is often incomplete, and the user who disagrees with a score sees a single percentage that has to account for all of it. Producing a ranked list was never the hard part.

The rest of the system narrows the question before the score runs. Matching considers public trips only, and it excludes trips the caller already administers or has joined, trips that are full, trips whose members have blocked or been blocked by the caller, and trips that overlap a confirmed trip the caller joined. Those conditions are enforced at the point of the action, in the functions described in Section 2.4; the score is a recommendation, and the two are kept apart on purpose.

3.2.2 Algorithm Inputs

The score compares two sides. The rider side is a snapshot taken when the search starts, with the live profile as the fallback: the highest gear level, the language list, the interest groups, the budget tier, and the one-to-three values for ride style and personality. The

trip side is read from the members of each candidate trip. A trip does not carry a skill level or a budget of its own; its character comes from the people already in it, which is why a trip with no completed profiles is treated as unknown.

The search request carries the dates and the location. A user searches from a resort, a region, or a wider area, and that point anchors the distance dimension. Distance is the haversine distance [19] between the search point and the trip coordinates, rounded to whole kilometres, and either side can lack coordinates. Section 3.2.3 describes what the score does with that case.

The last input is the optional network flag. When a rider enables it and belongs to at least one network, the score checks whether any current member shares a network. The rider's network identifiers are read from the live profile rather than from the request, so a client cannot claim a membership it does not hold.

3.2.3 Scoring Algorithm Design

The score is a weighted sum [17] over seven dimensions, with an eighth added as an optional bonus. Table 3.1 lists the weights and what each dimension compares. Skill carries the largest weight at 30 because it decides how a group spends its day, language follows at 20 because a shared language decides whether planning works, and the remaining dimensions sit between 12 and 6.

Every dimension produces a value between zero and one before the weight applies. Skill compares the highest gear level on each side: equal levels score 1, a difference of one level scores 0.7, and a wider gap scores 0. Language is binary, since any shared language is enough. Groups and budget measure a share, the fraction of the rider's groups present among the members and the fraction of members on the same budget tier. Ride style and personality compare the rider's value against the member average and lose one third for each point of difference on the one-to-three scale, which keeps the dimension above zero even at the widest gap. Distance starts at 1 for the same resort and decays linearly to 0 at four thousand kilometres.

The weighted sum is divided by 92 when the distance is judged. When either side lacks coordinates, the distance dimension is dropped from the calculation and the sum divides by 86, so an unknown distance neither adds nor subtracts. The network weight of 8 also stays out of the denominator, so enabling the feature can only add points and a trip's score is unchanged when the option is off. The result is rounded, capped at 100, and kept above zero for any positive score, and a trip that cannot be judged returns a null score with a reason code rather than a zero.

The endpoint returns up to three reasons drawn from the dimensions that contributed most, the full breakdown of the eight values, and the label of the pass that found the trip. That is what lets the result card on the Match hub explain itself. The scoring function is reproduced in Appendix A.

Table 3.1: Matching dimensions and their weights. The network weight is added outside the denominator.

Dimension	Weight	What it compares
Skill	30	Highest gear level on each side
Languages	20	Whether the rider and the members share at least one language
Groups	12	Share of the rider's interest groups present among the members
Budget	8	Share of members on the same budget tier
Ride style	8	Distance between the rider's value and the member average
Personality	8	Same comparison as ride style
Distance	6	Haversine distance to the resort, capped at 4000 kilometres
Network (optional)	8	Whether any member shares a network with the rider; added as a bonus
Denominator		92 with a distance anchor, 86 without one; the network weight stays out

3.2.4 Broadening and the Score Floor

A single exact query is a poor experience on a young marketplace. The number of public trips is small, and a rider who searches for one resort on one week can find nothing even when a trip two hundred kilometres away would have worked. The engine therefore runs a ladder of widening passes and stops as soon as it has enough results to show.

Figure 3.1 shows the ladder and Table 3.2 lists what changes at each pass. The first pass searches the exact resort and date window, and the second widens the dates by seven days on each side and keeps that window for every later pass. Passes three to six widen the location through a radius ladder of 50, 100, 200, and 400 kilometres, computed with the same haversine formula as the distance score. The seventh pass drops the location gate and the age and gender filters, and a trip is collected by the first pass that captures it, which becomes the scope label the result card shows.

The search stops after a pass when at least five results sit at or above the score floor of 35 percent. Nothing is hidden by the floor. Scores below 35 percent still come back, sorted behind the recommended ones, because a rider may prefer a weak match they understand to no result at all. A trip that cannot be judged at all, because it has no members and no distance anchor, receives a null score and a reason code, sorts after every judged result, and is never marked as recommended.

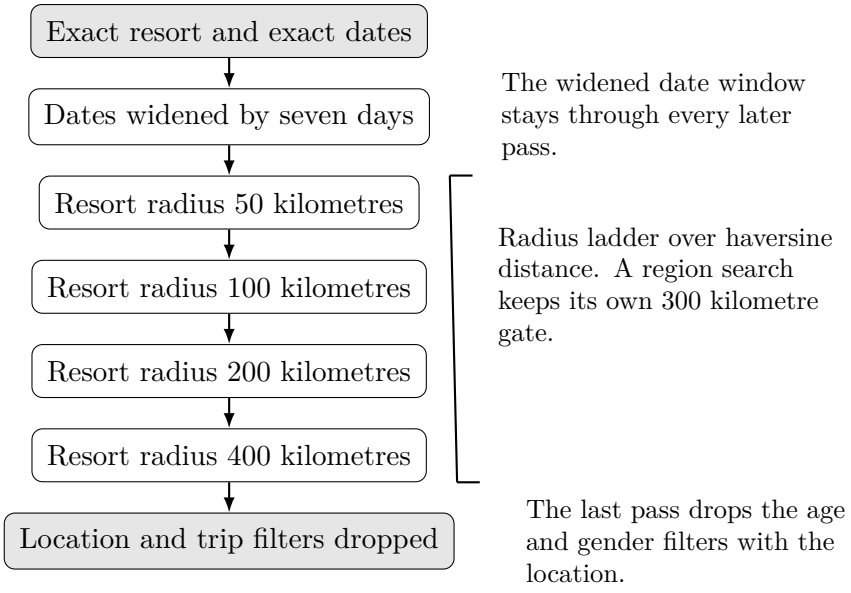


Figure 3.1: The broadening ladder. The search stops once at least five results sit at or above the 35 percent score floor; results below the floor are still returned, behind the recommended ones.

Table 3.2: Broadening passes in order. Trip filters are the age range and the gender list.

Pass	Dates	Location	Trip filters
1	Exact	Same resort	Applied
2	Plus or minus 7 days	Same resort	Applied
3	Plus or minus 7 days	Radius 50 km	Applied
4	Plus or minus 7 days	Radius 100 km	Applied
5	Plus or minus 7 days	Radius 200 km	Applied
6	Plus or minus 7 days	Radius 400 km	Applied
7	Plus or minus 7 days	Dropped	Dropped

3.2.5 Iterative Refinement

The matcher that shipped went through the iterations listed in Table 3.3. The first prototype, written before the internship, compared each trip against the preferences of its current members. A newly created trip has one member, the creator, so its score reflected one person and then changed as others joined, and a trip with no member data at all scored zero and appeared as the worst possible match rather than as an unknown one.

The search-based rewrite moved matching to a request with explicit criteria, and the v2 backend replaced the trip-side comparison with a snapshot of the rider's preferences and a single scoring pass. The trip types rework added public-only eligibility and the distance ladder. The score floor and the reasons arrived in the following weeks, after testing showed that a long list of low scores helped nobody, and the network dimension followed as an opt-in bonus for riders in the same club.

The null-score fix came last. It replaced the zero shown for a trip without member data or coordinates with a null score and a reason code, which stopped a missing measurement from reading as a bad match; those trips sort after the judged results and stay out of the recommended group.

Table 3.3: Matching iterations during the internship. Dates are merge dates.

Iteration	Date	Change
Prototype	2025	Scored each trip against the preferences of its current members
Search-based rewrite	July 13	Trip creation split into steps and matching rewritten around a search request (PR #20)
Matching v2 backend	July 29	Explicit search criteria, a v3 data migration, and one scoring pass (PR #48)
Trip types v4	August 3	Public-only eligibility and the resort distance ladder (PR #49)
Score floor and reasons	August 17	Floor at 35 percent, result grouping, reasons, and the breakdown (PR #57)
Network matching	August 18	Opt-in network dimension added outside the denominator (PR #60)
Null scores	August 24	Unjudgeable trips return a null score with a reason instead of zero (PR #64)

3.2.6 Limitations and Future Work

The engine is a scoring function with constants, and it does not learn. The weights in Table 3.1 reflect how the attributes in Section 1.4 decide whether a trip works, and they stayed fixed through the internship. A learned model needs outcomes to train on, and the product had a few months of trips to its name, while a static function is explainable, which mattered more at this stage.

The heavier dependency is data quality. A trip is scored through its members, so a trip whose members have not filled in their profiles is judged on less. When there is nothing to compare, the score says so; it cannot invent signal where none exists.

The next step would be to treat the weights as a starting point and adjust them from observable outcomes, such as a join that follows a recommendation. A second step would make the weights depend on the rider, since a beginner and an advanced skier do not value the same dimensions. Both require the kind of data the first release was designed to collect.

3.3 Building an Agentic Development Workflow (Piste)

3.3.1 Motivation

Every change on FrostLink went through a pull request and a device test before merge, which Section 1.9 describes as the team standard. The standard depended on a reviewer having time and context. When a review happened late, or the branch touched an area the reviewer had never opened, it slipped, and with three developers and a schema that kept moving, that happened often.

Piste is my attempt to make review quality less dependent on who happens to be free. It is a workflow built on opencode [9] that turns the team's habits into commands: a plan before code, a checklist for review, a fixed shape for findings, and PR descriptions written in a consistent structure. Nothing in the workflow decides on its own; it prepares the decision and stops. I am still the one who approves a plan, posts a review, or opens

a pull request. The company encouraged the use of AI tools during the internship, and Piste was my answer: a way to give my planning and review habits a repeatable shape without handing over the judgments. I built it for my own work and used it alone; the rest of the team did not adopt it.

3.3.2 Architecture

Piste is five commands, two subagents, and four skills, listed in Table 3.4. The commands are the entry points a person types; `/work` produces a phased plan, `/review-pr` and `/write-review` carry the review, and `/setup` and `/describe-pr` support the repository and the pull request.

The subagents are where the review work happens, and both are read-only by construction. Each agent declares its own permissions, so the reviewer can run `git diff`, `gh pr diff`, the repository search, the test command, and the type checker, and it cannot edit a file. The verifier holds a smaller allowlist that excludes `gh`, because it needs only the diff and the claim it is checking.

The four skills carry the standards instead of repeating them in each command. `plan-quality` defines what makes a plan worth approving, `pr-review` defines the severity levels and the review checklist, `git-workflow` covers branch and commit conventions, and `pr-writing-style` fixes the voice of a review and the structure of a PR description. Appendix D reproduces the review command.

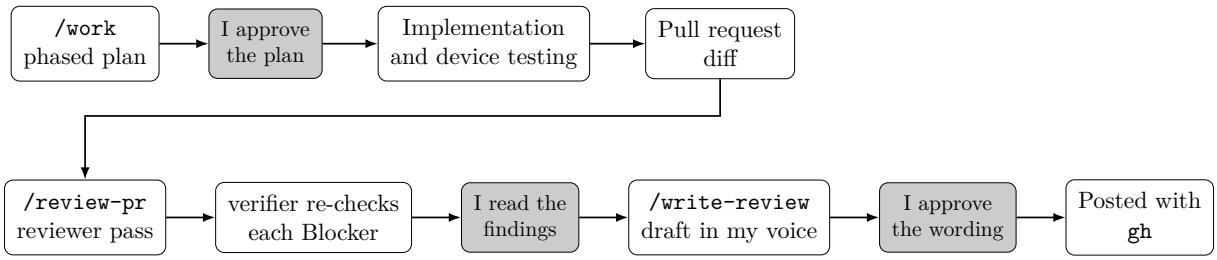


Figure 3.2: The Piste flow. Planning and building run in the top row, review and delivery in the bottom. Each shaded gate is a point where the workflow stops and waits for me, and the reviewer and verifier never edit the code they examine.

Table 3.4: Piste components. Both subagents are read-only, and each declares its own command allowlist.

Component	Kind	Role
`/setup`	Command	Explores a repository and fills in its agent instructions
`/work`	Command	Turns a ticket or a described feature into a phased plan and stops
`/review-pr`	Command	Runs the reviewer pass and verifies every Blocker finding
`/write-review`	Command	Drafts the final review in my voice and posts it after approval
`/describe-pr`	Command	Drafts a pull request description from the diff
reviewer	Subagent	Produces findings grouped by severity; never edits code
verifier	Subagent	Re-checks one Blocker claim with fresh eyes; never edits code
`plan-quality`	Skill	What makes a plan worth approving
`pr-review`	Skill	Severity levels and the review checklist
`git-workflow`	Skill	Branch naming and commit conventions
`pr-writing-style`	Skill	Voice for reviews and structure for PR descriptions

3.3.3 Human-in-the-Loop Design

The stopping rule is the part of Piste I would keep first. `/work` runs in plan mode and produces a plan; it does not create a branch or edit a file, and the plan exists so that a wrong assumption is caught before code makes it expensive. `/review-pr` produces findings and never fixes them. The reviewer agent is read-only, and its instructions say that a silent fix would hide a disagreement the reviewer should surface instead. `/write-review` drafts the final review and stops before posting, and the post runs through `gh` only after I approve the exact wording. `/describe-pr` drafts a description and does not open the pull request.

Each command ends where a person has to decide, and that is on purpose. A workflow that merged its own changes would be faster and would also detach the responsibility for a wrong merge from the person who understands the system, so the decisions stayed with me.

3.3.4 Adversarial Verification

A review agent can be confidently wrong: a finding phrased as fact about a path the reviewer did not follow to the end. The command's answer is a second opinion with no stake in the first one. Every finding marked as a Blocker is sent to the verifier, one fresh instance per finding, with only the diff and the claim. The verifier does not receive the reviewer's reasoning and is told not to assume the claim is correct. It returns one of three answers: confirmed, not confirmed, or uncertain, with a short explanation.

The report I read tags each Blocker with the verifier's answer, and a finding the verifier disproves is removed with a line that says it was dropped and why. A silently removed finding would leave me unsure whether the review had been thorough or just tidy.

3.3.5 Use on FrostLink

By mid-September I had reviewed 48 pull requests with the workflow, 41 of them inside the internship window. Those reviews produced 61 passes: 27 approvals, 22 change requests, and 12 comment-only responses. The numbers come from the repository's review history, and the work they represent is the ordinary review load described in Section 1.9.

I also used `/work` for my own tickets and `/describe-pr` for my own pull requests, including matching v2, the trip types rework, the score floor work, and the security rules suite. The workflow stayed a personal tool, installed into my working copy and used by me alone.

The value I could point to was consistency. Every pull request received the same checklist, the same severity definitions, and the same structure for the write-up, including the ones opened late in the week.

3.3.6 Limitations and Lessons

The workflow has limits I would not overstate away. The verifier reduces false positives; it does not remove them, and a Blocker that both agents accept can still be wrong. The commands read the code and the repository instructions, so their output is only as good as that context: a vague project description or an outdated command list produces vague findings. The parts of review that need a running application also remained mine, because

pushing a build to a device, watching a notification arrive, and testing a map on both platforms are outside anything the commands can do.

Piste works because it encodes standards that already existed and leaves the judgments in place. I cannot measure a speed improvement from it, but review became more even: every change received the same checklist, and the reasoning behind a finding was explicit enough that I could disagree with it.

3.4 Production Engineering: Theory vs Reality

Four problems showed up after the first builds, and none of them were visible in the architecture: a schema that kept changing while the rules depended on it, an Expo project owned by a personal account, an account deletion flow that looked like two delete calls, and platform differences that only a device revealed. The subsections below take them in turn and end each one with what fixed it.

3.4.1 Firebase Rules Reconciliation

The rules described in Section 2.7.2 were estimated at one week and took three. The estimate assumed a stable schema. The data model changed through July, and every change that touched a collection brought a rule question: a new field on the trip document, a new subcollection under the chats, a rename in the membership path. Each change sent me back into the rules file and back through the emulator suite.

The suite is what kept the rewrite affordable. It grew through the summer, and at its final size a rule change that broke a legitimate path failed locally in minutes instead of surfacing as a permission error during a device test. The failures forced decisions about which client writes were legal and which were not, and those decisions are how fields such as the account role and the moderation counters ended up closed to clients entirely.

The most expensive part was reconciliation. The rules branch and main diverged while chat, trips, and networks changed, and merging the two meant re-deciding several rules against code that had moved. The suite caught the regressions the merge introduced.

One gap remains, and it is worth stating plainly. Network membership and subgroup criteria gate which events appear in the Events feed, and event creation is checked on the server, but `joinTrip` does not re-check those conditions. A modified client could join a network event it cannot see. The fix would repeat the event criteria check inside `joinTrip`; it is not in the shipped version, and closing it is the first rules-related change I would make. The excerpt in Appendix B shows the helper functions and the trips block that the rules suite exercises.

3.4.2 Build and Release Engineering

Releases were manual by design and by circumstance. The repository has no continuous integration pipeline, and Section 2.8.2 describes the profiles and the checklist the team followed before every release. This subsection covers what that checklist does not, which is everything that breaks before a build exists.

The Expo project belonged to a developer's personal account rather than the company organization, which blocked builds for everyone but its owner; the fix moved it to the organization (PR #3). Configuration sat in the wrong place as well: the Android Google services file was not tracked, and the Google web client identifier was written into a screen,

so a build failed and a sign-in path depended on a source edit. Both moved into the application configuration (PR #5). Google Sign-In then needed the signing fingerprints registered with the OAuth client, which took a round of failed sign-ins to discover.

The release line carried three more faults that belonged to the platform rather than the code. An iOS build crashed at launch under Hermes in a production bundle because of Firebase Auth initialization, and the fix pinned the bundler to preserve class names and added an initialization fallback. The SDK 57 upgrade left the committed Android native directory stale, so cloud builds succeeded while local builds failed until the directory was regenerated (PR #96). A TestFlight build then failed Apple Sign-In because the entitlement was missing from the entitlements file (PR #101). None of the three showed up in the emulator.

3.4.3 Account Deletion and Auth Edge Cases

Account deletion is the case where the distance between the design and production was widest. The design was two operations: remove the user document and remove the authentication record. The implemented flow has to survive a cleanup that fails halfway, because a profile that is gone while the authentication record survives can sign back in to a half-deleted account.

The fix was order and a marker. The function tombstones the profile with a deletion status before it touches anything else, removes memberships, anonymizes messages, deletes chats that would be left with a single participant, and deletes the authentication record last. The client checks the marker at cold start and on every sign-in path, so a tombstoned account cannot return through a provider that still holds a session. A cleanup that fails leaves a visible record instead of a silent split.

Two rounds of testing added the cases the design missed. Cancelling registration after the authentication provider had created a user produced accounts with no profile, and the orphan fix (PR #61) removes the user from the client and falls back to a sign-out when the provider requires a fresh login. Re-authentication before deletion needed different handling for password, Google, and Apple, and the Google path cached a session that did not match the account being deleted, which produced a confusing failure until the sign-out before sign-in was made conditional. The work ran from early July to late August across more than a dozen commits.

3.4.4 Cross-Platform Fragmentation

One React Native codebase does not produce one behavior. The differences that cost time sat in the layers the framework touches but does not unify: keyboards, maps, pickers, notifications, and sign-in.

Keyboard handling produced three separate fixes. Android needed the window adjusted when the keyboard opened, which a configuration plugin made permanent across native regeneration. iOS then hid the chat composer behind the keyboard in a way the shared keyboard-avoiding wrapper did not solve, so the composer moved to a listener on the keyboard frame (PR #67). Tapping outside a text field also failed to dismiss the keyboard on iOS under the new architecture, which became a small component applied across the screens with text input (PR #70).

Maps were a configuration split: Android drew with Google Maps once a key was configured, while iOS drew a blank until the API key was wired into the native application

delegate and the Podfile (PR #79). Date pickers crashed on iOS when the picker was unmounted inside its own change handler, which the shared sheet had to work around (PR #69), and the profile date-of-birth picker opened twice on Android until it moved to the same sheet. Sign-in carried its own platform branches, from a native module regression on Android (PR #46) to the Apple entitlement and the nonce handling on iOS. Each fix was small. Together they were the cost of shipping one application on two platforms, and device testing was the only way they surfaced.

3.5 Evaluation and Results

This section reports what was verified and how. The security rules ran against the Firestore emulator, and the moderation parser had a small fixture suite. Matching did not have an automated test, and its checks were manual. The performance fixes were verified through their request patterns, and releases were verified by a human checklist. Table 3.5 summarizes the state of each area.

Table 3.5: What each area was verified against.

Area	Verification	Result
Security rules	Roughly 150 checks across nine areas against the emulator	Passing; regressions caught during reconciliation
Cloud Functions	Nine moderation parser fixtures over the streaming response shapes	Passing
Matching	Manual checks on real trip data, with no automated test	Score floor, broadening, and null scores confirmed by hand
Releases	Manual checklist on Android and iOS	Five versions from 1.1.2 to 1.2.1

3.5.1 Security Rules and Functions

The rules suite is the strongest verification in the project. It covers profile writes and blocks, trips, memberships, chats, reports, and application settings, and it asserts both directions: that the legitimate paths succeed and that the forged ones fail. The two kinds of assertions are close to balanced, with 60 success checks and 92 failure checks. The failure checks are the interesting half, because they encode the writes the client should not be able to perform, such as promoting the account role, editing a message beyond its reactions, or deleting a membership the caller does not own.

The suite proved its value during the rules reconciliation, where it failed on merged code that looked correct and caught boundaries the merge had opened. The rules are a specification written in a language nobody reads for pleasure, and the tests are the readable version.

The function tests cover the parts of the backend that are pure enough to isolate. The moderation pipeline parses a streaming label response, and nine fixtures pin the parser: the envelope shapes it accepts, the malformed bodies it rejects, and the cases where it fails closed and returns no result instead of a clear verdict.

3.5.2 Matching Validation

Matching has no automated test. The score function lives inside the HTTP endpoint, and I verified its behavior the way most of the application was verified during the internship:

by running the Match hub against real trip data and reading the results. The score floor was checked by confirming that results at or above 35 percent were grouped as recommended, that the list did not stop when fewer than five results cleared the floor, and that results below it still appeared behind the recommended ones.

The broadening ladder was checked the same way. Search points with few nearby trips moved the results down the ladder, and the label on each card matched the pass that had found that trip, which confirmed that a trip is collected once and reported with the right scope. The null-score fix was checked against trips with no members and trips whose members had no profile data: they returned the null score with the matching reason code, they sorted after the judged trips, and they no longer read as poor matches.

This is the weakest verification in the chapter. An automated test around the scoring function is the first thing I would add, and the Piste checklist asks for a regression test on every bug fix; the matching fixes were verified by hand instead.

3.5.3 Performance and Release Health

Section 1.8 lists two performance fixes. Neither produced a read total, so this report states what each change did to the request pattern rather than a number the project never collected. The Your Trips screen fetched each membership's trip document in sequence, so a member with 30 trips waited on 30 round-trips; the fix loads the documents concurrently, which collapses the wall-clock to roughly one round-trip. The Events feed ran three queries in sequence, one for the caller's memberships, one for the profile read, and one for the trips, and it returned events without an order; the fix runs the first two in parallel and applies a descending creation-time order with a limit of 80 and the composite index that supports it. Neither change altered the security rules, because the events query remained provable and the trip reads remained single-document reads.

Release health was measured by shipping. Five versions went out between August 6 and September 14, across Android and iOS, including the tester APK profile and TestFlight builds. The verification around each release stayed manual: the rules suite, the function tests, and a device pass on each platform. I have no crash-rate figures to report, and the project never collected them.

3.6 Conclusion and Reflections

Building FrostLink taught me the difference between a project that works and a product that ships. The matching engine turns a judgment call into a formula that the person it affects can read. Piste carries a review standard without taking the decisions away from the reviewer. The production section follows the assumptions that stopped holding once the system met users.

What stayed mine is what could not be delegated: picking the weights other people will disagree with, reading a drafted review line by line before it goes out, testing the build on a device before a tester sees it, and saying plainly that a gap is still open. A tool or a formula can carry the routine. The judgment stays with the engineer.

General Conclusion

FrostLink is developed by Tanit Consulting AB, and the internship covered nine weeks there, from July 1 to August 31, 2026, and continued part time into September. Three releases shipped inside the window, 1.1.2 through 1.1.4, and the 1.2.x line followed two weeks later.

The parts I understand best are the matching engine and the security rules. Matching began as a comparison against the members each trip already had and ended as a weighted score over seven dimensions, with a broadening ladder, a 35 percent score floor, and null scores for trips the engine cannot judge. The security rules grew into a 387-line file, and their emulator suite caught regressions that the merged code had opened. Account deletion was rebuilt around a tombstone after testing showed how a cleanup that fails halfway could split an account in two.

Three gaps remain. `joinTrip` does not re-check the network and subgroup criteria that gate the Events feed, so a modified client could join an event it cannot see. Matching has no automated test, and its verification was manual, on real trip data. The release process has no continuous integration, and every version shipped through the same human checklist. Those are the first changes I would make, in that order.

The internship changed what I count as finished work. In coursework the environment holds still and the code is the deliverable. Here the schema moved under the rules, the two platforms disagreed about keyboards and maps, and a build was only as good as its last device test. I leave with two habits: write the plan before the code, and state the limits of the evidence plainly.

Bibliography

[1] Broski. https://www.broski.app. Accessed September 2026.

[2] buddies.ski: Find your perfect ski partner. https://buddies.ski. Accessed September 2026.

[3] Cloud firestore data model. https://firebase.google.com/docs/firestore/data-model. Accessed September 2026.

[4] Eas build. https://docs.expo.dev/build/introduction/. Accessed September 2026.

[5] Expo documentation. https://docs.expo.dev. Accessed September 2026.

[6] Firebase documentation. https://firebase.google.com/docs. Accessed September 2026.

[7] Get started with cloud firestore security rules. https://firebase.google.com/docs/firestore/security/get-started. Accessed September 2026.

[8] Meetup. https://www.meetup.com. Accessed September 2026.

[9] opencode documentation. https://opencode.ai/docs. Accessed September 2026.

[10] React native documentation. https://reactnative.dev/docs/getting-started. Accessed September 2026.

[11] Ski buddies. https://www.skibuddies.org. Accessed September 2026.

[12] Skibuddy app. https://myskibuddy.app. Accessed September 2026.

[13] Skibuds: Find ski and snowboard friends. https://skibudsapp.com. Accessed September 2026.

[14] Snoji: Ski trips and buddies. https://apps.apple.com/us/app/snoji-ski-trips-buddies/id6749374694. App Store listing, accessed September 2026.

[15] Snowflake is the ski tinder app. https://www.snowpak.com/news/snowflake-is-the-ski-tinder-app. SnowPak, accessed September 2026.

[16] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development. https://agilemanifesto.org, 2001. Accessed September 2026.

[17] Ralph L. Keeney and Howard Raiffa. *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. John Wiley and Sons, 1976.

[18] Ken Schwaber and Jeff Sutherland. The scrum guide: The definitive guide to scrum: The rules of the game. `https://scrumguides.org`, 2020. Accessed September 2026.

[19] Roger W. Sinnott. Virtues of the haversine. *Sky and Telescope*, 68(2):159, 1984.

Appendix A

Matching Algorithm Excerpts

The listing reproduces the scoring core of `matchTrip` with the comments shortened and the blank lines removed. The weights themselves are listed in Table 3.1; the full function also builds the per-dimension breakdown and the reason factors that the Match hub shows.

```
function scoreTrip(
  trip: any,
  si: ScoringInput,
  members: any[],
): { normalized: number; judgeable: boolean; scoreStatus: 'no_members' | 'no_member_data' | null; subs:
    SubScores } | null {
  try {
    // A trip is judgeable when a member carries matching data or both sides
    // anchor a distance. Unjudgeable trips map to score: null, never 0.
    const hasMemberData = members.some((m) =>
      (Array.isArray(m?.preferences?.gear) && m.preferences.gear.length > 0) ||
      (Array.isArray(m?.languages) && m.languages.length > 0) ||
      (Array.isArray(m?.preferences?.groups) && m.preferences.groups.length > 0) ||
      !!m?.preferences?.budgetTier ||
      m?.preferences?.ridePreference != null ||
      m?.preferences?.personality != null ||
      (m?.networks && Object.keys(m.networks).length > 0)
    );
    // Skill
    let membersHighest = 0;
    for (const m of members) {
      const gear = Array.isArray(m?.preferences?.gear) ? m.preferences.gear : [];
      for (const g of gear) {
        const lvl = Number(g?.level || 0);
        if (!Number.isNaN(lvl) && lvl > membersHighest) membersHighest = lvl;
      }
    }
    const skillMatch = membersHighest && si.userHighest
      ? (membersHighest === si.userHighest ? 1 : (Math.abs(membersHighest - si.userHighest) <= 1 ? 0.7
    : 0))
      : 0;
    // Language
    let languageScore = 0;
    let languageSharedCount = 0;
    if (members.length) {
      const memberLangs = new Set<string>();
      members.forEach((m) => (Array.isArray(m?.languages) ? m.languages.forEach((l: string) =>
    memberLangs.add(l)) : null));
      const shared = si.userLangs.filter((l) => memberLangs.has(l));
      languageSharedCount = shared.length;
      languageScore = shared.length ? 1 : 0;
    }
    // Groups
    const memberGroups = new Set<string>();
    members.forEach((m) => (Array.isArray(m?.preferences?.groups) ? m.preferences.groups.forEach((g:
    string) => memberGroups.add(g)) : null));
    const commonGroups = si.userGroups.filter((g) => memberGroups.has(g));
    const groupsSharedCount = commonGroups.length;
    const groupsScore = si.userGroups.length ? (commonGroups.length / si.userGroups.length) : 0;
```

```
    // Budget
    let budgetScore = 0;
    if (si.userBudget && members.length) {
      const same = members.filter((m) => m?.preferences?.budgetTier === si.userBudget).length;
      budgetScore = same / members.length;
    }
    // Ride and personality
    const avgRide = members.length
      ? members.reduce((acc, m) => acc + (Number(m?.preferences?.ridePreference) || 0), 0) / members.
     length
      : null;
    const rideScore = (si.userRidePref != null && avgRide != null)
      ? Math.max(0, 1 - Math.abs(si.userRidePref - avgRide) / 3)
      : 0;
    const avgPersonality = members.length
      ? members.reduce((acc, m) => acc + (Number(m?.preferences?.personality) || 0), 0) / members.
     length
      : null;
    const personalityScore = (si.userPersonality != null && avgPersonality != null)
      ? Math.max(0, 1 - Math.abs(si.userPersonality - avgPersonality) / 3)
      : 0;
    // Distance, when both sides anchor a point. Either side missing one
    // removes the dimension from the denominator instead of counting a zero.
    let distanceScore = 0;
    let distanceKm: number | null = null;
    let distanceKnown = false;
    if (si.searchLat != null && si.searchLon != null &&
        trip?.location?.lat != null && trip?.location?.lon != null) {
      distanceKnown = true;
      const dkm = haversineKm(si.searchLat, si.searchLon, trip.location.lat, trip.location.lon);
      distanceKm = Math.round(dkm);
      const clipped = Math.min(4000, dkm);
      distanceScore = 1 - clipped / 4000;
    }
    // Unjudgeable when no member carries matching data and distance is unknown.
    const judgeable = distanceKnown || hasMemberData;
    const scoreStatus = judgeable
      ? null
      : (members.length > 0 ? 'no_member_data' : 'no_members');
    // Network: opt-in, binary, and additive. The weight never enters the
    // denominator, so a trip keeps its score unless a network matches.
    let networkScore = 0;
    let matchedNetworkIds: string[] = [];
    if (si.networkEnabled && si.userNetworks.length) {
      const callerNetworks = new Set(si.userNetworks);
      const shared = new Set<string>();
      for (const m of members) {
        const nets = m?.networks;
        if (!nets) continue;
        for (const id of Object.keys(nets)) {
          if (callerNetworks.has(id)) shared.add(id);
        }
      }
      matchedNetworkIds = Array.from(shared);
      networkScore = matchedNetworkIds.length > 0 ? 1 : 0;
    }
    const weightSum = (distanceKnown ? WEIGHT_SUM : WEIGHT_SUM - SCORING_WEIGHTS.distance)
                    - SCORING_WEIGHTS.network;
    const rawScore =
      (SCORING_WEIGHTS.skill * skillMatch) +
      (SCORING_WEIGHTS.language * languageScore) +
      (SCORING_WEIGHTS.groups * groupsScore) +
      (SCORING_WEIGHTS.budget * budgetScore) +
      (SCORING_WEIGHTS.ridePref * rideScore) +
      (SCORING_WEIGHTS.personality * personalityScore) +
      (SCORING_WEIGHTS.distance * distanceScore) +
      (SCORING_WEIGHTS.network * networkScore);
    // Clamp at 100 because the network bonus can lift a raw score past 100,
    // and floor a positive score at 1 so a small real match never reads as 0.
    const rounded = Math.round((rawScore / (weightSum || 1)) * 100);
    const normalized = Math.min(100, rawScore > 0 ? Math.max(1, rounded) : rounded);
    return {
      normalized,
```

```
      judgeable,
      scoreStatus,
      subs: {
        skill: skillMatch,
        language: languageScore,
        groups: groupsScore,
        budget: budgetScore,
        ride: rideScore,
        personality: personalityScore,
        distance: distanceScore,
        languageSharedCount,
        groupsSharedCount,
        distanceKm,
        network: networkScore,
        matchedNetworkIds,
      },
    };
  } catch {
    return null;
  }
}
```

Appendix B

Firestore Security Rules Excerpts

The listing reproduces the helper functions and the trips match block from the 387-line rules file. The chat, membership, report, and settings blocks follow the same shape: deny by default, prove every write, and constrain list queries to the ones the application issues.

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    function signedIn() {
      return request.auth != null;
    }

    function isOwner(uid) {
      return signedIn() && request.auth.uid == uid;
    }

    function isTripMember(tripId) {
      return exists(/databases/$(database)/documents/tripMembers/$(tripId + '_' + request.auth.uid));
    }

    function isTripAdmin(tripId) {
      return signedIn()
        && get(/databases/$(database)/documents/trips/$(tripId)).data.admins is list
        && get(/databases/$(database)/documents/trips/$(tripId)).data.admins.hasAny([request.auth.uid])
     ;
    }

    // Resource-based admin check on the trip being written: safe inside
    // transactions, where the get() above would raise an evaluation error.
    function isTripAdminNow() {
      return signedIn() && resource.data.admins is list && request.auth.uid in resource.data.admins;
    }

    match /trips/{tripId} {
      // Public and event trips are discoverable; anything else (private,
      // ski_quest) is readable only by its members and admins.
      // get vs list split: a single-doc read of a missing trip must return
      // not-found rather than a deny, because the client distinguishes the
      // two with snapshot.exists(). List queries only evaluate existing
      // documents, so no null branch is needed there.
      allow get: if signedIn() && (
        resource == null
        || resource.data.type in ['public', 'event']
        || isTripMember(tripId)
        || (resource.data.admins is list && resource.data.admins.hasAny([request.auth.uid]))
      );
      // List reads are only provable for the type-constrained discovery
      // queries the application uses (public and event). Member and admin
      // branches cannot be proven in a list query, and including them would
      // silently deny every other list read.
      allow list: if signedIn() && resource.data.type in ['public', 'event'];
      allow create: if false;
      allow update: if isTripAdminNow();
      // Deleting a document that no longer exists is a no-op, so double
```

```
49        // leave and double delete races pass without an evaluation error.
50        allow delete: if !exists(/databases/$(database)/documents/trips/$(tripId)) || isTripAdmin(tripId)
      ;
51      }
52    }
53  }
```

Appendix C

Additional Screenshots

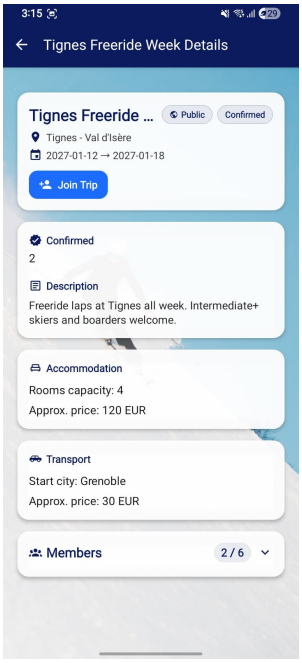


Figure C.1: Trip details screen for a trip the viewer has not joined, with the join action and the member count.

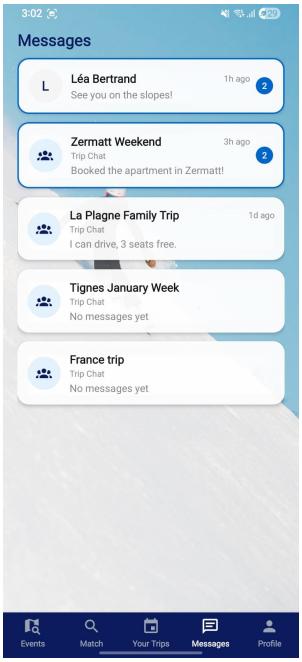


Figure C.2: The unified chat screen with trip rooms and direct messages.

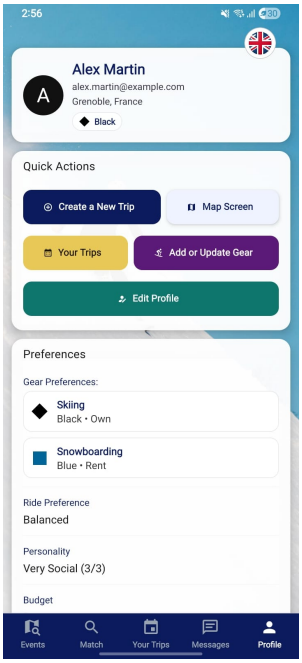


Figure C.3: The profile screen with the gear and riding preferences.

Appendix D

Piste: Commands, Agents, and Skills

The commands, subagents, and skills are listed in Table 3.4. The listing below reproduces the review command, which runs the adversarial verification described in Section 3.3.4: every Blocker finding is re-checked by a fresh instance before it reaches me.

```
---
description: Review a diff/PR - depth scales to what each part of the diff actually
    needs, auto-verifies any Blocker
agent: plan
---

Review target: $ARGUMENTS

No argument -> review the current branch's diff against its base. A PR number/URL ->
     fetch that
PR's diff (`gh pr diff`) and description instead.

Steps:

1. Invoke `@reviewer` with the diff (and description, if reviewing a PR) to produce
    findings
   grouped by severity, per the `pr-review` skill. It calibrates its own depth per
    part of the
   diff - no flag needed, it goes deeper on risky or uncertain areas on its own.
2. For every finding marked **Blocker**: invoke a fresh `@verifier` instance per
    finding, giving
   it only the diff and that specific claim - not the reviewer's reasoning. Do this
    automatically,
   without asking.
3. Present the final report with each Blocker tagged **confirmed**, **not confirmed
    **, or
   **uncertain** based on the verifier's answer. Drop findings the verifier
    disproved, but say so
   explicitly rather than silently removing them ("Blocker dropped after
    verification: ...").
4. Show findings with full file:line detail - don't summarize away specifics. Keep
    the report to
   the verdict line and findings: no praise section, no narration about the review
    process.
```