Technical University of Crete
School of Electrical and Computer Engineering

Diploma Thesis

Development of a Social Ridesharing App to Serve the Members of the Technical University of Crete Community

---

Spyridon Charitakis

Thesis Committee
Supervisor: Professor Georgios Chalkiadakis (TUC)
Professor Michail G. Lagoudakis (TUC)
Assistant Professor Nikolaos Spanoudakis (HMU)

Chania, October 2025

Πολυτεχνειο Κρητης
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων

Διπλωματική Εργασία

Ανάπτυξη Εφαρμογής Κοινωνικού Διαμοιρασμού Διαδρομών για τα Μέλη της Κοινότητας του Πολυτεχνείου Κρήτης

Σπυρίδων Χαριτάκης

Επιτροπή
Επιβλέπων: Καθηγητής Γεώργιος Χαλκιαδάκης (ΠολΚρ)
Καθηγητής Μιχαήλ Γ. Λαγουδάκης (ΠολΚρ)
Επίκουρος Καθηγητής Νικόλαος Σπανουδάκης (ΕλΜεΠα)

Χανιά, Οκτώβριος 2025

Thesis Committee:

Georgios Chalkiadakis (Supervisor)
Professor, School of Electrical and Computer Engineering
Technical University of Crete, Chania, Greece

Michail G. Lagoudakis
Professor, School of Electrical and Computer Engineering
Technical University of Crete, Chania, Greece

Nikolaos Spanoudakis
Assistant Professor, Department of Electronic Engineering
Hellenic Mediterranean University, Greece

Abstract

Ridesharing, the joint trips among multiple users using a common vehicle, is emerging as one of the most useful innovations in modern urban transportation. In the context of increasing environmental challenges and the need for sustainable transportation solutions, this practice offers significant benefits. Not only by reducing carbon dioxide emissions, but by also providing more cost-effective and flexible transportation options, as well as alleviating road traffic congestion.

This diploma thesis focuses on the development of a multiplatform application aimed at promoting group travel within the Technical University of Crete (TUC) community, in the city of Chania. The application stands out for its ability to offer an all-in-one product for both passengers and drivers by including both a passenger and a driver mode. Users, after selecting their role, can either create a new ride to help fellow members of the TUC community reach their destination, or make a request in order to receive suggested rides that fit their spatial, time, and capacity criteria. Passengers and drivers can see all available rides and all available requests in real time through dedicated screens. All users, with minimal personal data displayed, are listed on the Community screen.

Initially, due to the geographical characteristics of the area where the Technical University of Crete is located, we separated the points of interest inside the Campus into two lines. This separation was made so that detours are avoided. Then, we selected the five most popular routes, to and from the Technical University of Crete, and created points on them to build complete routes. A two-stage greedy algorithm is utilized to propose suggested rides to passengers and they, in their turn, can choose which one to enter. The algorithm can also create cross-line suggestions, meaning that even if the passenger's route is partially included in the driver's route, with a slight change in the pickup or drop-off point to match the driver's, more passengers can be served. The entire process is fully assisted with notifications and real-time data updates using a combination of WebSockets and the Publish/Subscribe messaging service. The selected architecture is the Model-View-ViewModel (MVVM) so that we can achieve a clean separation of concerns. Design-wise the Android part follows the Material You guidelines and its iOS counterpart, the Apple Human Interface guidelines. That way we ensured that the UI is fully aligned with the latest design standards from Google and Apple. After the beta version release of the application, user evaluations and feedback was used to identify bugs, improve usability, and shape its future versions.

Beyond improving the waiting and travel experience by reducing both times, this application actively contributes to reducing the number of vehicles on the road, thereby decreasing the carbon footprint and promoting

environmental sustainability. By using this application, the community of the Technical University of Crete can become a pioneer in utilizing innovative urban transportation solutions. This could serve as a model of sustainable mobility and inspire other communities. The contribution of this thesis to the development and application of innovative solutions in the field of ridesharing reflects an effort towards a sustainable future for mobility, offering users a high-quality and efficient experience.

Περίληψη

Οι συνεπιβατικές διαδρομές μεταξύ πολλαπλών χρηστών χρησιμοποιώντας ένα κοινό όχημα, αναδεικνύονται ως μια από τις πιο χρήσιμες καινοτομίες στις σύγχρονες αστικές μεταφορές. Στο πλαίσιο των αυξανόμενων περιβαλλοντικών προκλήσεων και της ανάγκης για βιώσιμες λύσεις μεταφορών, αυτή η πρακτική προσφέρει σημαντικά οφέλη. Όχι μόνο με τη μείωση των εκπομπών διοξειδίου του άνθρακα, αλλά παρέχοντας πιο οικονομικές και ευέλικτες επιλογές μεταφοράς, καθώς και ανακούφιση από την κυκλοφοριακή συμφόρηση.

Η παρούσα διπλωματική εργασία εστιάζει στην ανάπτυξη μιας πολυπλατφορμικής εφαρμογής που στοχεύει στην προώθηση των ομαδικών μετακινήσεων εντός της κοινότητας του Πολυτεχνείου Κρήτης, στην πόλη των Χανίων. Η εφαρμογή ξεχωρίζει για την ικανότητά της να προσφέρει ένα ολοκληρωμένο προϊόν, τόσο για επιβάτες, όσο και για οδηγούς, υλοποιώντας λειτουργία επιβάτη, αλλά και οδηγού. Οι χρήστες, αφού επιλέξουν τον ρόλο τους, μπορούν είτε να δημιουργήσουν μια νέα διαδρομή για να βοηθήσουν τα υπόλοιπα μέλη της κοινότητας του Πολυτεχνείου Κρήτης να φτάσουν στον προορισμό τους, είτε να υποβάλουν αίτημα για να λάβουν προτεινόμενες διαδρομές που ταιριάζουν στα χωρικά, χρονικά και χωρητικά κριτήρια που επέλεξαν. Οι επιβάτες και οι οδηγοί μπορούν να βλέπουν όλες τις διαθέσιμες διαδρομές και όλα τα διαθέσιμα αιτήματα, σε πραγματικό χρόνο μέσω ειδικών οθονών. Όλοι οι χρήστες, με την προβολή ελάχιστων προσωπικών δεδομένων, εμφανίζονται στην οθόνη «Community».

Αρχικά, λόγω των γεωγραφικών χαρακτηριστικών της περιοχής στην οποία βρίσκεται το Πολυτεχνείο Κρήτης, χωρίσαμε τα σημεία ενδιαφέροντος εντός της Πολυτεχνειούπολης σε δύο γραμμές. Αυτός ο διαχωρισμός έγινε ώστε να αποφευχθούν οι παρακάμψεις. Στη συνέχεια, επιλέξαμε τις πέντε πιο δημοφιλείς διαδρομές από και προς το Πολυτεχνείο Κρήτης και δημιουργήσαμε σημεία σε αυτές για να κατασκευάσουμε ολοκληρωμένες διαδρομές. Χρησιμοποιήσαμε έναν άπληστο αλγόριθμο δύο σταδίων ο οποίος δημιουργεί προτεινόμενες διαδρομές για τους επιβάτες οι οποίοι με τη σειρά τους, μπορούν να επιλέξουν σε ποια θα ενταχθούν. Ο αλγόριθμος μπορεί επίσης να δημιουργήσει προτάσεις διασταυρούμενων διαδρομών, πράγμα που σημαίνει ότι ακόμη και αν η διαδρομή του επιβάτη περιλαμβάνεται εν μέρει στη διαδρομή του οδηγού, με μια μικρή αλλαγή στο σημείο επιβίβασης ή αποβίβασης για να ταιριάξει με αυτό του οδηγού, μπορούν να εξυπηρετηθούν περισσότεροι επιβάτες. Όλη η διαδικασία υποστηρίζεται πλήρως με ειδοποιήσεις και ενημερώσεις δεδομένων σε πραγματικό χρόνο με έναν συνδυασμό WebSockets και την υπηρεσία μηνυμάτων Publish/Subscribe. Η επιλεγμένη αρχιτεκτονική είναι η Model-View-ViewModel (MVVM), ώστε να επιτύχουμε σαφή διαχωρισμό αρμοδιοτήτων. Από άποψη σχεδιασμού, η υλοποίηση Android ακολούθησε τις κατευθυντήριες γραμμές του Material You και η αντίστοιχη iOS αυτές του Apple Human Interface. Με

αυτόν τον τρόπο, διασφαλίσαμε ότι η διεπαφή χρήστη εναρμονίζεται πλήρως με τα πιο πρόσφατα πρότυπα σχεδίασης των Google και Apple. Μετά την κυκλοφορία της beta έκδοσης της εφαρμογής, οι αξιολογήσεις και τα σχόλια των χρηστών χρησιμοποιήθηκαν για τον εντοπισμό σφαλμάτων, τη βελτίωση της χρηστικότητας και τη διαμόρφωση των μελλοντικών εκδόσεών της.

Πέρα από τη βελτίωση της εμπειρίας αναμονής και μετακίνησης μειώνοντας και τους δύο χρόνους, αυτή η εφαρμογή συμβάλλει ενεργά στη μείωση του αριθμού των οχημάτων στο δρόμο, περιορίζοντας έτσι το αποτύπωμα άνθρακα και προωθώντας την περιβαλλοντική βιωσιμότητα. Χρησιμοποιώντας αυτήν την εφαρμογή, η κοινότητα του Πολυτεχνείου Κρήτης μπορεί να γίνει πρωτοπόρος στην εφαρμογή καινοτόμων λύσεων αστικών μεταφορών. Αυτό θα μπορούσε να χρησιμεύσει ως ένα μοντέλο βιώσιμης κινητικότητας και να εμπνεύσει και άλλες κοινότητες. Η συμβολή της παρούσας διπλωματικής εργασίας στην ανάπτυξη και εφαρμογή καινοτόμων λύσεων στον τομέα της κοινής χρήσης οχημάτων, αντικατοπτρίζει μια προσπάθεια για ένα βιώσιμο μέλλον στον τομέα της κινητικότητας, προσφέροντας στους χρήστες μια υψηλής ποιότητας και αποδοτική εμπειρία.

Acknowledgements

I would like to thank all those people that contributed to making this thesis possible. My supervisor Professor Georgios Chalkiadakis, for his guidance and help throughout all the steps of this work and for his thorough review of this thesis.

Moreover, I would like to thank Professor Michail G. Lagoudakis; and Assistant Professor Nikolaos Spanoudakis who supported me throughout the development by troubleshooting and advising me in the best possible way.

Lastly, I would like to thank my wife, my family, and my friends. Elena, Emmanouil, Evangelia, Anna, Evelina, Antonis, and Kostas thank you for all your support throughout this process. This thesis is for you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Sharing economy [1, 2, 3] is a type of economic paradigm with its main characteristic being the sharing among peers. It typically involves peer-to-peer transactions for the shared use of goods and services using the Internet to share space, resources, and workforce. Uber, Lyft, Airbnb, and VRBO are among those companies that utilize this model. They take advantage of the underutilization of these products and services in order to benefit financially but at the same time reduce waste, costs, and improve the overall efficiency within the community. Likewise, sustainable means of transportation, such as walking, cycling, electric scouter usage, and metro/tram, promote and target the reduction of vehicle use, minimization of environmental carbon footprint and cost. Combining these two, we can potentially create greener, more functional, and friendlier cities for the decades to come.

Now, ridesharing [4, 5, 6, 7] is fundamental to collaborative sustainable mobility. It is a form of shared transportation where different individuals share a common vehicle for their trip. This leads to minimization of individual costs for each participant and maximization of efficiency. There are two main categories of ridesharing. The first is the non-commercial ridesharing which is a peer-to-peer, cost-sharing, and non-profit way of sharing a vehicle between people that travel in the same direction. In most cases, the participants of the coalition are not professionals, and the arrangement occurs without using a specific platform, thus not considered a commercial service. The second is called ride-hailing or mobility as a service (MaaS). In this case, there is a professional driver that offers their services through applications like Uber and Bolt. There is no cost-sharing, but rather a fee that depends on the distance and the time that the vehicle is occupied.

The advantages of ridesharing are multiple and considerable [6]. A large-scale adoption of this model could significantly reduce traffic congestion in urban environments by minimizing the number of moving vehicles needed to serve the needs of their citizens. Furthermore, the reduced costs of traveling by splitting the fuel and/or other costs makes it more attractive for all the parties involved. Reduced emissions would improve air quality and make the energy profile of communities cleaner.

It is particularly important to implement a ridesharing solution because public transportation in Chania is expensive and operates on irregular schedules. Moreover, the routes are infrequent, there is crowding during rush hours, and apart from the students, is barely used by anyone else. This mostly negative image is painted and reinforced by recent studies, such as that of Bogiatzis [8], as well as the Public Services Evaluation Report conducted by the Greek Government [9]. In the latter, the Chania public transportation system scored an underwhelming 4,7 out of 10, based on citizens' opinions.

In this thesis, we handle the ridesharing problem of the Technical University of Crete, by utilizing a peer-to-peer solution within the community but without implementing a cost-sharing mechanism. In addition to taking into account the spatial, time, and capacity constraints, we created popular graph-based routes so that we can simplify the ridesharing procedure. This is the first time for a Greek thesis, where a complete cross-platform ridesharing application is not only prototyped, but beta tested in real-world conditions, shipped, and listed on both major Stores.

1.1 Thesis Contributions

The main objective of ridesharing is to increase the time and travel efficiency of people who need a vehicle and those who drive within the same area. In this thesis, we focus on creating a cross-platform ridesharing solution specifically for the Technical University of Crete community.

- This is the first time a cross-platform ridesharing system is developed at a Greek institution of higher education, and to the best of our knowledge it is the first implementation ever in the context of a thesis, with the technology stack we used. Presumably, it is among the very few cross-platform social ridesharing apps, developed specifically for university communities worldwide.
- We identified requirements for this problem domain, specific use case scenarios and given these and the particulars of the problem application domain, i.e. that we are talking about the TUC Campus and Chania city, we specified a ridesharing protocol that efficiently connects passengers and drivers in a simplistic and convenient way.
- We created graphs with specific stops that connect the Campus to the five most popular destinations in the city, on which users can travel bidirectionally.
- The application utilizes the CAS authentication protocol to ensure that only verified members of the Technical University of Crete can access and use it, something that enhances safety, trust, and accountability.

- Both passenger and driver modes are implemented, something that gives users the flexibility to use the same application regardless of their role (passenger or driver) without having to switch to a different application. Both types of users can specify their time, spatial, and capacity criteria, with the former submitting a request and the latter creating a ride. Then a two-stage greedy algorithm matches them using graph-theory-based routes and generates suggested rides for the passengers, who in their turn can join.
- Passengers can see all available rides and drivers can see all passenger requests; all in real time through dedicated screens, something that improves visibility on both sides.
- A Community screen is implemented, which displays all TUC users of the application (respecting any sensitive data), which increases engagement and the feeling of being part of a community.
- It is easy for users to use this application, not only because of the application itself, but also because of its operation, which resembles that of a fixed-route bus service.
- We used various technologies to create this application, with the base being a Kotlin Multiplatform Project (KMP) that houses the Android frontend that was built using Kotlin and the Jetpack Compose framework, and the common code that is responsible for the communication with the backend. Xcode was used for the development of the iOS/iPadOS/MacOS/VisionOS frontend using the Swift language and the SwiftUI framework. That makes the application able to run on a large number of devices from iPhones, iPads, Android phones, foldables and tablets, to Mac computers and even the latest virtual reality headset from Apple, the Apple Vision Pro with custom layouts where needed.
- The backend utilizes the FastAPI framework with Python and various other solutions so that the application can operate 24/7/365.

The testing phase is taking place in the City of Chania from the 21st of July 2025 and is expected to continue, until at least the end of September 2025. Throughout this period, users can provide feedback through the application.

Our thesis has already resulted in a conference publication: Spyridon Charitakis, Nikolaos Spanoudakis, Georgios Chalkiadakis : "TUC Ridesharing: A University-Oriented Social Ridesharing App", 2nd International Conference "Circular Economy: The pathway towards a Sustainable Development", Chania, Greece, September 17-19, 2025.

1.2 Thesis Outline

The rest of this thesis is organized into five chapters as outlined below. Chapter 2 contains all the theoretical background that helped us shape the foundation of this work. We present some basic concepts from graph theory, cooperative game theory, and coalition formation in various domains. Then we review related work, where we feature practical implementations of previous work in the field of ridesharing, as well as more or less game-theoretic approaches and work done at the Technical University of Crete in the ridesharing domain. In Chapter 3 we highlight our approach to solving the transportation problem of the TUC community. There, we list the functional and non-functional requirements of the application, the basic use cases, the ridesharing protocol, as well as a detailed view of our match-making algorithm and a presentation of the in-app usage experience through two real-world usage scenarios. In Chapter 4, we present the frontend and backend design and implementation of our application, demonstrating the architecture, IDEs, languages, frameworks, and we do a presentation of the user interface together with the different layers that combined, shape our full-stack application. Chapter 5 contains the evaluation of our work that took place in the city of Chania during the beta testing period. Lastly, in Chapter 6 we conclude our thesis and propose future extensions to enhance the application's usability, features, and user experience.

Chapter 2

Background and Related Work

In this chapter we cover the theoretical background of our work. We present basic notions of Graph Theory that we utilized when we created the routes, and helped us model the network, with each stop representing a node, and the segments that connect them the edges.

We also present concepts from cooperative game theory and coalition formation, as part of our work was to put forward a protocol by which teams (or "coalitions") of people are formed to be put together into a vehicle–given specific criteria used to this end. Finally, we present work related to the ridesharing domain.

2.1 Graph Theory

Graph theory [10] is the research and study of the mathematical structures (known as graphs) used to model the dyadic relations between discrete objects. Graphs have various applications in everyday life and cover many fields due to their ability to model connections and relationships. They are used in neural networks, algorithms and data structures, as well as supply chain networks and neuroscience. Graphs consist of vertices and edges, with the latter connecting pairs of vertices.

Definition 2.1. (Graph [11]) *A Graph G = (V(G),E(G)) or G = (V,E) consists of two finite sets. V (G) or V, the vertex set of the graph, which is a non-empty set of elements called vertices and E(G) or E, the edge set of the graph, which is a possibly empty set of elements called edges, such that each edge e in E is assigned as an unordered pair of vertices (u, v), called the end vertices of e.*

We define |V| = n to be the order of G and |E| = m to be the size of G

Periodically, we might encounter a vertex where both of its ends connect to the same edge. In this case, we call the vertex a loop or a self-loop. By defining loops, we can also define simple graphs. Simple graphs are graphs that do not

have parallel edges or loops. Multigraphs on the other hand, can have multiple edges between two vertices. Examples of a simple graph and a multigraph can be seen in Figure 2.1 from [11] below.

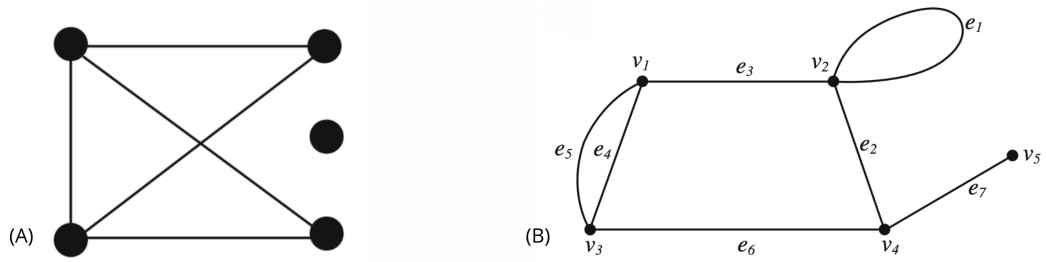


FIGURE 2.1: A simple graph in (A) and a multigraph in (B).

Sometimes in real life, when we have to model a problem, a graph is not enough. For example, when we model electrical networks, one-way traffic in the streets of a city or the states of a finite-state machine in software engineering. In these occasions we use directed graphs, and in this way we can leverage the natural way directed graphs show the relation of sequence between two or more objects. Directed graph logic is implemented inside our matching algorithm to check the correct direction of the rides.

Definition 2.2. (Directed graph [12]) *A directed graph D is an ordered triple (V(D), A(D), ψd) consisting of a nonempty set V(D) of vertices, a set A(D), disjoint from V(D), of arcs, and an incidence function ψD that associates with each arc of D an ordered pair of (not necessarily distinct) vertices of D. If a is an arc and u & υ are vertices such that ψD(a) = (u, υ), then a is said to join u to υ; u is the tail and υ is its head.*

Ignoring the directions between the nodes of a digraph can help us to extract the underlying graph that is the base of the digraph. This can be especially valuable if we want to apply undirected algorithms or just simplify our analysis. A digraph and its underlying graph can be seen in Figure 2.2 from [12].

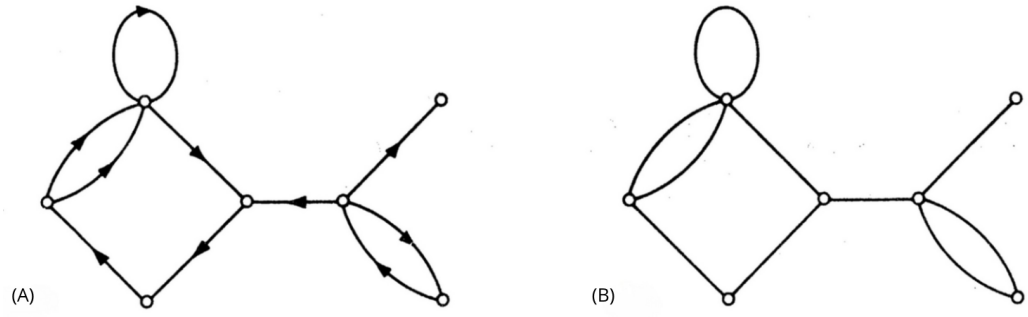


FIGURE 2.2: A digraph D in (A) and the underlying graph of D in (B).

There are various types of digraphs. A digraph without loops or parallel edges is called a simple digraph. A symmetric or bidirectional digraph is a digraph that for every edge (a, b) there is also an edge (b, a), or in simple terms every arc comes paired with its reverse. On the other hand, asymmetric digraphs can have loops but can have no more than one directed edge between a pair of vertices.

Another important type of graph is the connected graph. For example, in GPS applications, every stop or intersection is considered a node and the roads or tracks that connect them are the edges. Combining them, we have a connected graph in which users find their desired route and travel to their destination. In logistics connected graphs apply as well, with nodes being the warehouses and factories, while the shipping routes are considered the edges. By modeling the transportation process as a connected graph, we ensure that the supply chain does not have gaps and the product flow remains frictionless. In our case, connected graph logic is utilized when we check if the passenger's route is part of the sub-route created by the driver's starting and ending point (connectivity check within the induced subgraph).

Definition 2.3. (Connected graph [12]) *Two vertices u and υ of G are said to be connected if there is a (u, υ)-path in G. Connection is an equivalence relation on the vertex set V. Thus, these is a partition of V into nonempty subsets V1, V2, . . . , Vω such that two vertices u and υ are connected if and only if both u and υ belong to the same set Vi. The subgraphs G[V1], G[V2], . . . , G[Vω] are called the components of G. If G has exactly one component, G is connected; otherwise, G is disconnected. We denote the number of components of G by ω(G).*

Connected and disconnected graphs are shown in Figure 2.3 from [12].

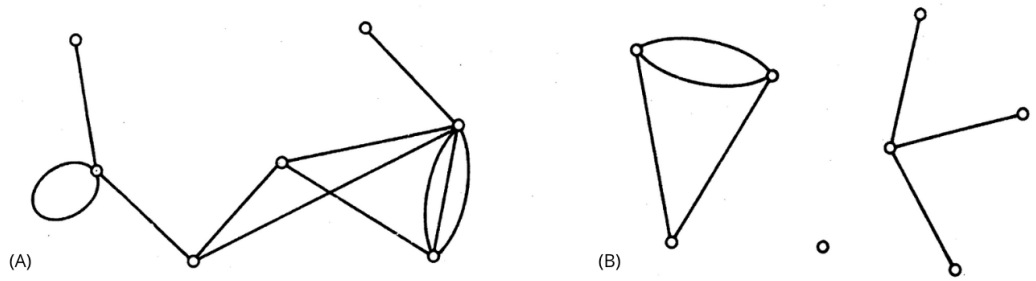


FIGURE 2.3: A connected graph in (A) and a disconnected graph with three components in (B).

One special type of graph is called a tree. A tree is a connected undirected graph that has no cycles, and thus any two of its vertices are connected by a unique path. Sometimes it is convenient to choose a vertex of a tree and call it the root node, then all the other nodes are called child nodes. Trees are used in many aspects of everyday life. For example, they can be used for efficient data insertion, deletion, and shorting. Examples of trees can be found in Figure 2.4 from [13].

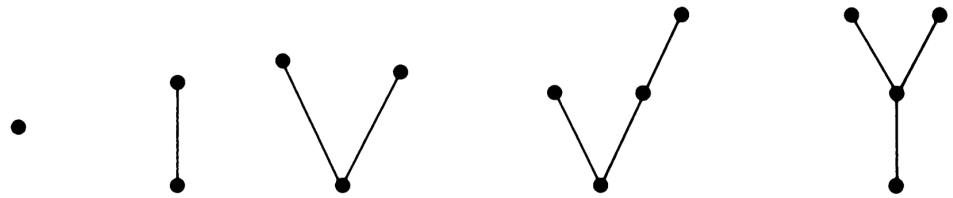


FIGURE 2.4: Trees with 1, 2, 3 and 4 vertices.

2.2 Coalition Formation and Cooperative Game Theory

Cooperative game theory gives us the theoretical background and theoretical tools to form coalitions [14]. It assumes agents are rational utility maximizers and the emphasis on the game-theoretic approaches is on sharing the payoffs among the agents that participate in the coalition. Having said that, there are many real-world scenarios that do not fully adhere to cooperative game theoretic assumptions. For instance, there are algorithms that try to heuristically form in the best possible way and assign members into coalitions to maximize the utility of the central design. This links to the problem of optimal coalition structure generation, which is to form the optimal partition of agents from the point of view of a central designer.

From a theoretical point of view, what distinguishes coalition formation from team formation is that the agents have to divide the value of the generated solution among themselves. They deliberate about who to join, what to do, and how to share the payoff. This is interesting from a theoretical point of view, but not a number-one consideration in many real-world scenarios. For instance, in our case it is not important at first to divide the value of the coalition. Of course, it would be of interest to compensate the driver for the cost that occurred during the ride. We will refer to that in future work. The terms team formation and coalition formation will be used in this thesis interchangeably.

In our work, we used coalition formation mechanics in the match-making algorithm to form groups of people that can be accommodated in the same vehicle and travel together on a compatible timeframe and route. There are many algorithms and protocols for coalition formation in various domains.

Shehory and Kraus in [15] present a Distributed Negotiation Protocol, Polynomial-Kernel-based Coalition Formation Mechanism (DNPK-CFM); these protocols can be used for coalition formation towards coalitional task execution. This specific protocol uses a convergence algorithm to get self-interested agents to join other coalitions so that they can increase their payoffs. Before negotiation, the agents calculate the values of coalitions of size K1 to K2. Then, all agents start the coalition procedure as single units. The proposal process begins with any of the coalition Cp choosing an associate Cr, only if $|Cp| + |Cr| \leq K2$. The combined value $V(C_{p+r})$ of the coalition is calculated and if the merging is beneficial, then a proposal is created that includes the new merged coalition, the new task redistribution, and the new payoff for the agents. Cr recalculates the proposal and if both are equal, then the two coalitions merge into C_{p+r} and the other coalitions are informed about the action. The process described above is repeated until no other proposal can be issued by the coalitions, meaning that a steady state is reached or until the agreed upon time ends. The protocol involves a second optional stage, where destruction proposals are allowed.

In the task execution domain, Sandholm et al. [16] showcase an algorithm for coalition structure generation that finds a coalition formation that is close to optimal and within a specific bound from it, only by searching a small subset and not all possible coalitions. This can be extremely beneficial when there are too many coalition structures to count and evaluate due to time and cost constraints. The algorithm searches the two bottom levels of the coalition structure graph; then if there is time left or if the entire graph has not been searched, it continues a breadth-first search from the top of the graph. Then the coalition structure with the highest value is returned. It is important to note how the bound k is computed. After the two bottom levels are searched, the total number of coalition structures searched is $n = 2^{a-1}$ and $k = a$. If there is more time or means to spare, then k is further reduced as more levels are searched from the top, using breath-first search.

Kraus et al. in [17] address coalition formation in the Request-For-Proposal (RFP) domain, where a requester issues a complex task divided into subtasks, and service providers join together to tackle it. The proposed coalition formation protocol helps agents negotiate and form coalitions by giving them two simple heuristics to select coalition partners. The "marginal" heuristic that guides the agent to choose a coalition with the maximum value, and the "expert" heuristic that guides the agent to search for tasks with a small number of competitors. The protocol consists of a central manager that supports two roles - the auctioneer and the coalition formation manager role - and agents that can participate. It is divided into rounds and ends when there are no other tasks to auction, no other agents participate, or the value of the remaining tasks is set to zero. At the start of the auction, the auctioneer announces the price $\mathrm{P}(T_i)$ for each task $T_i \in \Im$ that will be paid to the coalition that will perform T_i and the factor δ by which the prices of every unallocated task will be reduced in each round. For every task, the auctioneer verifies that the members of the proposing coalitions can execute it and assigns each task to the first qualified coalition. After a coalition is awarded a task, its members cannot break their contract and can only be paid $\mathrm{P}(T_i)\,\delta^r$ if they complete the entire task. The experimental evaluation showed that the 'marginal' heuristic is better when there is complete information and the 'expert' heuristic when the information is not complete. In both cases heuristics were valuable and stable.

Chalkiadakis in [18] presents both novel theoretical concepts and their practical applicability for coalitional task execution, e.g. in variants of the 'request for proposals' domain originally proposed by Kraus et al. in [17]. The CfP section describes how agents negotiate and propose to join or form coalitions. Any of the agents can become a proposer and propose changes to the existing coalition structure. The proposals include a new demand or a new action while staying in the same coalition, join another coalition with a new demand and a suggested action, or break away and form a new singleton. In order for the proposal to be accepted, all the members of the coalition compare the new payoff to the existing one and must agree to it; otherwise it is rejected. There are two versions of the protocol, the best reply (BR) where the proposer makes actions based on maximizing their payoff, and the best reply with experimentation (BRE) where their overall payoff is not increased; the latter happening so that coalitions move away from a stable state that is not always of the best structure.

At each round, the agents engage in the coalition formation process based on their current beliefs and execute agreed upon actions and observe the consequent result. The agents' beliefs are then updated. The process then repeats. The types of agents are four: The Non-myopic & Full Negotiation, the Myopic & Full Negotiation, the Non-myopic & One-Step Proposers and lastly the Myopic & One-Step Proposers. Myopic agents act only thinking

about the current payoff, while the Non-Myopic agents consider the long-term value of their actions. Full Negotiation agents negotiate continuously until the coalition structure becomes stable while the One-Shot Proposal describes the procedure where a random proposer makes a proposal which is accepted or rejected, and no other negotiation takes place after that.

Of course, there are many attempts at team formation and coalition formation in the ridesharing domain.

2.3 Related Work: Forming Coalitions in the Ridesharing Domain

Coalition formation in ridesharing refers to the process of dynamically creating groups of people that travel together with a common vehicle, and most of the time combine it with detour minimizing and fair cost distribution. This is the core problem that all researchers try to solve when it comes to addressing the ridesharing problem. There are many solution attempts in this field, some more or less practical and others more or less game theoretical.

Oregon Health & Science University (OHSU) in early 2024 released its official "OHSU Carpool" application. It is used by its employees, students, and other OHSU members in order to share rides to and from the university locations. Users can book their rides up to 14 days in advance, and the application will notify them if there is a match. Rides are free for the passengers; however drivers can earn incentives, which are limited to one ride per weekday, and are credited through their MyCommute account. The application is available in both Android and iOS through a third-party service provider, in contrast to TUC Ridesharing, which is fully built in-house.

University of Guelph in late 2024 announced their official commute app, the "U of Guelph Commute/UoGuelph Commute" application. Its a campus-exclusive transportation application to plan routes and share rides. The passengers enter their home and work/school addresses, are matched with commuters who have compatible schedules, and then they choose their preferred carpool partner. The application is free to use; the university mentions that users can split commuting costs, but to the best of our knowledge, there is no way to do that inside the application. As with OHSU Carpool and unlike TUC Ridesharing, a third-party service provider (Rideshark) is used to power this application.

Rubayath Alam et al. present a practical implementation in [19]. The authors created a web-based ridesharing platform for North South University (NSU) students, to help them travel to and from the campus in the city of Dhaka in Bangladesh. Users register using a valid NSU email address so that only valid members can have access. After logging in, users can either create a

ride or search for one. If a user finds a compatible ride, they send a join request and wait for approval. The system also supports notifications, so that users are notified about important events; as well as a payment mechanism. The ride can take place in a private vehicle where users decide the fare between them or in a public vehicle where they share it.

Another effort in building a working ridesharing solution is from Devansh Sanghvi [20]. Its aim is the creation of a web-based ridesharing application using various technologies like HTML5, CSS3, JavaScript, PHP, MySQL, AJAX and JSON. The user interface is straightforward and enables users to register and log in. After the user has logged in, they can use the website to search for a ride by entering the departure and destination locations. If there are available rides that match the criteria, they are displayed together with the driver's details. The passenger then contacts the driver to arrange the details of the ride. The driver on the other hand, can create a new ride by specifying the date, time, available seats, starting and ending points, and the contribution per seat that each passenger must pay to join the ride. There are no details about a testing phase, but the author stated that the site is operational with all its features being fully functional.

Bistaffa et al. in [6] acknowledge the significant benefits for communities and individuals who will adopt a ridesharing model for their daily transportation. At the same time, they understand that there is a lack of effective incentives policies by regulatory authorities who cannot assess the benefits and cost of such an action. The proposed solution is to create a real-time, scalable, peer-to-peer ridesharing algorithm that can be tuned to favor the environment or the quality of service, together with a comprehensive quantification of the environmental benefits. The system was proven to be capable of handling hundreds of trip requests issued by a large number of individuals in real-time, and group them in a very short amount of time in shared vehicles. The testing phase of this solution using real-world datasets showed extremely promising results by reducing CO_2 emissions by 70,78% and traffic congestion by 80.08% when the algorithm was tuned to maximize the environmental benefits.

Santi et al. in [21] propose a shareability network framework in order to balance the trade-offs between the benefits (travel costs, emissions, etc.) and the passenger discomfort (increased travel time). Essentially it interprets the ridesharing problem in a graph-theoretic context where every taxi is considered a node and each trip can be shared if there is a route that both: stops at all pickup and drop-off points in a way that respects the flow of the passengers' trips and also makes sure that all of them reach their destination by a specific time threshold compared to their solo trip duration. The framework was tested by applying it to 150 million rides, which roughly equates to one year of New York taxi trips. They discovered that in real-life scenarios, where all ride requests are grouped withing one-minute time windows and passengers are fine with an up to five-minute extra delay, around half of solo rides are converted to shared.

Which equates to a 25% drop in the total number of cars needed on the road.

From a more game-theoretic point of view, Bistaffa et al. in [22], propose an alternative solution to the social ridesharing problem, with users being members of a social network. The suggested way for forming efficient traveler coalitions that minimize the overall cost of the system is by modeling the formation problem as a Graph-Constrained Coalition Formation (GCCF) problem. That is, for a coalition to become viable, its members must induce a connected subgraph of the social network. This implementation also allows users to specify spatial and temporal preferences for their rides. In addition to this, a kernel-stable payment solution is introduced to achieve a fair distribution of the ride costs between the members. This approach achieved great cost reductions, up to -36,22% compared to the implementation of zero ridesharing and showed promising scalability with large numbers of agents.

To conclude this section, we refer to relevant diploma theses submitted at the Technical University of Crete under the supervision of Professor Georgios Chalkiadakis, which combine elements from graph theory, cooperative game theory, coalition formation theory, preference aggregation, and social choice.

Pagkalos in [4] acknowledges the low widespread adoption of ridesharing and suggests that the solution is making it more enjoyable than the competitive transportation solutions. Aside from only considering the route criteria between the coalition members, he proposes that ridesharing should take into account the preferences of the users, so that their travel time becomes more enjoyable. These preferences in the form of soft constraints are gender, age, and type of employment. The goal is to form feasible coalitions between agents, reduce driver detours, and satisfy their preferences in the best possible way. In the beginning, the process involves the use of hypergraphs to partition the city graph based on the initial route of the driver and the much needed initial clustering between the agents. Its vertices represent the passengers, and the hyperedges the cars provided by the drivers. Thus, the number of hyperedges is the same as the number of drivers. For each driver hyperedge, a branch-and-bound algorithm finds the subset of passengers that maximize the coalition value, and the highest-value coalition is selected with the use of a greedy algorithm. Then, for each vehicle, the system computes the optimal pickup and drop-off points with a branch-and-bound algorithm and calculates the overall cost of the trip. Driver compensation is derived from the total cost of the trip with the detours, minus the driving costs if they were driving on their own. Passengers split the remaining amount according to the distance they covered. The proposed solution was systematically evaluated using the graphs of the four cities of Crete, Chania, Rethymno, Heraklion, and Agios Nikolaos. It showed that it produces high-quality preference satisfaction coalitions and is exceptionally effective for small and middle-sized cities, while for larger cities the program needs more processing power to run. Pagkalos developed a framework for preferences-aware

ridesharing using Artificial Intelligence concepts and ran simulation experiments to evaluate it. However, compared to our work, his approach lacks a practical implementation that would make it easily accessible to users.

Asproudi [5] in her work, based on [4], developed a preference-aware game theoretic framework for social ridesharing with the purpose of maximizing the satisfaction of participants. When forming a coalition, aside from the hard constraints that must be met, there are some soft constraints in the form of attributes of each individual that the passengers they travel with ideally satisfy. These attributes are the gender, employment, and age range of the passengers. The driver has a service area which is a region around the planned route they would initially follow. All passengers who will be considered to enter the driver coalition must have their pickup and drop-off points in that area. Then the agent matching is carried out in two stages. First, for each passenger, a compatibility score is calculated, based on detour costs and the compatibility of passenger preferences with the driver; with the help of the Gale-Shapley algorithm. In the second stage, passengers are matched with each other through preference aggregation. This two-stage approach ensures that stable coalitions are formed. Then a combination of a branch-and-bound algorithm and the Dijkstra algorithm is used to find the optimal sequence of stops with the shortest path. Lastly, the kernel solution is implemented for the payment allocation. The experimental evaluation of this work in the city of Chania showed that there is high overall coalition satisfaction and efficient passenger distribution. One shortcoming was that the kernel transfer scheme was slow and resource intensive. Asproudi extends the work of Pangalos [4] by adding preference aggregation between the passengers, and conducted simulations to test the effectiveness of her system. Compared to our work, it lacks the deployment on an application that could host her system, and as a result of this, the real-world impact it could have on end users.

Chapter 3

Our Approach

In this chapter we detail our approach to solving the ridesharing problem within the Technical University of Crete (TUC) community. We start by presenting both the functional and non-functional requirements of the TUC Ridesharing application and the process by which we defined them. Then we present the basic use cases and our ridesharing protocol; finally, we present two usage examples that showcase the complete end-to-end ridesharing process.

3.1 Requirements Analysis of the Application

The first stage of designing our application was the identification of requirements. These became clear in a relatively short period of time after studying the literature [6, 18, 22] as well as through meetings with Professor Georgios Chalkiadakis and Assistant Professor Nikolaos Spanoudakis. The application should only be accessible to members of the Technical University of Crete, so we had to implement some form of *authentication*. After discussing with the institution's IT department, we concluded that the best way would be to implement the SSO Central Authentication Service (CAS) protocol. Secondly, the application should be on both ends *modular* and *as simple as possible*. Modularity was crucial to the design because this work is going to be the foundation of future work that will adjust and expand its features. That is why we implemented the Model-View-ViewModel (MVVM) architecture; in order to achieve clean separation of concerns. For the match-making algorithm, we chose to implement a simple two-stage greedy algorithm that would generate suggested rides for the passengers, who can then decide which to join. Users should coordinate in some way, so we implemented Firebase Cloud Messaging notifications that inform them about various events that take place during the ridesharing process. The crucial real-time data that must be delivered to the users are handled by WebSockets, with the help of the Publish/Subscribe messaging pattern. Specifically, Table 3.1 lists the *functional* and Table 3.2 the *non-functional* requirements of the system.

Authentication & modes	All users must be authenticated to enter the application. Upon successful login they must be able to choose their mode of operation.
Passenger request submission	A passenger can submit a new ride request by specifying: The pickup and drop-off points, the date and time of the pickup (at least 15 minutes in the future), and the number of seats needed.
Driver ride creation	A driver can create a ride by specifying: The starting and ending point of the ride, the date and time the ride starts (at least 15 minutes in the future), the vehicle's available seats and details (color, brand, model and license plate number).
Match-making algorithm	When a new ride is created, a new passenger request is submitted, a passenger joins a ride, or a passenger cancels a reservation, the system must generate a list of suggested rides which will be sent in real-time via WebSockets to the passengers. The passengers can then choose to join the suggested ride or ignore it.
Active rides	Passengers must be able to see in real-time all available rides.
Active requests	Drivers must be able to see in real-time all available passenger requests.
User list	Users should be able to view the list of users, with only minimal personal data displayed.
Real-time updates	The backend must send in real-time ride suggestions to the passengers. In the same way drivers must receive the details of the passengers that joined their ride.
User notifications	The notifications that are sent to the users must include the following types: Ride request submission, ride request cancelation, reservation cancelation, ride completion, new ride suggestion, ride unavailability, ride start, request expiration, ride creation, ride cancelation, new passenger join, passenger reservation cancelation, ride approaching, ride start reminder, ride auto-cancelation, ride completion reminder and ride autocompletion.

Table 3.1: Functional requirements.

Architecture of the frontend	The application must be modular to facilitate maintenance and future expansion.
Application & data security	Only members of the TUC shall be able to access the application using their institutional credentials so that there is an enhanced feeling of safety when using the application. All traffic must be secured via HTTPS so that the data that is transferred to and from the server is secure.
Supported operating systems	The application must be multi-platform and support as many devices as possible, so that all members of the community can use it.
Supported OS versions	The supported OS versions must be recent so that we can implement the latest documentation solutions without at the same time, maintain obsolete ones.
User interface layout	The user interface must be modern and simple because it mainly targets young people who are keener to use an application that is visually appealing and efficient. The UI on both platforms must be locked to portrait orientation, as in most modern ridesharing applications.
User feedback	The application should utilize a WebView to display a feedback form and collect user feedback in the testing phase of the application. After testing is completed, it should be repurposed to navigate the users to the store listings, where they can submit reviews to further improve the system.
Backend hosting	The backend must be hosted on the Technical University of Crete's servers to enable institutional authentication.
Service availability	The service must be available 24/7/365 to the users, because application usage cannot be restricted to the academic calendar and hours.

Table 3.2: Non-functional requirements.

3.2 Basic Use Cases

In this section, we present the basic use cases of our application. We show how the application behaves in specific scenarios and what the user should expect when using it. The basic use cases are: user login, join a driver's ride, and offer a ride to a passenger.

3.2.1 User login

Actor: A TUC member - a person who owns valid TUC credentials.
Use case: TUC member login.

- The TUC member enters their credentials in the login form.
- The application backend communicates with the IdP provider to validate the data submitted.
- The TUC member is redirected inside the application.

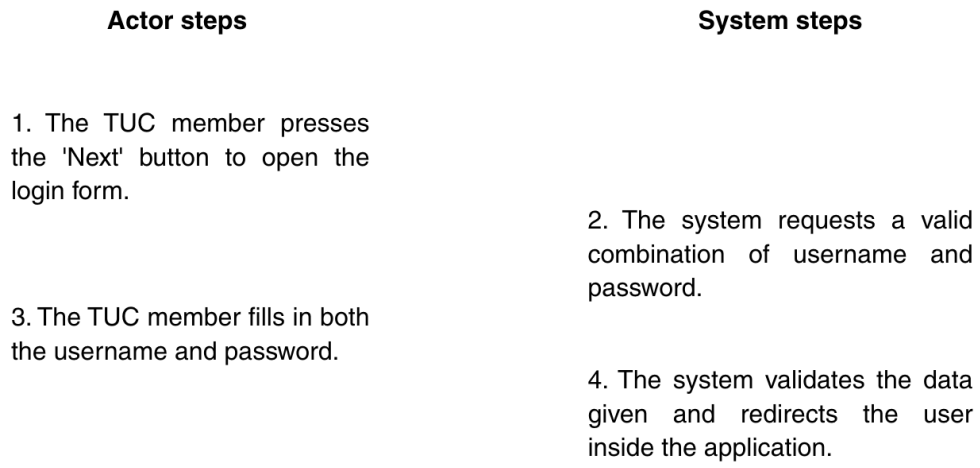


Figure 3.1: Use case 1 flow of events.

Use case one: Exceptions.

- The TUC member fills in both the username and password.
  [**Invalid Credentials**]

  [**Invalid Credentials**] In case of invalid credentials, the system clears both fields and informs the user about the reason the login process did not proceed.

3.2.2 Join a driver's ride

Actor: A TUC member - a person who owns valid TUC credentials.
Use case: A TUC member joins a ride.

- The TUC member presses the **Passenger Mode** button within the application, enters passenger mode, and fills in the details of the request.
- The system processes the request and returns suggested rides to the passenger.
- The TUC member selects and joins the ride of their liking.

Actor steps	System steps
1. The TUC member presses the 'Passenger Mode' button inside the application to enter passenger mode.	
	2. The system navigates them to the Passenger Main screen.
3. The TUC member fills in the pickup point, drop-off point, seats needed, adjusts the date and time and presses the 'Submit the request' button.	
	4. The system validates, stores the data and generates suggested rides that are sent back to the passenger.
5. The TUC member selects the ride of their liking, and joins it by pressing the 'Join' button of that particular ride.	
	6. The system confirms the join request and returns the ride data to the passenger.

Figure 3.2: Use case 2 flow of events.

Use case two: Exceptions.

- The TUC member fills in the pickup point, drop-off point, seats needed, adjusts the date and time, and presses the **Submit the request** button. [**Empty Fields**] [**Date & Time Combination**]

  [**Empty Fields**] In case of empty fields, the system informs the user with an error dialog and highlights the empty fields in red.

  [**Date & Time Combination**] When the date and time combination is not at least 15 minutes in the future, the date and time fields are highlighted in red and the system shows an error dialog explaining the issue.

- The system validates, stores the data and generates suggested rides that are sent back to the passenger. [**No compatible rides**]

  [**No compatible rides**] When there are no compatible rides for a request, the passenger does not receive any suggested rides back.

- The TUC member selects the ride of their liking and joins it by pressing the **Join** button of that ride. [**Seats filled before joining**]

  [**Seats filled before joining**] A suggested ride is visible to many compatible passengers at the same time. If some users allocate a certain number of seats, the **Join** button becomes disabled for those whose requests exceed the remaining available seats. This prevents overbooking.

- The system confirms the join request and returns the ride data to the passenger. [**Seats filled after pressing the Join button**]

  [**Seats filled after pressing the Join button**] Sometimes passengers might press the **Join** button of a suggested ride almost simultaneously. For that specific scenario where the seats from both requests cannot be served from the available seats of the specific ride, the user that pressed the button first enters the ride, and the other sees an error message that informs them about the seat unavailability.

3.2.3 Offer a ride to a passenger

Actor: A TUC member - a person who owns valid TUC credentials.
Use case: A TUC member offers a ride.

- The TUC member presses the **Driver Mode** button to enter driver mode and fills in the details to create the ride.
- The system processes the new ride and returns it as a suggested ride to passengers who made compatible requests.
- The TUC member receives the information of the passengers who joined the ride.

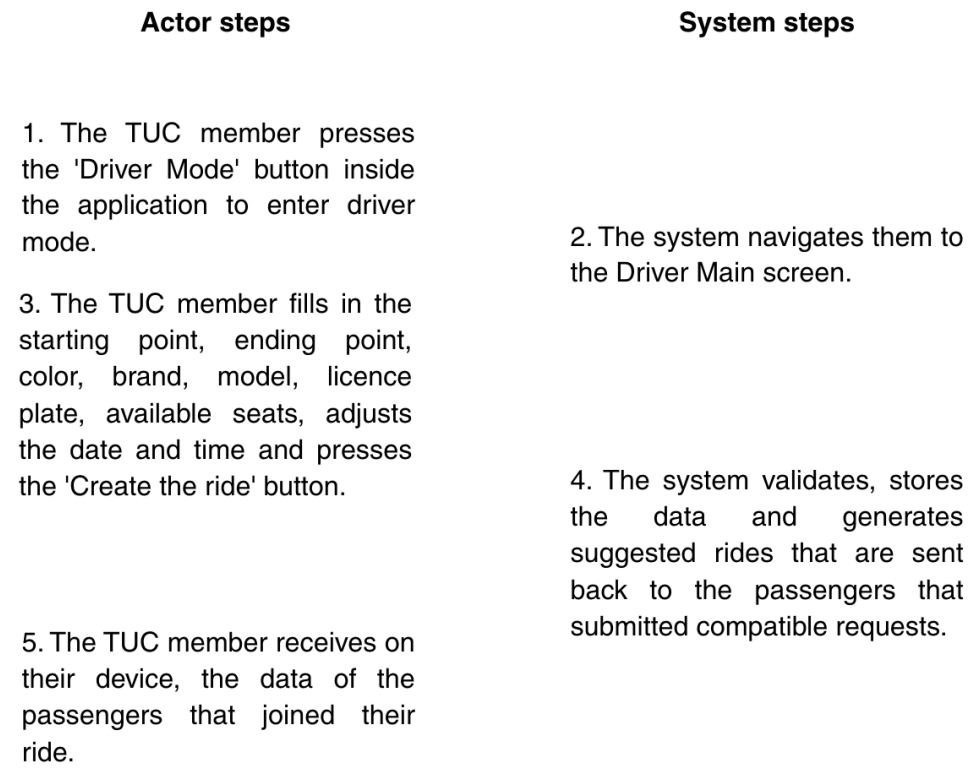


FIGURE 3.3: Use case 3 flow of events.

Use case three: Exceptions.

- The TUC member fills in the starting point, ending point, color, brand, model, available seats, adjusts the date and time, and presses the **Create the ride** button. [**Empty Fields**] [**Date & Time Combination**]

  [**Empty Fields**] In case of empty fields, the system informs the user with an error dialog and highlights the empty fields in red.

  [**Date & Time Combination**] When the date and time combination is not at least 15 minutes in the future, the date and time fields are highlighted in red and the system shows an error dialog explaining the issue.

- The system validates, stores the data, and generates suggested rides that are sent back to passengers that submitted compatible requests.
  [**No compatible requests**]

  [**No compatible requests**] When there are no compatible requests, the ride cannot be paired and shown as a suggested ride.

- The TUC member receives on their device the data of the passengers that joined their ride. [**Passengers did not join the ride**]

  [**Passengers did not join the ride**] When passengers do not join a ride, the driver does not receive passenger data on their device.

3.3 Ridesharing Protocol

Our application includes a ridesharing protocol that can be seen in Figure 3.4. It comprises the set of rules that govern how passengers and drivers communicate, match, and eventually travel together. For the sake of simplicity, we will separate our protocol into three distinct parts. The pre-matching phase, the matching phase, and the coalition formation and travel phase.

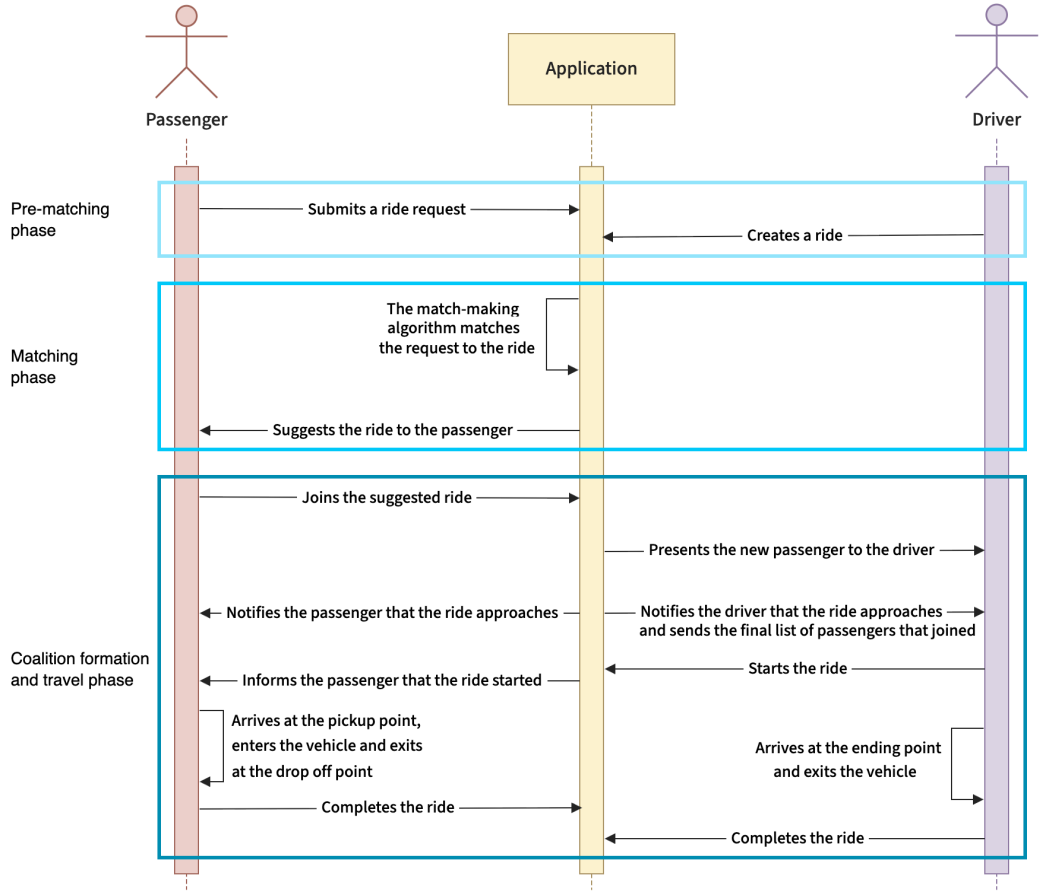


FIGURE 3.4: Ridesharing protocol sequence diagram.

3.3.1 Pre-matching phase

The user, after entering passenger mode, selects their pickup and drop-off points from the predefined lists of locations. Then they adjust the date and time with the help of the integrated date and time pickers and specify how many seats they need because the application supports multiple seat allocation per request. In the end, they press the **Submit the request** button so that the request can be sent to the backend. Similarly, the driver enters driver mode and selects

their starting and ending points from the predefined lists of locations. Then they adjust the date and time with the help of the integrated date and time pickers, specify the color, brand, model, license plate number, and available seats of the vehicle. In the end, they press the **Create the ride** button so that the new ride can be sent to the backend.

3.3.2 Matching phase

Following the previous phase, the data is now stored in the requested_rides and offered_rides tables and the match-making algorithm that handles all the matching logic of the application is triggered. The algorithm 3.1 that we put forward is responsible for generating suggested rides for passengers to join, and is invoked every time a ride is created, a ride request is submitted, when a passenger cancels their reservation and when a passenger joins a ride. The algorithm runs in all these cases to ensure that the data displayed on the users' screens is always accurate. These suggestions are the output of the algorithm 3.1.

Each time the algorithm is triggered, it scans the two tables that hold the passenger requests and the driver rides, to find matches. For every passenger request, it checks if the time criteria are met, which means that the driver's start time must be in the time window created by the passenger's pickup time plus one hour. If this constraint is met, then it proceeds to check if there is a spatial match with that driver. First, it looks for a direct match inside the 20 predefined routes that we describe in Subsection 3.3.2.1, which means that the passenger's travel route must be included in the driver's route and the vehicle's available seats must be sufficient. If so, a match is created, is saved in the suggested_rides table, and is sent to the passenger together with a notification to inform them about the event.

Aside from the above direct matching of the algorithm, another matching stage is implemented that tries to make indirect matches between the drivers and the passengers, meaning that rides and requests that head to the same destination can match, even though they belong to different lines. We implemented this because occasionally, rides that match the exact passenger's criteria may not exist, but there may be others that are 'good enough' and can transport them to their destination; with the trade-off that their pickup or drop-off point will be altered to match the driver's. For example, if there is a passenger that wants to travel from the Campus A2 Amphitheater to the National Stadium at 15:00, and there is an available ride from the Campus Library to 1866 square at 15.30, the indirect suggestion would be generated, and the passenger would be able to join that ride just by walking a short distance to the Campus Library point, which is a small trade-off. A similar process occurs when the request and the ride are directed to Campus, with the

difference that the drop-off point of the passenger would be altered to match the driver's ending point.

This procedure continues for every passenger and driver combination that is available at that time, and up to three (3) suggestions can be sent to each passenger together with a notification.

3.3.2.1 A match-making algorithm for our Setting

The match-making algorithm summarized in 3.1 is the core of the ridesharing process and of the application as a whole. The type we chose to use is a two-stage greedy one. We started designing it by creating the city graphs on which the whole algorithm is based. The first step was to find a way to separate the points of interest inside Campus, so that drivers do not have to take detours. The approach we chose was to split the points into two lines. Line one consists of the Campus Electrical & Computer Engineering Building point, Campus Student Residence point, Campus Athletic Fields point, Campus A2 Amphitheater point, and the Campus Main Gate point. Line two consists of the Campus Computer Lab point, the Campus Mineral Resources Engineering Department Building point, Campus Restaurant point, Campus Library point, and the Campus Main Gate point, the last of which is shared with line one.

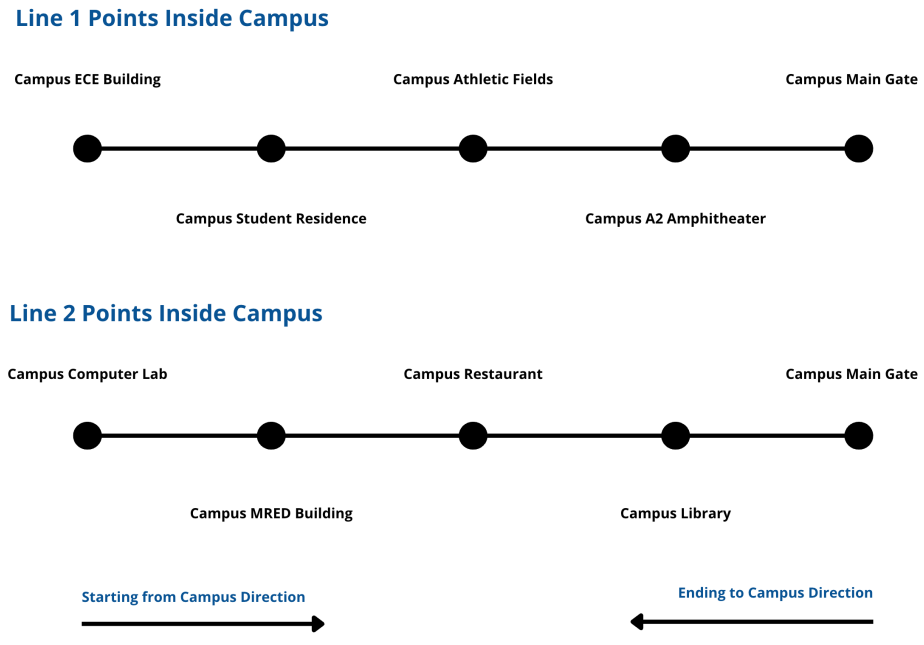


FIGURE 3.5: Line 1 and line 2 points of interest inside Campus.

Algorithm 3.1 Match-making algorithm

```
1: Input
2: routes ← predefined paths (List[Route], case-insensitive trimmed matching)
3: requestedRides ← all ride requests with status = AVAILABLE (List[RequestedRide])
4: offeredRides ← all ride offers with status = AVAILABLE (List[OfferedRide])
5:
6: Output
7: suggestions ← list of final suggestions (List[RequestID, OfferID)])
8:
9: Variables
10: cutoff ← maximum allowed start time for a request (Time)
11: matchType ← direct or indirect suggestion type (String)
12: indirectType ← fromCampus / toCampus / null (String or Null)
13: seats ← vehicle's available seats (Integer)
14: seatsNeeded ← requested seats (Integer)
15:

16: function GENERATE_SUGGESTIONS
17:     requestedRides ← fetchAvailableRequestedRides()
18:     offeredRides ← fetchAvailableOfferedRides()
19:     suggestions ← []
20:     mergedRoutes ← pairs(routes, i ↔ i+10: "starting" for i ∈ [0..4], "ending" for i ∈ [5..9])
21:     for all request in requestedRides do
22:         cutoff ← (request.pickupDate for + request.pickupTime) + 1 hour
23:         for all offer in offeredRides do
24:             if offer.startDateTime ∉ [request.pickupDateTime, cutoff]
25:                 continue
26:             end if
27:             if directMatch(request, offer, routes) and seats ≥ seatsNeeded
28:                 matchType ← "direct"
29:                 indirectType ← null
30:             else if indirectMatch(request, offer, mergedRoutes) and seats ≥ seatsNeeded
31:                 matchType ← "indirect"
32:                 indirectType ← fromCampus / toCampus / null
33:             else
34:                 continue
35:             end if
36:             if !duplicate(request, offer)
37:                 createSuggestion(request, offer, matchType, indirectType)
38:             end if
39:         end for
40:     end for
41:
42:     return suggestions
43: end function
```

The second step in designing the algorithm was to determine the routes that we will incorporate into our algorithm. We wanted to include as many popular student points and destinations as possible, but not include too many to the point that it would complicate our design too much. The five routes we chose have the following destinations: **City Center**, **Kounoupidiana**, **Eleftherias Square**, **Koum Kapi**, and **Souda**. Combining these routes with the two lines inside the Campus, we ended up with the **20 final routes**. Five line one starting from Campus, five line one ending to Campus, five line two starting from Campus, and five line two ending to Campus. The complete route list can be seen below.

Campus to City Center	Campus ECE Building → Campus Student Residence → Campus Athletic Fields → Campus A2 Amphitheater → Campus Main Gate → Chalkiadakis S/M Akrotiriou → Attikon Summer Cinema → National Stadium → Old Chania Market → 1866 Square
Campus to Kounoup.	Campus ECE Building → Campus Student Residence → Campus Athletic Fields → Campus A2 Amphitheater → Campus Main Gate → Chalkiadakis MAX S/M → Gourounakia Restaurant → Pancreta Bank → Aristotelous → Iridos
Campus to Eleftherias Square	Campus ECE Building → Campus Student Residence → Campus Athletic Fields → Campus A2 Amphitheater → Campus Main Gate → Chalkiadakis S/M Akrotiriou → Attikon Summer Cinema → Iroon Polytechneiou → Eleftherias Square
Campus to Koum Kapi	Campus ECE Building → Campus Student Residence → Campus Athletic Fields → Campus A2 Amphitheater → Campus Main Gate → Chalkiadakis S/M Akrotiriou → Attikon Summer Cinema → National Stadium → Splatzia Square → Koum Kapi
Campus to Souda	Campus ECE Building → Campus Student Residence → Campus Athletic Fields → Campus A2 Amphitheater → Campus Main Gate → Nykterida Restaurant → Mega Place → Kato Souda → Souda Square

TABLE 3.3: Line 1 routes starting from Campus.

City Center to Campus	City Hall → National Bank → National Stadium → Attikon Summer Cinema → Chalkiadakis S/M Akrotiriou → Campus Main Gate → Campus A2 Amphitheater → Campus Athletic Fields → Campus Student Residence → Campus ECE Building
Kounoup. to Campus	Iridos → Aristotelous → Pancreta Bank → Gourounakia Restaurant → Chalkiadakis MAX S/M → Campus Main Gate → Campus A2 Amphitheater → Campus Athletic Fields → Campus Student Residence → Campus ECE Building
Eleftherias Square to Campus	Eleftherias Square → Iroon Polytechneiou → Attikon Summer Cinema → Chalkiadakis S/M Akrotiriou → Campus Main Gate → Campus A2 Amphitheater → Campus Athletic Fields → Campus Student Residence → Campus ECE Building
Koum Kapi to Campus	Koum Kapi → National Stadium → Attikon Summer Cinema → Chalkiadakis S/M Akrotiriou → Campus Main Gate → Campus A2 Amphitheater → Campus Athletic Fields → Campus Student Residence → Campus ECE Building
Souda to Campus	Souda Square → Kato Souda → Mega Place → Nykterida Restaurant → Campus Main Gate → Campus A2 Amphitheater → Campus Athletic Fields → Campus Student Residence → Campus ECE Building

TABLE 3.4: Line 1 routes ending to Campus.

Campus to City Center	Campus Computer Lab → Campus MRED Building → Campus Restaurant → Campus Library → Campus Main Gate → Chalkiadakis S/M Akrotiriou → Attikon Summer Cinema → National Stadium → Old Chania Market → 1866 Square
Campus to Kounoup.	Campus Computer Lab → Campus MRED Building → Campus Restaurant → Campus Library → Campus Main Gate → Chalkiadakis MAX S/M → Gourounakia Restaurant → Pancreta Bank → Aristotelous → Iridos
Campus to Eleftherias Square	Campus Computer Lab → Campus MRED Building → Campus Restaurant → Campus Library → Campus Main Gate → Chalkiadakis S/M Akrotiriou → Attikon Summer Cinema → Iroon Polytechneiou → Eleftherias Square
Campus to Koum Kapi	Campus Computer Lab → Campus MRED Building → Campus Restaurant → Campus Library → Campus Main Gate → Chalkiadakis S/M Akrotiriou → Attikon Summer Cinema → National Stadium → Splatzia Square → Koum Kapi
Campus to Souda	Campus Computer Lab → Campus MRED Building → Campus Restaurant → Campus Library → Campus Main Gate → Nykterida Restaurant → Mega Place → Kato Souda → Souda Square

TABLE 3.5: Line 2 routes starting from Campus.

City Center to Campus	City Hall → National Bank → National Stadium → Attikon Summer Cinema → Chalkiadakis S/M Akrotiriou → Campus Main Gate → Campus Library → Campus Restaurant → Campus MRED Building → Campus Computer Lab
Kounoup. to Campus	Iridos → Aristotelous → Pancreta Bank → Gourounakia Restaurant → Chalkiadakis MAX S/M → Campus Main Gate → Campus Library → Campus Restaurant → Campus MRED Building → Campus Computer Lab
Eleftherias Square to Campus	Eleftherias Square → Iroon Polytechneiou → Attikon Summer Cinema → Chalkiadakis S/M Akrotiriou → Campus Main Gate → Campus Library → Campus Restaurant → Campus MRED Building → Campus Computer Lab
Koum Kapi to Campus	Koum Kapi → National Stadium → Attikon Summer Cinema → Chalkiadakis S/M Akrotiriou → Campus Main Gate → Campus Library → Campus Restaurant → Campus MRED Building → Campus Computer Lab
Souda to Campus	Souda Square → Kato Souda → Mega Place → Nykterida Restaurant → Campus Main Gate → Campus Library → Campus Restaurant → Campus MRED Building → Campus Computer Lab

Table 3.6: Line 2 routes ending to Campus.

At this point, we were able to match passengers and drivers traveling in the same direction to all the points inside and outside the Campus. The issue with this implementation was that for example, if a passenger wanted a ride from the Campus Computer Lab, which is in Line two, to the 1866 square and there was a ride that was starting from the Campus Student Residence and was ending to the 1866 square, our logic would not create a suggestion; even though the pickup point and the starting point are inside the Campus and the direction of the vehicle is compatible with the route of the passenger. The same issue arises when the pickup point and the starting point are outside the Campus and the drop-off point and the ending point are inside the Campus.

To solve this, we implemented cross-line suggestions, meaning that passengers can receive suggestions for their request even though the lines do not match. This scenario obviously requires that the time and seat criteria are met. In this situation, what we do is this: If the passenger's ending point is inside the Campus, we alter it to match the driver's ending point, and if the

passenger's pickup point is inside the Campus, we alter it to match the driver's starting point. The passenger is informed about the change in the pickup or drop-off point, in the dialog that appears when they press the **Join** button in the cross-line suggestion card. In both cases, we do it because the application currently does not have incentives for the drivers and the least we could do was to not allow detours. Basically, we want to encourage drivers to create new rides, knowing that they will not change their initial solo route if they share their ride, they will just help fellow members of the community arrive to their destination. The end-to-end description of the algorithm workflow is as follows:

- In the beginning, it scans both requested_rides and offered_rides tables and finds all rows where the status is `AVAILABLE`. We do this so that we can filter out all irrelevant data.
- Right after we merge the line one and line two starting from Campus routes, and we do the same with the line one and line two routes ending to Campus. By doing this, we create **10 merged pair routes** that we use to handle cross-line matching.
- Then we loop over the passenger requests, we normalize to lowercase the pickup and drop-off points of each passenger, we trim the whitespace and compute the one-hour time window in which the ride must start. The maximum allowed ride start time is the pickup time plus one hour.
- Then we move to the ride offers, and for each we normalize to lowercase their starting and ending point and trim the whitespace. Immediately after, we loop through the driver offers for each request, and we check if the time windows are compatible. The acceptable start time must be no more than the pickup time plus one hour. If it is not within the acceptable limits, then we move to the next offer. If the time constraint is met, then we proceed below to the matching logic.

The suggestions can be generated in two ways:

The first is the direct way, where the algorithm looks for a linear substring from the driver's starting point to the driver's ending point from the list of the 20 predefined routes, which also includes the passenger's pickup point and ending point in the correct order in which they exist on the route. In short, the passenger's route must be included in the driver's route. We also check if there are enough seats to accommodate the passenger request, if the seats are not enough, then we skip that specific ride and we continue with the next one.

The second way is the indirect way. If the direct matching above fails, then we try to create a cross-line matching. The algorithm evaluates the 10 merged routes and starts checking if on any of these routes the driver's starting point, driver's ending point, passenger's pickup point, and passenger's drop-off point are all present. The Campus points here are grouped together, so their order does not matter, but the stops outside the Campus must be in a specific order as in the direct routes. In short:

- If the route‘s direction is from inside Campus to destinations outside, then the match can occur even if the passenger's starting point is from the other line. For example, if the driver's starting point is the Campus A2 Amphitheater and the passenger's pickup point is the Campus Library which is a line two point. In this scenario, the suggestion is generated, but the passenger is informed that to join the ride, their pickup point must be altered to match the driver's starting point.
- If the route's direction is from locations outside the Campus to locations inside, then the match can also happen despite the passenger's drop-off point being from the other line. For example, if the driver's ending point is the Campus Student Residence and the passenger's drop-off point is the Campus Computer Lab which is a line two point. The suggestion in this case is also generated, but the passenger is informed that to join the ride, their drop-off point must be altered to match the driver's ending point.

In both cases, we check if there are enough seats to accommodate the passenger request. If the seats are not enough, then we skip that specific ride, and we continue with the next. The main idea of cross-line matching is that passengers can trade off a bit of convenience, i.e. walk a really small percentage of the whole ride, to travel by car the rest of it. This is by far a better option than having no ride at all. Passengers receive both types of suggestions (direct & indirect) and can decide which fits them best.

After a suggestion is generated, we check if there is a duplicate in our suggested_rides table and if the suggestion for that particular request is at most the third. If so, we insert the new row to the suggested_rides table. If the suggestion is indirect, we mark it as one. We also save the direction or the ride, whether it is `from_Campus` or `to_Campus`. We do this for the **Join** function to know which passenger point (pickup or drop-off) to alter and match the driver's. At the same time, the suggestion is published through Publish/Subscribe and sent over by the suggestions WebSocket, together with an FCM[1] notification, to inform the passenger.

At this point, the algorithm's work is complete. But there is one more step for the passenger to join the ride. After the suggested ride card is displayed

[1]Firebase Cloud Messaging (FCM) is Google's push notification service that is used to deliver push messages to different platforms (Android, iOS, Web).

on their screen, they have the option to join or ignore the ride. If they opt to join, then a new row is inserted in the matched ride table, and the passenger data is sent to the driver together with an FCM notification. The passenger's screen is cleared of all suggested ride cards and the requested ride card. A new ride summary card appears that contains the summary of the trip with all relevant locations, together with all the driver and vehicle information. In this way, they can comfortably recognize the vehicle when it approaches the pickup point and have a clear view of their trip. Similarly, the driver ride summary card appears on the driver's screen 12 minutes before the departure, showing the points where they have to pick up and drop off passengers, together with their personal information.

3.3.3 Coalition formation and travel phase

This phase starts when the passenger presses the **Join** button on a suggested ride card that was generated at the end of the previous phase. By doing so, the driver receives the data of the passenger that joined the ride so that they know who the new passenger is, and their pickup and drop-off points. Up to about 12 minutes before the departure, new passengers can enter the ride if the vehicle capacity allows. At this point, the ride locks. The backend sends a notification to inform all members of the coalition that the ride is approaching and to get ready. This helps coordinate the people involved even if they did not pay attention to the ride's start time or the application. The driver receives an additional data-only notification with the final list of passengers that they will travel with. Then on the Driver Main screen all cards are cleared and a new driver summary card appears that contains a summary of the trip's stops and the data of all passengers. A different notification is sent to the passengers that had the suggested ride on their screens but did not press the **Join** button in time and missed the ride, telling them that the ride is unavailable and the suggestion is cleared from their screen.

The driver should at this point head to the vehicle and when they are ready to depart, they should press the **Start** button on their driver summary card. This action triggers a notification that is sent to all passengers and informs them about the event so they can even better schedule their pickup. After this point, the process includes the commuting from every point of the predefined locations to the other and the pickup and drop-off of the passengers. Each passenger after exiting the vehicle just has to press the **Complete** button on their ride summary card in order to reset the application UI to its initial state and register the ride as completed on the backend. As soon as they do, they receive a notification that informs them that their request was completed. Similarly, when the driver arrives at the ending point and exits the vehicle, when they press the **Complete** button, they receive the same notification, and their UI resets and is ready for the next ride.

At any given point in the process, both the passenger and the driver can cancel their request or ride. If a passenger did not join a ride and canceled their request, the ride is registered as canceled, the UI is reset to its initial state, and the WebSocket closes. If they have already joined a ride, then all the above actions occur, and at the same time a notification is sent to the driver to inform them about the reservation cancelation. Additionally, if the cancelation occurs before the ride locks, then the passenger's card is cleared from the driver's screen and is not included in the summary.

When the driver cancels, if no one has joined their ride, the UI is reset to its initial state, the WebSocket closes, the ride is registered as canceled, and they receive a notification that informs them about the successful cancelation. On the other hand, if there were passengers who joined the ride before the cancelation, they are notified, and are prompted to cancel their reservation and search for another ride.

There is also a reminder mechanism for the driver where if they forget to press the **Start** button, five minutes after the ride start time, they receive a reminder notification. If they fail to press the **Start** button 15 minutes after the scheduled ride start time, the ride is automatically canceled, and all members involved are notified.

One hour after the ride start, all members are checked if they canceled or completed their ride. If neither of these occurred, then a notification is sent to prompt them to complete the ride. Fifteen minutes after that notification, an additional check is made. If the status of the ride/requests is still not `CANCELED` or `COMPLETED`, it is updated to `COMPLETED`, and the affected users receive a notification which informs them about the action and resets their application UI.

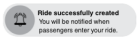
This protocol combines simplicity and flexibility by including the most popular routes and points inside and outside the Campus while at the same time it does not complicate things by requesting GPS locations or any other data from the users. It aims to make transportation among the members of the Technical University of Crete straightforward and not overly complicated. The final goal is to deliver a reliable utility application that performs consistently as intended.

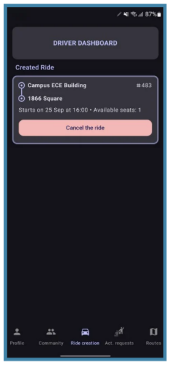
3.4 Usage Examples

In this section, we will present two usage scenarios that include common usage patterns of the application. The first scenario is the simplest possible and is used to demonstrate the most common use of the application. It involves a newly created ride and the transportation of a passenger to their destination. The driver's phone is in dark mode, and the texts that refer to them are in blue color.

A student of the TUC, at 15:21 on 25.09.2025, creates a ride that starts from the Campus ECE Building and ends to 1866 Square with a starting time of 16:00 in order to serve his fellow students who may need a ride.

The route details are passed to the created ride card and he receives a notification confirming that his ride was created successfully, indicating that the process is proceeding normally.



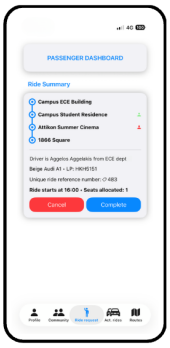


On the same day, another student of the TUC, searches for a ride from the Campus Student Residence to the Attikon Summer Cinema with a pickup time of 15:57 and makes the request.

The route details are passed to the requested ride card and he receives a notification confirming that his request was created successfully, reassuring him that the process is progressing normally for him as well.

The algorithm verifies that the time, location, and capacity criteria match the passenger's request, and generates a suggested ride. The passenger is notified of the new suggestion and taps 'Join' to enter the ride.

All the cards on the passenger's screen disappear and a new card is created that contains the ride summary. The passenger still has the option to cancel his ride.



At the same time, the passenger's card appears on the driver's screen, accompanied by a notification.

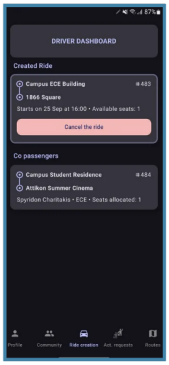


FIGURE 3.6: Usage scenario 1, part 1 of 2.

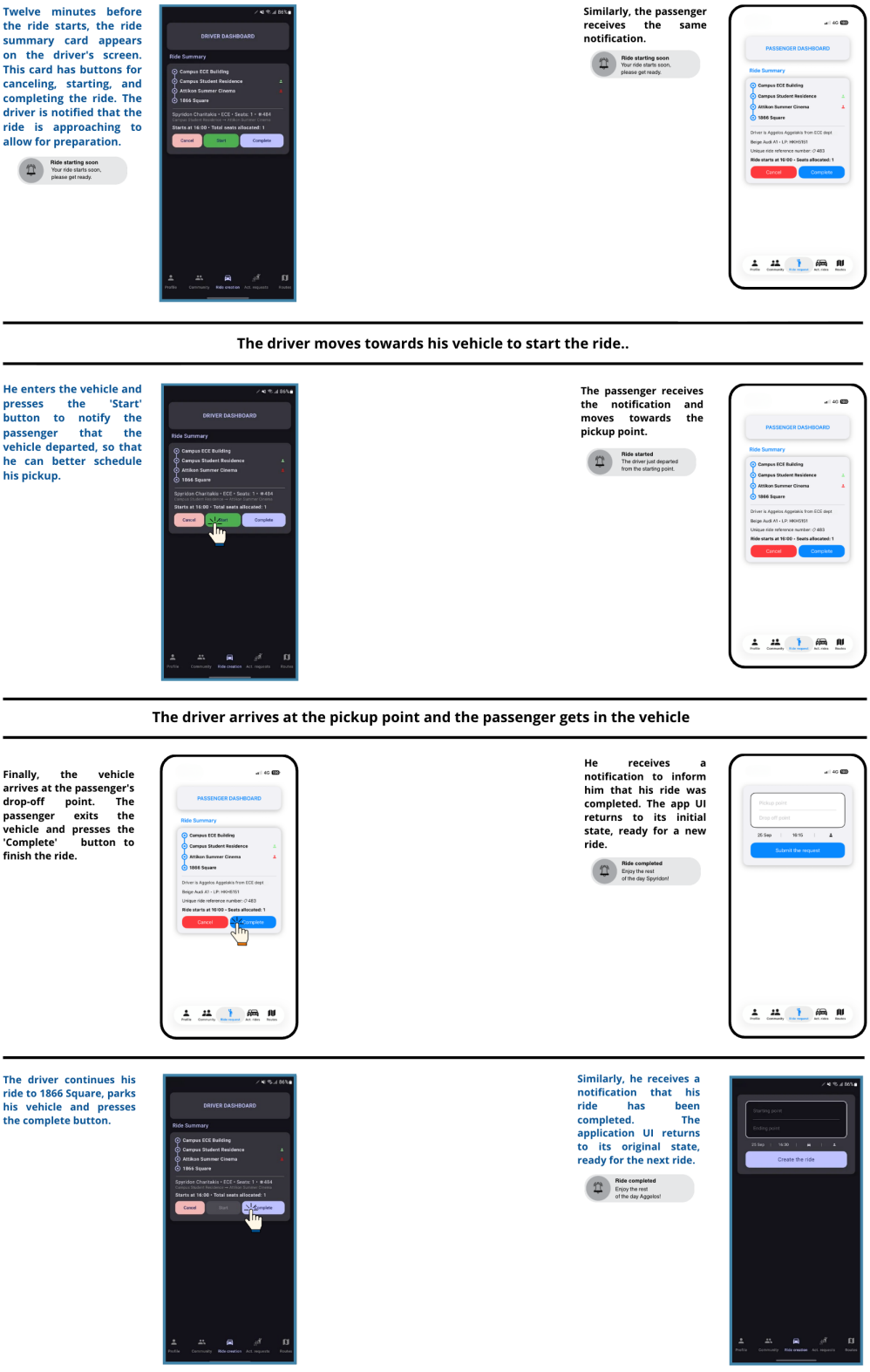


FIGURE 3.7: Usage scenario 1, part 2 of 2.

The second scenario is more complex, involving more people, a cancelation, real-time screen updates, and cross-line matches. In the beginning a suggested ride is shown on the screens of two different people who want to book a ride. The first makes the reservation for them and three of their friends and after they cancel it, the ride becomes available again for the second user and their two friends. Then they book the ride and travel together to their destination. The driver's phone is in dark mode, and the texts that refer to them are in blue color.

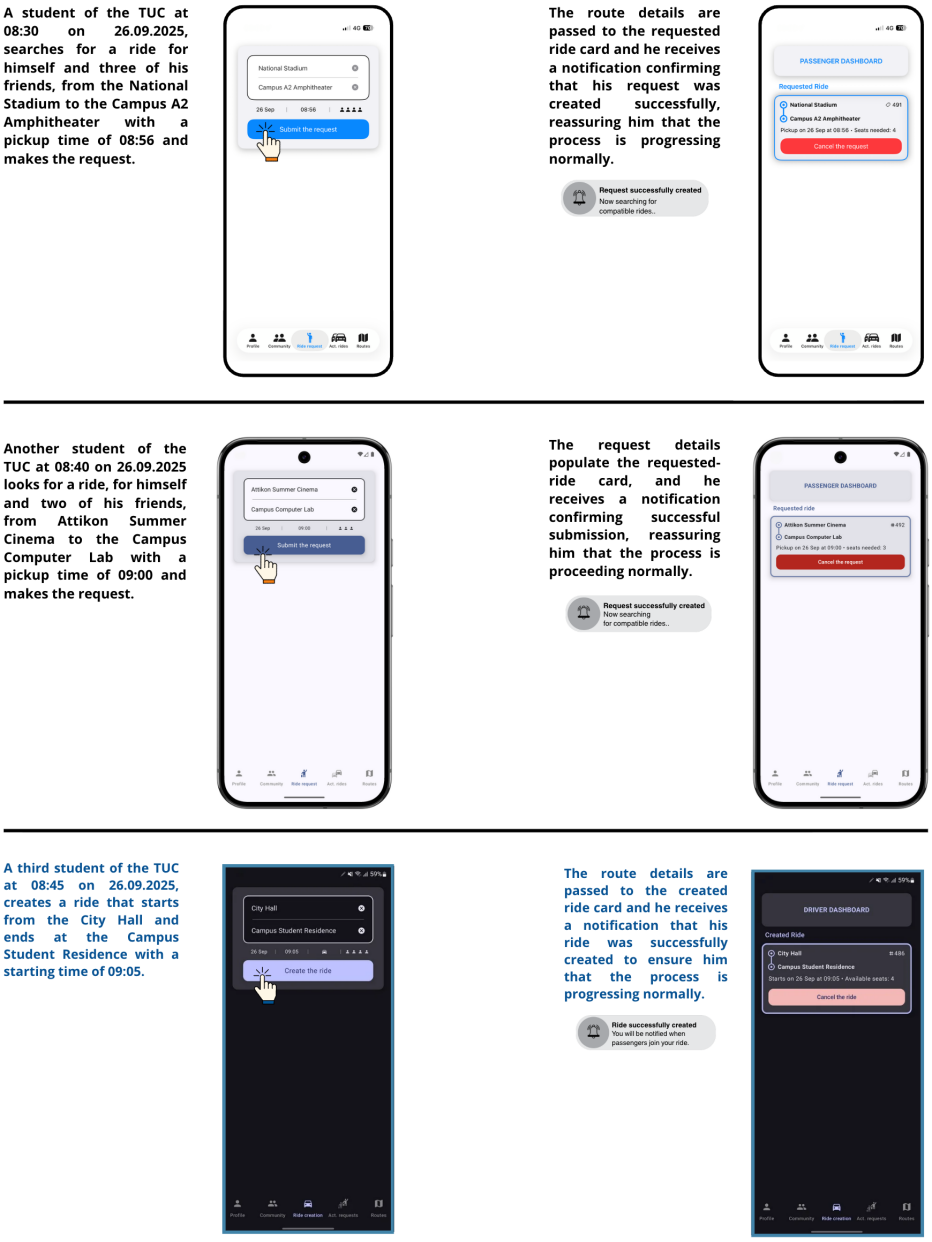
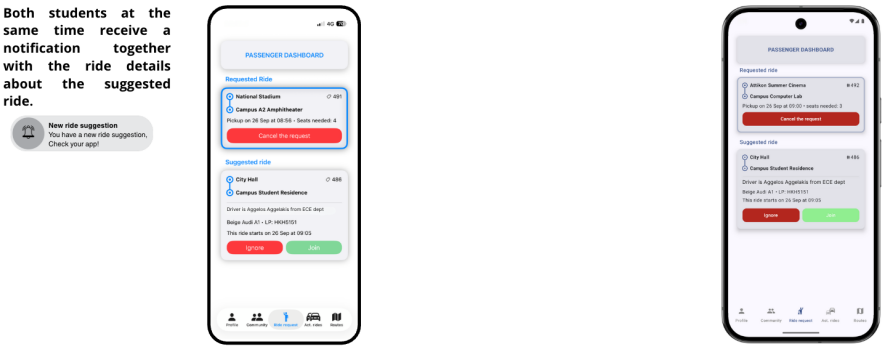
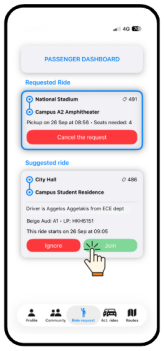


FIGURE 3.8: Usage scenario 2, part 1 of 4.

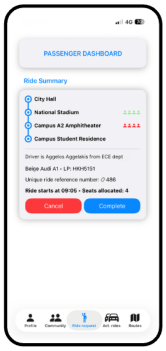
Both students at the same time receive a notification together with the ride details about the suggested ride.



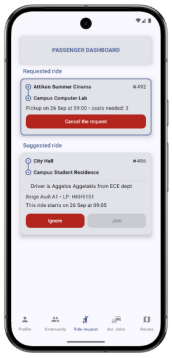
The first student presses the 'Join' button and enters the ride with his three friends.



The passenger ride summary card appears on the screen of the first student with all the details of the ride.



At the same time the 'Join' button on the screen of the second student gets disabled, because the ride no longer fits the criteria (the seats are no longer enough to serve his request).



The data of the passenger that made the reservation, appear on the driver's screen together with a notification.

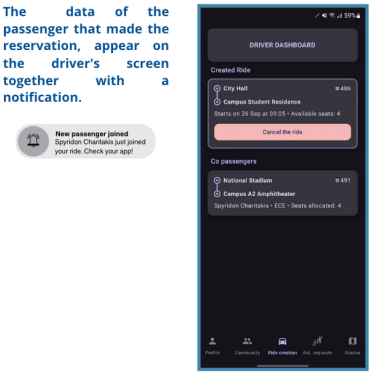
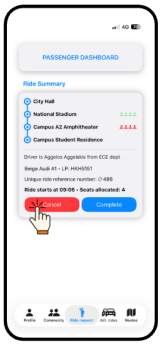


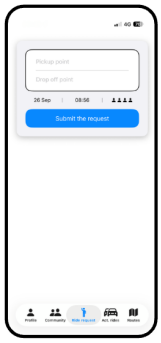
FIGURE 3.9: Usage scenario 2, part 2 of 4.

At 08:50, the student who reserved the seats receives an email stating that the course he planned to attend with his friends has been rescheduled to the next day.



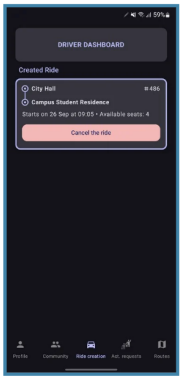
The UI returns to its initial state and the student receives a notification that notifies him that his reservation was succesfully canceled.

Request Canceled
Your ride request was successfully canceled

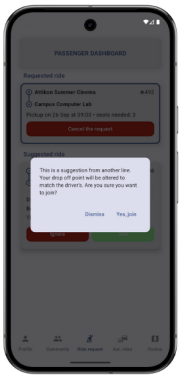


The card with the data of the passenger that canceled the reservation disappears from the driver's screen who also receives a notification about the event.

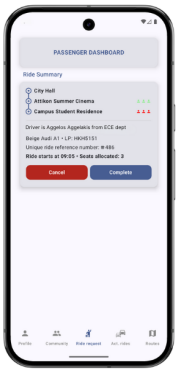
Passenger Cancelation
Spyridon Charitakis just canceled their reservation



At the same time the suggested ride becomes available again for the second student and the two of his friends. He presses the 'Join' button and he sees that the ride is from a different line (line 1).He agrees to alter his ending point to match the driver's and he joins the ride.

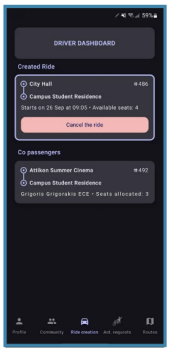


The passenger ride summary card immediately appears on his screen.



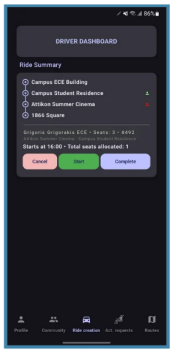
The passenger's details appear on the driver's screen, and a notification informs him that a new passenger has joined the ride.

New passenger joined
Grigoris Grigorakis just joined your ride. Check your app!



Twelve minutes before the ride begins, the ride summary card appears on the driver's screen with buttons to Cancel, Start, and Complete the ride. The driver also receives a notification that the ride is approaching to allow for preparation.

Ride starting soon
Your ride starts soon, please get ready.



Similarly, the passenger receives the same notification.

Ride starting soon
Your ride starts soon, please get ready.

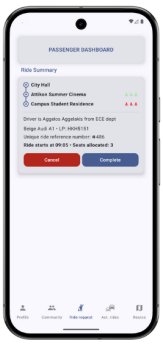
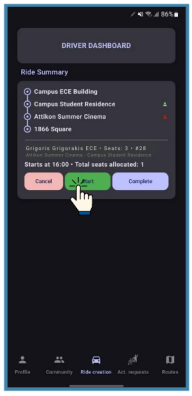


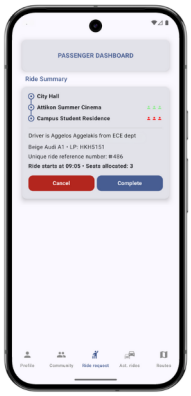
FIGURE 3.10: Usage scenario 2, part 3 of 4.

The driver moves towards his vehicle to start the ride..

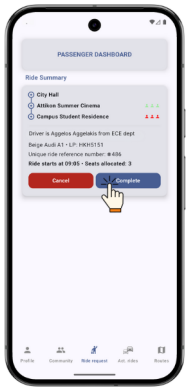
He enters the vehicle and presses the 'Start' button to notify the passenger that he departed.



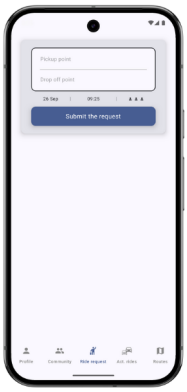
The passenger receives the notification on his device and moves with his friends towards the pickup point.



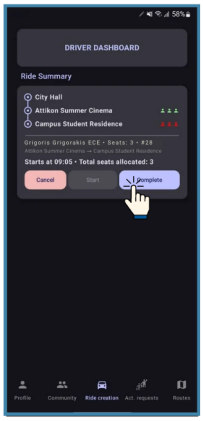
Finally, the vehicle arrives at the passenger's pickup point, the passengers enter and they travel together until their drop off point. The passengers exit the vehicle and the student that made the reservation presses the 'Complete' button to complete the ride.



He receives a notification that his ride was completed. The application UI has returned to its original state, ready for the next ride.



Following that, the driver parks his vehicle and presses the 'Complete' button on his ride summary card.



He similarly receives a notification that his ride was completed. The UI has returned to its initial state.

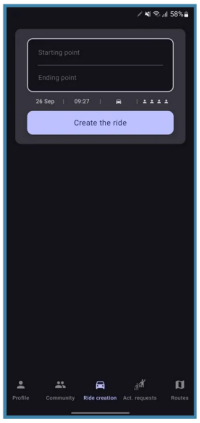


FIGURE 3.11: Usage scenario 2, part 4 of 4.

Chapter 4

Frontend and Backend Design and Implementation

Building a full-stack, cross-platform ridesharing application requires several coding languages, IDEs, and frameworks. Furthermore, it includes careful planning and closely following the guidelines of the official documentation for each platform. The foundation of the application is a Kotlin Multiplatform Project (KMP) that houses the Android frontend and the common code that is used by both our OSes. TUC Ridesharing at a high level is an Android and iOS application with a common FastAPI python backend, and MVVM architecture.

The architecture of the application is Model-View-ViewModel (MVVM). We chose this architecture in order for the application to be modular, easier to maintain, and expand in the future. The three individual parts are: The Model (CommonMain) which focuses on the communication with the backend, the View (Jetpack Compose/SwiftUI) which focuses on rendering the UI, and the ViewModel that handles the UI logic and manages the various states. This approach ensures a clean separation of concerns. By building native UIs, we enhance the user experience because we can follow the latest guidelines from both documentations. We followed the Material You design guidelines for the Android frontend and those of the Apple Human Interface for the iOS, in order to have a clean, modern, and cohesive experience for the users on both platforms.

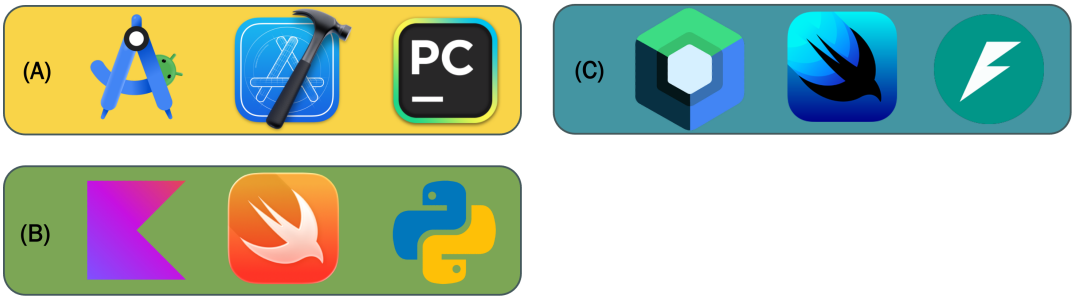


FIGURE 4.1: IDEs in (A), languages in (B), and frameworks in (C).

4.1 System Architecture

We used multiple IDEs, languages, and frameworks to complete this project. The IDE we used for the Android and the common client code is Android Studio. It is the recommended solution in the Android documentation and it is extremely easy to use. It makes the procedure of integrating the project with the Firebase solutions we needed (Firebase Cloud Messaging and Firebase Crashlytics) effortless. The language we used is Kotlin which is the official and fully supported language by Google, has simple syntax which dramatically reduces boilerplate code, and is future proof because Google invests heavily on its development. That is why many of the new libraries that come out are Kotlin-only. The framework used is Jetpack Compose. We chose this particular framework because it is officially backed and supported by Google. It is the recommended solution to build Android apps and makes coding simple because it is a declarative UI framework.

The iOS part was created using Xcode, which is the suggested development tool from Apple. It has a very simple interface and seamless integration with all the Apple SDKs. Furthermore, it supports code autocompletion, UI and unit testing. The language used is Swift, which is the preferred and suggested language by Apple. It is open source, widely used, and has deep integration with Xcode. The framework used is SwiftUI, Apple's modern declarative framework. It makes state management easy by utilizing property wrappers and is multi-platform across Apple platforms by design, meaning that with one codebase we can run an application in iOS, iPadOS, MacOS, VisionOS, watchOS and tvOS.

The server side of the application was built using PyCharm. Our IDE decision was based on the fact that it has the same development team behind it as Android Studio. That means that if someone has limited experience with the former, they could become familiar with this tool in a very short period of time, which was our case. Furthermore, JetBrains offered a one-year free license for qualified students, which upgraded the IDE to the Professional Edition, and made it more feature rich, and the development in it even faster. The framework used is FastAPI and the language is Python. We chose this powerful combination to accelerate our development. Integration with our PostgreSQL database, Firebase Cloud Messaging notifications, and CAS authentication became seamless. Communication with the frontends became effortless through REST endpoints.

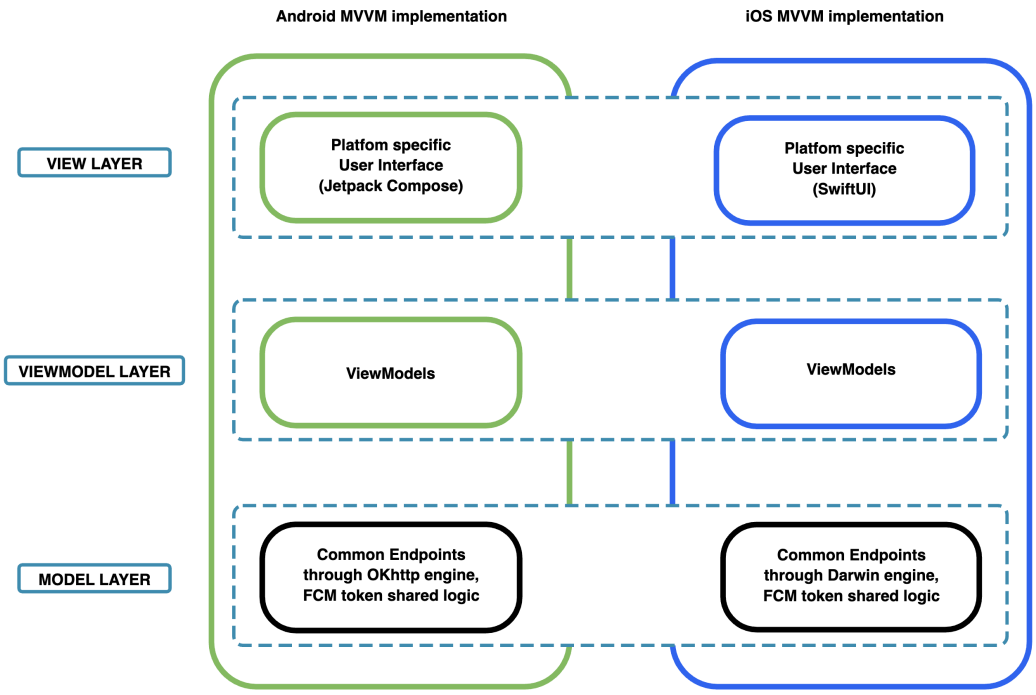


FIGURE 4.2: Implementation of the MVVM architecture on both platforms.

4.2 User Interface/User Experience

The user interface on each platform consists of a splash screen and 10 different screens. The Android part was built using the Jetpack Compose framework, the Kotlin language, and following the latest Material You guidelines. The iOS counterpart was built using the SwiftUI framework, the Swift language, and following the Apple Human Interface guidelines. High-resolution images of the interface and its different layouts can be found here. The description of each screen is as follows:

- The **Splash screen** that consists of a classic splash screen with a theme-adaptive application logo and background.
- The **Login1of2 screen** which contains a **Next** button that navigates the user to the login WebView.
- The **Login2of2 screen** which is the actual WebView screen that presents the login form where the users enter their credentials.
- The **Mode screen** that is the target screen after a successful login and the one the users land on when they are already logged in and reopen the

application. On this screen, users select their mode (passenger or driver) of operation.

- The **Profile screen** that contains the personal information of the users. Here they can select a profile photo, change the theme of the app, give feedback though the integrated feedback form, open the FAQ page or log out.
- The **Community screen**, where each user of the application has its own card, that displays minimal data about them. The purpose of this screen is to reinforce the feeling of the community within the application and become the base for a planned feature that involves ridesharing social score.
- The **Passenger Main screen** where the passenger-side ridesharing procedure takes place. It contains a date picker, a time picker, pickup and drop-off search bars and a seat selector. After submitting a request, it displays suggested rides and when the passenger joins a ride, it displays the passenger ride summary card. All fields are validated to ensure they contain valid data. The pickup and drop-off points are checked to be non-empty and that they contain valid locations from the predefined lists. The date and time combination is checked to be at least 15 minutes in the future, and the seats field is checked to be non-zero. If all the data is correct, it is sent to the backend where it is stored, and a successful response is sent back to the client side that contains the data of the request. This triggers the collapse of the search card, the creation of the requested ride card and opens the WebSocket stream after three seconds so that the passenger is able to receive suggested rides in real-time.
- The **Active requests screen**, that displays all the requests with status `AVAILABLE` and pickup time no more than 45 minutes in the past, to align with the matching algorithm time-window. That way, drivers only see requests within a time window in which they can match.
- The **Driver Main screen** where the driver-side ridesharing procedure takes place. It contains a date picker, a time picker, starting and ending point search bars, the vehicle description data fields (color, brand, model, license plate number) and the available seats selector. All fields undergo validation to verify data correctness. The starting and ending point are checked to be non-empty and that they contain valid locations from the predefined lists. The color, brand, and model fields are checked to be non-empty and that contain a choice from the predefined lists of the application, the license plate to consist of three letters followed by four numbers. The date and time combination is checked to be at least 15 minutes in the future, and the seats field is checked to be non-zero. If all the data is correct, it is sent to the backend where it is stored, and a successful response is sent back to the client

side that contains the data of the ride. This triggers the collapse of the search card, the creation of the created ride card and opens the WebSocket stream so that the driver can receive in real-time the data of the passengers that joined the ride. The driver ride summary card is also created and displayed here, 12 minutes before the departure.

- The **Active rides screen**, that displays all the rides with status `AVAILABLE`, have at least one available seat, and start at least 15 minutes in the future. They remain visible until 12 minutes before their start time, to align with the ridesharing process constraints.
- The **Routes screen** that contains the 20 route cards, 10 route cards starting from Campus and 10 ending to Campus. Each location has a small description and contains a map link to the exact coordinates of the point.

For both passengers and drivers on their main screens, we implemented mechanisms to make their life easier and the whole process of creating a ride or submitting a request, more streamlined. The predefined date and time combination when the user enters either of the two main screens is 20 minutes in the future, so that they have enough time to fill in the data without having to interact with the two pickers. For the same reason, the seat selector value is saved to local storage so that in subsequent requests, if the user does not want to change the seat value, they do not have to deal with it again. The same local storage solution is implemented for the color, brand, model, and license plate fields. All these mechanisms make the whole process faster and ensure that the core data of each user is only set up once, the first time.

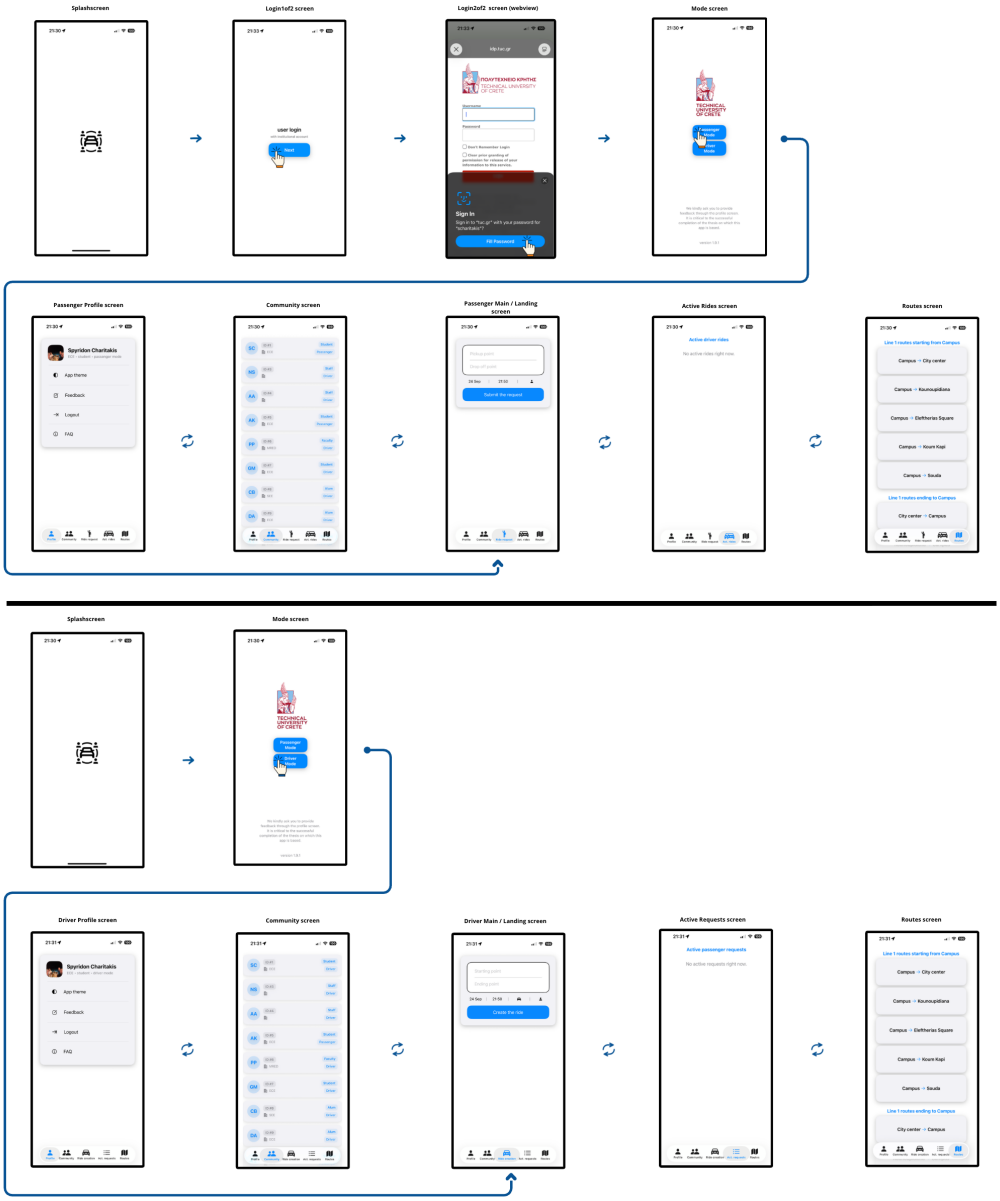


FIGURE 4.3: Navigation flow when logged out, and in subsequent sessions.

For larger screens on both platforms, we designed custom implementations to utilize the extra real estate and make elements look better overall. In general, the Splash screen, Login1of2 screen, Login2of2 screen, and Mode screen remained the same across all screen sizes. On Android we customized the two main screens with more horizontal padding for the search and the ridesharing cards so that they do not look stretched. The Profile screen also received more horizontal padding, a larger profile photo window, and custom

font size for the user information. On the Routes screen, we implemented a two-column layout where we put the 10, line one route cards on the left and the 10, line two route cards on the right. The Community, Active rides, and Active requests screens, all utilize a two-column layout where half the number of cards they display are on the right and the other half on the left. This exact layout can be seen on Android tablets and on the internal screen of foldables. It is important to point out that foldables use the phone layout when the user operates the application on the external screen, and the same happens when the user makes use of the internal screen in split-screen together with another application. This is especially useful when using TUC Ridesharing side by side with Google Maps. Screenshots of the different Android layouts can be seen in Figure 4.4 and Figure 4.5 below.

FIGURE 4.4: Two-column layout of the Community screen, on foldables and Android tablets.

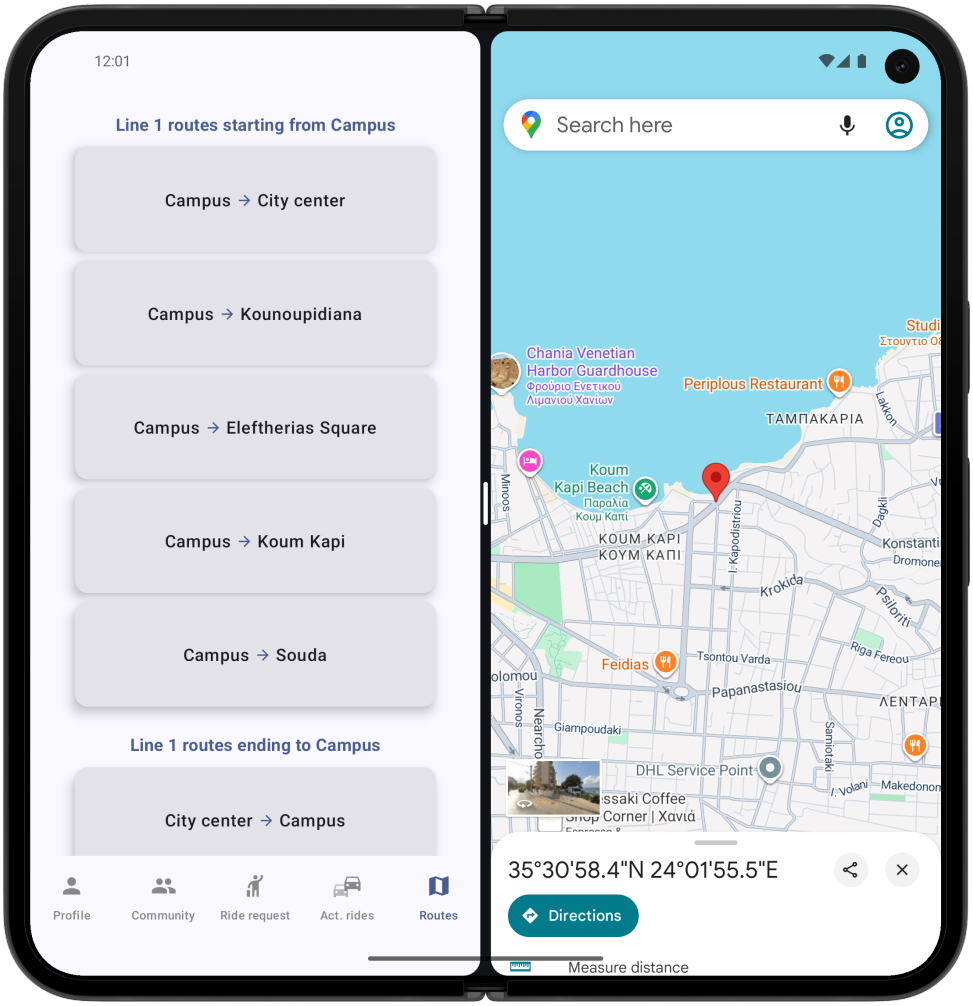


FIGURE 4.5: Phone layout of the Routes screen, when in split-screen mode.

On iPadOS/MacOS (Apple Silicon)/VisionOS we customized the two main screens with more horizontal padding for the search and the ridesharing cards so that they do not look stretched. The Profile screen got a complete overhaul with Split View layout with sidebar. The photo picker, the feedback WebView, the theme picker, and the logout option are all customized. The Community, Active rides and Active requests screens, all utilize a two-column layout just like in the Android version. Half the number of cards they display is on the right and the other half on the left. On the Routes screen we implemented a Split View layout where we put all 20 route cards on the left. When the user presses the **info** button on each location, on the right side of the screen it displays the location on the embedded map and its description on top. Screenshots of the alternative iPadOS/MacOS/VisionOS layout can be seen in Figure 4.6 and Figure 4.7 below.

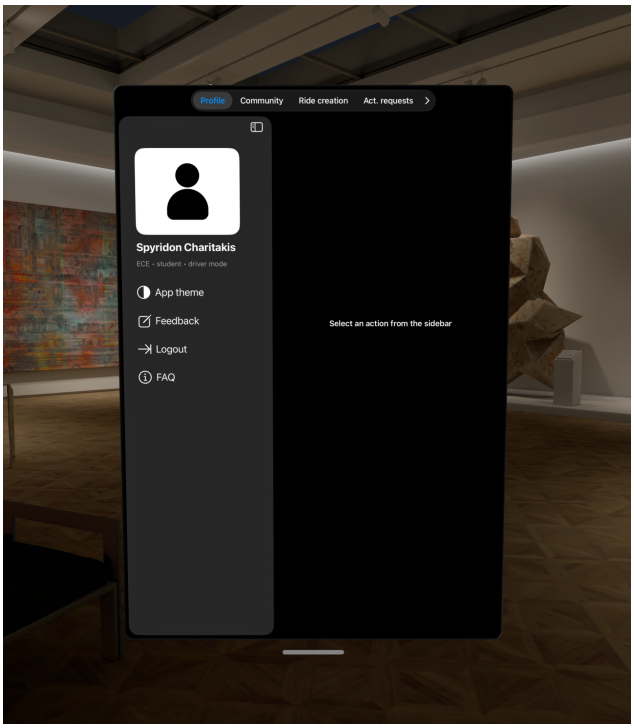


FIGURE 4.6: Split View layout Profile screen on VisionOS.

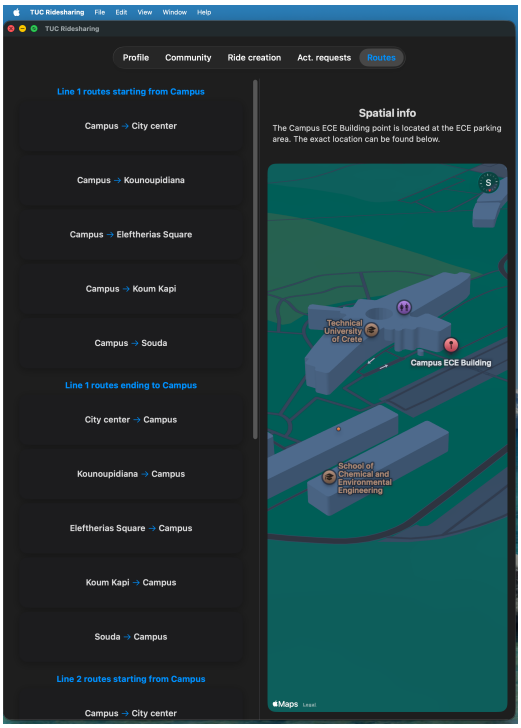


FIGURE 4.7: Split View layout Routes screen on MacOS.

4.3 ViewModel Layer

Our application utilizes platform-specific ViewModels that collect, save, and restore data, as well as handle the lifecycles of the ridesharing process and manage the WebSocket connections. Each platform has six ViewModels. The offeredRide ViewModel that supports the driver-side ridesharing process, the requestedRide ViewModel that supports the passenger-side ridesharing process, the useAs ViewModel that collects the user mode. The communityScreenViewmodel that assists the Community screen, the availableRequestsViewmodel and the availableRidesViewmodel that maintain the functionality on the Active requests and Active rides screens, respectively. The description of their functionality can be found below.

- **OfferedRide ViewModel**: It supports the Driver Main screen and the driver ridesharing process. It calls the ride creation endpoint and stores the successful response. It is also used to cancel or complete the ride by calling the cancelation and completion endpoints, respectively. The complete handling of the WebSocket also takes place here. The WebSocket opens when there is a successful response after the ride creation for the device to receive the passenger cards when they join the ride. On both Android and iOS, a lifecycle observer monitors when the application is in background or foreground, in order to close or open the WebSocket. It also checks if the ride is 'locked' so that the WebSocket does not open again after the ride summary is visible.

- **RequestedRide ViewModel**: It handles the passenger ridesharing process. It calls the passenger request endpoint and persists its data upon successful response. It receives suggested rides over the passenger WebSocket and exposes them. It handles the passenger join and displays the ride summary card. As in the offeredRide ViewModel the complete handling of the WebSocket takes place here. The WebSocket opens three seconds after an active request is created and then closes when the application goes to the background. When users re-enter the app, it opens again if the request is not 'locked', meaning if the passenger ride summary is visible. Cancelation and completion of the passenger request is handled here by calling the corresponding endpoints.

- **UseAs ViewModel**: The useAs ViewModel tracks the mode the user is in (passenger or driver mode), captures the input and exposes it to local storage and the backend via the updateUseAs endpoint.

- **CommunityScreen Viewmodel**: This ViewModel assists the Community screen. Its job is to track the current list of users, check that the data load correctly, and display error messages if there is an issue. It opens the

WebSocket, updates the UI and closes the connection when the user leaves the screen.

- **AvailableRequests Viewmodel**: It supports the Active requests screen with the live list of available passenger requests. It opens and closes the Websocket, and updates the UI.
- **AvailableRides Viewmodel**: The availableRides ViewModel keeps the Active rides screen updated with the live list of available ride offers. It opens the WebSocket, refreshes the list when there is an update, and closes the WebSocket when the user leaves the screen. It also exposes states to update the UI.

4.4 Network Layer

The network layer of our application resides in the commonMain module of the Kotlin Multiplatform Project and is used directly from the Jetpack compose Android part and through the generated shared framework in iOS/iPadOS/MacOS/VisionOS. It contains all the necessary components for the correct communication with the backend layer. Those are the data models, the FCM token handling logic, and the client endpoints. It also contains platform-specific engines for both Operating Systems, OkHttp for Android and Darwin for iOS. Our commonMain uses 10 REST endpoints in total for POST, PATCH, GET, and five WebSocket streams. These are:

- **PATCH / users / {user_id} / use_as**
  It is used for updating the mode of the user by utilizing their user_id. It gets triggered when the user presses either the **Passenger Mode** or the **Driver Mode** button on the Mode screen.
- **POST / requested_rides**
  It is used for submitting ride requests. It contains all the data of the request like date, time, pickup point, drop-off point, and seats needed. It gets triggered from the **Submit the request** button on the Passenger Main screen.

  The response includes: request_id, user_id, passenger_pickup_point, passenger_dropoff_point, pickup_time, pickup_date, seats_needed.

  It inserts the ride request to the database and updates its status to `AVAILABLE`. Three seconds later, it triggers the match-making algorithm to perform matches, schedules a Redis key to expire one hour after the pickup time (if the request at that time is still `AVAILABLE`) and notifies the passenger about the successful submission of the ride.

- **PATCH / requested_rides / {request_id} / cancel**
  It is used for canceling a ride request. It uses the request_id and its tasks are to update the status of the request to `CANCELED` and notify the passenger about the successful action. If the passenger had previously joined a ride, then the seats they allocated are released, the driver is notified about the event, the match-making algorithm is triggered, and the event is published so that other passengers can allocate the newly released seats.

- **PATCH / requested_rides / {request_id} / complete**
  It is used for the passenger ride completion. It is triggered by the **Complete** button on the passenger ride summary card . It updates the status of the request to `COMPLETED` and sends a notification to the passenger, to inform them about the success of the action.

- **PATCH / suggested_rides / {suggestion_id} / ignore**
  It is used for ignoring a suggested ride from the list. The suggested ride status changes to `IGNORED` and it gets cleared from the passenger's screen (the suggestion WebSocket only forwards suggestions with `AVAILABLE` status). It gets triggered when the passenger presses the **Ignore** button on a suggested ride card.

- **POST / join_ride / request_id, offer_id**
  It is used from a passenger in order to join a ride. It utilizes the passenger request_id, the driver offer_id and first checks if the ride is indirect, in order to adjust the pickup or drop-off point of the passenger. Then calls the join_ride function to check if the status of both the ride request and the ride offer is `AVAILABLE` and if the ride has enough available seats. If so, the new match is inserted to the matched_rides table, the requested ride status is updated to `MATCHED` and the allocated seats are subtracted from the available seats of the ride. All request_ids that have a suggestion from that offer are retrieved and the backend publishes to the suggested_rides:<passenger_request_id>. Then, the WebSocket listens and delivers the up-to-date details of the ride, to the affected passengers. The match-making algorithm is triggered to recompute suggestions for everyone, and the new ride suggestions are published and pushed to all compatible passengers via FCM notifications, Redis and WebSockets. The data of the new passenger that joined the ride are published to the matched_rides:<offer_id> and the WebSocket handler sends them to the driver's device.

  The response that the passenger receives contains the following: match_id, passenger_request_id, driver_offer_id, passenger_pickup_point, passenger_dropoff_point, driver_starting_point, driver_ending_point, start_time, start_date, seats_allocated, driver_name, driver_surname, driver_department, color, brand, model, license_plate.

This endpoint is triggered when the user confirms the join on a suggested ride card.

- **PATCH / offered_rides/{offer_id}/ cancel**
  It is used for canceling a created ride. It uses the offer_id and its tasks are to delete the scheduled 12 minute (before the ride) key from Redis, update the status of the offered ride to `CANCELED` and find all passengers affected. Then it updates their suggestion status to `CANCELED`, Publish/Subscribe publishes to the suggested channels to update their list of suggested rides through WebSockets and simultaneously an FCM notification is sent to inform them about the event. The driver receives an FCM notification with the list of the affected passengers and Publish/Subscribe publishes on the matched channel to clear the passengers that previously joined. If the driver did not have any matches up until the cancelation, the FCM they receive just updates the ride status and informs them about the successful cancelation.

- **PATCH / offered_rides / {offer_id} / complete**
  It is used for the ride completion. It updates the status of the offered ride to `COMPLETED` and sends a notification to the driver to inform them about the success of the action. It gets triggered by pressing the **Complete** button on the driver's ride summary card.

- **POST / offered_rides**
  It is used for creating a ride. It contains all the ride data: date, time, starting point, ending point, available seats, color, brand, model and license plate number of the vehicle. It gets triggered from the **Create the ride** button on the Driver Main screen.

  The response includes the: offer_id, ride_starting_point, ride_ending_point, start_time, start_date, available_seats.

  It inserts the ride to the database, updates its status to `AVAILABLE`, triggers the match-making algorithm to perform matches and notifies the driver about the successful ride creation. Then five TTL keys are scheduled. One that schedules a Redis key to fire 12 minutes before the departure (to inform all members of the ride that it approaches and deliver to the driver the final passenger list). One that is triggered five minutes after the ride start time and checks if the driver pressed the **Start** button (and notifies them if they did not). One that auto-cancels the ride if 15 minutes after the departure, the driver has not yet pressed the **Start** button. One that is triggered one hour after the scheduled ride start time and checks if the ride status of all the members of the coalition is `CANCELED/COMPLETED` or otherwise it informs them to complete their ride. Lastly, one that one hour and 15 minutes after the scheduled ride start time, it auto completes the

ride of all the members of the coalition whose status is not yet `CANCELED`/`COMPLETED` and resets their application UI.

- **PATCH / offered_rides / {offer_id} / start**
  It is used for starting the ride. It updates the status of the offered ride to `IN_PROGRESS` and notifies via an FCM notification all the matched passengers about the event. It gets triggered when the driver presses the **Start** button on the driver ride summary card.
- **WS / ws / matched_rides / driver_offer_id**
  It is a WebSocket for streaming the passenger data of people who joined a ride, to the Driver Main screen. On connect it retrieves all matched passengers and subscribes to the channel matched_rides:<offer_id> to get new matches, i.e. passengers that joined their ride.
- **WS / ws / suggested_rides / passenger_request_id**
  It is a WebSocket for streaming suggested rides to the Passenger Main screen. On connect it retrieves all compatible suggestions with status `AVAILABLE` and subscribes to the channel suggested_rides:<passenger_request_id> to receive new suggestions.
- **WS / ws / users**
  It is a WebSocket stream for streaming the users' data from the users table. It is used on the Community screen.
- **WS / ws / available_requested_rides**
  This WebSocket is used to fetch all available passenger requests with status `AVAILABLE` that are at most, 45 minutes in the past.
- **WS / ws / available_offers**
  This WebSocket is used to fetch all rides with status `AVAILABLE`, that start at least 15 minutes in the future, have at least one available seat, and are at most 12 minutes before their starting time.

4.5 FastAPI Backend Layer

The backend of our application consists of four main components. The CAS authentication component, the PostgreSQL database component, the match-making algorithm component (described extensively in Section 3.3.2) and the notifications component. All four are utilized using the FastAPI framework in Python. The server also utilizes Redis Publish/Subscribe as the messaging pattern and the (NGINX) reverse proxy that accepts the HTTPS client calls, decrypts them, and forwards them as plain HTTP to the server. The complete server-side part of the application is hosted on the Technical University of Crete servers and runs as systemd 24/7/365 on a Linux virtual machine.

4.5.1 Central Authentication Service (CAS) authentication

The authentication module handles the flow that enables users to log in and log out. Only users of the Technical University of Crete can sign in using their institutional credentials. When the application launches, it checks the 'isLoggedin' flag to see if it is true or false. If the flag is true, then it launches the user on the Mode screen, meaning that they are logged in. However, if the flag is false, the application takes the user to the pre-login screen or Login1of2 screen where the **Next** button resides. When this button is pressed, the application builds the /login URL and opens a WebView. The backend redirects the user to the Technical University of Crete login screen where the user can enter their institutional credentials, and if they are valid, a Service Ticket is issued and the browser is redirected back to the /callback endpoint. There, the CAS serviceValidate endpoint is called to confirm that the Service Ticket is valid. If it is indeed valid, then an XML response is parsed that contains the basic profile data of each user, the name, surname, affiliation, and department.

If the user does not exist in the database, a new row is inserted into the users table. A new session token is created that is valid for 30 days and is stored in Redis. Then, the user is redirected back to the Mode screen of the application with the ticket, via a deep link. There, the ticket is extracted and the /callback endpoint is called again with client=true in the URL, in order to receive the JSON that contains the session token, the user profile data and a Set-Cookie header. The name, surname, affiliation, department, session token and the boolean flag isLoggedIn (the latter set to true) are saved in sharedPreferences in Android and UserDefaults in iOS. The cookie is automatically detected by the HTTP client and saved in its cookie jar. Then it is used to make authenticated API calls to the backend. At this point, the user is inside the application and can start using its functions. Both platforms are fully compatible with biometrics, Android with fingerprint and face authentication, and iOS with FaceID and TouchID.

The logout process is much simpler. When the user confirms the logout on the Profile screen, a performLogout() function is triggered that clears the isLoggedIn flag, the session token, and all the personal information of the user from local storage. At the same time, it clears the WebView's cookies and calls the backend's logout endpoint, which handles the server-side logout. There, the Redis session of the token is deleted, the actual session_token is cleared, and the CAS logout URL is called that ends the SSO session. As a last step, the user is redirected back to the Login1of2 screen inside the application; the same screen they land on whenever they launch TUC Ridesharing without being logged in. There is also an automatic logout mechanism that logs users out after 30 days, so they can log in again, and the server can issue a new session token.

4.5.2 PostgreSQL relational database

The database used for storing user and ridesharing data is a PostgreSQL relational database. It consists of six tables with the data integrity mandated through primary and foreign keys and enum-typed columns. The users_table that stores the four attributes returned after a successful login, as well as the mode in which the users operate the application. The device_tokens table that stores the device's unique token that is used to send FCM notifications. The requested_rides table that stores the data for each passenger request made. The offered ride table that stores the data for each ride created by the drivers. The suggested_rides table that contains all the suggested rides generated from the match-making algorithm, and the matched_rides table that contains all the data of the users that travel together.

Indexes are used in several places within the server to ensure faster lookups and operations in general. We put those indexes in place for operations that occur frequently. The four most important indexes are:

- **Ix_requested_rides_status** that speeds up the queries that search for passenger requests filtered by their status. For example, it is used in the match-making algorithm where the status of the requests must be `AVAILABLE`, and in the expiration listener, where requests whose status is still `AVAILABLE` one hour after their pickup time, must be canceled.

- **Ix_offered_rides_status** that speeds up queries that find rides based on their status. For example, it is used in the match-making algorithm where all created rides must have their status set to `AVAILABLE` so that all irrelevant data are not considered. Its also used when a passenger joins a ride, where there is a check of the offered ride status and if it is still `AVAILABLE` at that moment.

- **Ix_suggested_rides_by_passenger_and_status** that makes fetching all suggestions based on their status, for a passenger, faster. It is used in various cases, for instance the WebSocket for the suggested rides returns only rows whose status is `AVAILABLE`. It is also used when a passenger wants to join a ride, and we want to ensure that all the suggestions at that time are valid, through their correct status (`AVAILABLE`).

- **Ix_suggested_rides_by_driver_and_status** that accelerates operations regarding that particular suggested ride and their status. For instance, 12 minutes before the ride starts, suggestions that are still `AVAILABLE` must be expired so that the passengers are notified that the ride is no longer available to join. It is also used when a driver cancels a ride, and all passengers whose status is still `MATCHED` must be notified and prompted to search for another.

Row locks are implemented in the database to ensure safe concurrent operations and prevent race conditions to specific rows. This means that one transaction must commit or roll back so that the next transaction can take place. This is especially useful for our application because we prevent stale reads and updates. We use row locks in four different places on our server.

- In the **join_ride function** where the passenger request and the driver ride are locked before joining, to prevent overbooking.
- In the **driver's cancel ride endpoint** where the ride is locked to prevent an accidental passenger joining, at the same time the ride is canceled. It also prevents the 12-minute TTL key from expiring, and the server cannot send a "Ride starting soon" notification when the ride is in the process of being canceled.
- In the **passenger's cancel request endpoint** where the request is locked to prevent concurrent updates to that specific request. For example, from the one-hour expiration TTL key and prevent redundant updates from reaching the passenger.
- In the **one-hour expiration listener** where the request is locked. By doing this, we prevent duplicate updates from reaching the passenger. For example, when the expiration is seconds away and the passenger cancels the request at the same time. Then only one update will go through and the passenger will not end up receiving duplicate notifications for both events.

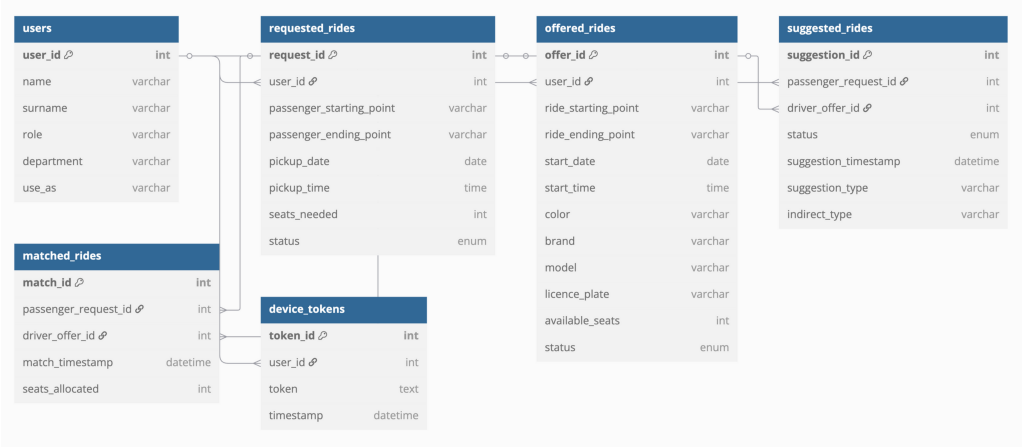


FIGURE 4.8: ER diagram of the database tables and their relationships.

4.5.3 Notifications

Our backend implementation involves the use of three different kinds of notifications by which users are notified about important events, and in short coordinate. These are the Redis Key-Expiry-Driven notifications, the WebSocket notifications via Redis Publish/Subscribe, and the FCM notifications.

The Redis Key-Expiry-Driven notifications are used by the server to schedule TTL keys and upon their expiration different actions are triggered. We have six different kinds. One to inform passengers that their request has expired. One for sending a notification 12 minutes before the ride starts, to inform its members that it approaches. One that is sent to the driver if five minutes after the scheduled departure they have not started the ride. One that is sent 15 minutes after the scheduled ride start time for the ride auto-cancelation. One that is sent one hour after the scheduled departure to inform the members of the coalitions to complete their ride and one that is sent one hour and 15 minutes after the scheduled ride start time to inform the members of the coalition that their ride was automatically completed and reset their application UI. All six Redis Key-Expiry-Driven notifications use FCM notifications to deliver updates to the users.

- Inside the request creation endpoint, we set the expiration time by getting the pickup time and adding one hour to it. Then Redis emits the expired-key event and the listener checks the status of the request. If it is `AVAILABLE`, meaning that the passenger did not cancel, complete or match during that time, then the status is set to `EXPIRED`. An FCM notification is sent to the passenger to inform them about the event and prompts them to create a new one.

- The second TTL key is scheduled inside the ride creation endpoint. There, the expiration time is calculated by getting the ride's start datetime and subtracting 12 minutes from it. When the timer gets to zero, Redis emits the expired-key event, the listener checks the status of the matched passengers and keeps only those whose status is `MATCHED`. This way users that canceled or completed their request do not receive any updates. The passengers that are still in the coalition, get an FCM notification which informs them that the ride starts soon and to get ready. The driver receives a similar notification and a second data-only one, that holds the final passenger list and helps create the driver ride summary card. The remaining passengers that have the suggested ride on their screen up until that moment but did not join, have their suggested rides expired in the database and the event is published, the WebSocket pushes the update to their devices and the suggested ride is cleared from their screens. At the same time an FCM notification is sent that informs them that the ride is no longer available. If a driver cancels

their ride before the 12 minute mark, then inside the cancel endpoint of the ride, the TTL key is removed, and the process stops at that moment.

- The third TTL key is also created inside the ride creation endpoint and has a time expiration of the ride's start datetime plus five minutes. When it expires, the listener checks the status of the ride and if it is not `IN_PROGRESS`, meaning that the driver did not press the **Start** button, a notification is sent to let them know that they should either start or cancel the ride. The list of active passengers on that ride is also checked and if its empty then no notification is sent.

- The fourth TTL key is created in the ride creation endpoint and has a time expiration of the ride'a start datetime plus 15 minutes. When it expires, the listener checks the status of the ride and if it is still `MATCHED`, meaning that the driver did not press the **Start** button, a notification is sent to all the members of the coalition and informs them, that the ride was automatically canceled. Again if the list of active passengers on that ride is empty then no notification is sent.

- The fifth TTL key is scheduled inside the ride creation endpoint and expires at the ride's start datetime plus one hour. When it expires, the listener checks the ride status of the users that are part of the coaliton. If their status is not `COMPLETED/CANCELED` they receive a notification that encourages them to complete the ride.

- The sixth TTL key is created in the ride creation endpoint and expires at the ride start datetime plus one hour and 15 minutes. When it expires, the listener checks the status of the users that were part of the coaliton. If the status of their ride is not `COMPLETED/CANCELED`, their status is updated to `COMPLETED` and they receive a notification that informs them about the action and resets their application UI.

The Websocket notifications are responsible for streaming real-time JSON updates on the five channels of the application. The first channel is the ws/suggested_rides:<passenger_request_id> and is used for the suggested rides every passenger receives, after they submit a ride request. Updates are pushed after the match-making algorithm is triggered, on suggested rides expiration, and on ignore. The second channel is the ws/matched_rides:<offer_id>. The data sent is the current list of passengers that joined the ride, and is pushed to the drivers after a successful passenger join, cancelation, and completion. The third channel is the ws/users that streams the user data on the Community screen. The fourth channel is the ws/available_offers that delivers the available rides on the Active rides screen, and lastly the fifth channel is the ws/available_requested_rides that streams all available requests on the Active requests screen.

Firebase Cloud Messaging (FCM) Notifications is the chosen push-notification solution, because it is fully compatible with Android, iOS, the FastAPI Python backend, and the Kotlin Multiplatform Project that houses our client calls. The notification mechanism operates as follows: In every installation, the device asks for a unique token from the FCM server. The FCM SDK sends the installation ID and the project credentials; then the FCM server validates the data and issues a URL-safe opaque string that is cached on the device with the help of the FCM SDK. The token is sent to our backend for storage and is used to send targeted notifications. Usually, the FCM token stays the same for the lifetime of the application installation, except for special occasions. For security reasons Google can force token rotation and in cases of multiple month token inactivity, a new one is issued the next time it is needed. When a notification is needed, a helper function is called that retrieves the user's token and sends the notification to the target device. Each message has title, body, target screen, and priority which is always set to high due to the nature of the application. There are 16 different events that trigger notifications; some send only one notification per event and some multiple. The number of distinct FCM notifications is 22. The complete FCM list can be found below.

- **Title:** "Ride successfully created".
  Body: "You will be notified when passengers join your ride".
  It gets triggered when a new ride is successfully created and the target is the Driver Main screen.

- **Title:** "Ride request successfully created".
  Body: "Now searching for compatible rides..".
  It gets triggered when a new ride request is successfully submitted and the target is the Passenger Main screen.

- **Title:** "New ride suggestion".
  Body: "You have a new ride suggestion. Check your app!".
  It gets triggered when a new ride suggestion is created and the target is the Passenger Main screen.

- **Title:** "New passenger joined".
  Body: "<passenger name and surname> just joined your ride!".
  It gets triggered when a passenger successfully joins a ride and the target is the Driver Main screen.

- **Title:** "Ride canceled".
  Body: "Your ride was successfully canceled".
  It gets triggered when a driver successfully cancels a ride and the target is the Driver Main screen. If passengers had joined the ride, then all of them receive a notification with:
  Title: "Ride Canceled".
  Body: "Your ride was just canceled by the driver. Please search for another".
  The driver also gets a slightly different notification with:
  Title: "Ride Canceled".
  Body: "Your ride was successfully canceled, and the following passengers were notified: <list of passengers notified>.

- **Title:** "Request canceled".
  Body: "Your ride request was successfully canceled".
  It gets triggered when a passenger successfully cancels a request and the target is the Passenger Main screen. If they had previously joined a ride, then the driver is also notified by a notification with:
  Title: "Passenger Cancelation",
  Body: "<passenger name and surname> just canceled their reservation".

- **Title:** "Scheduled ride starting soon".
  Body: "Your ride starts in 12 minutes, please get ready!".
  It gets triggered 12 minutes before the ride's start time and the target is the Passenger Main screen. The driver gets a slightly different notification with:
  Title: "Scheduled ride starting soon".
  Body: "Your ride starts in 12 minutes. If passengers joined your ride do not forget to press the Start button when you are about to depart!".
  A **data-only** notification is also sent to the driver 12 minutes before the ride starts, and includes the data of the passengers that are part of the ride. Data: passengersJson.

- **Title:** "Ride no longer available".
  Body: "The ride with id <offer_id> is no longer available.".
  It gets triggered 12 minutes before the ride's start and the target is the Passenger Main screen of the users that had the suggested ride on their screen but did not join.

- **Title:** "Ride started".
  Body: "The driver just departed from the starting point.".
  It gets triggered when the driver presses the **Start** button on their ride summary card and the target is the Passenger Main screen of the users that joined the ride.

- **Title:** "Ride did not start on time".
  Body: "Please press the Start button to confirm that your ride has begun, or press Cancel to abort it.".
  It gets triggered if five minutes after ride's start time, the driver has not yet pressed the **Start** button on the ride summary card and the target is the Driver Main screen.

- **Title:** "Ride completed".
  Body: "Enjoy the rest of the day <Firstname>!".
  It gets triggered when the driver presses the **Complete** button on their ride summary card and the target is the Driver Main screen.

- **Title:** "Ride completed".
  Body: "Enjoy the rest of the day <Firstname>!".
  It gets triggered when the passenger presses the **Complete** button on their ride summary card and the target is the Passenger Main screen.

- **Title:** "Request expired".
  Body: "Your ride request expired-please create a new one".
  It gets triggered one hour after the pickup time if the request's status is still `AVAILABLE`, and the target is the Passenger Main screen.

- **Title:** "Ride automatically canceled".
  Body: "Your ride was automatically canceled because you did not press the Start button within 15 minutes of the scheduled start time. Any passengers who joined your ride have been notified".
  It gets triggered 15 minutes after the ride's start time if the ride status is not `IN_PROGRESS` and the target is the Driver Main screen. The passengers that joined the ride get a similar notification with:
  Title: "Ride automatically canceled".
  Body: "Your ride was automatically canceled because the driver did not start within 15 minutes of the scheduled start time. Please cancel the current ride and search for another".
  The target screen is the Passenger Main screen.

- **Title:** "Ride <offer_id> completion".
  Body: "It has been an hour since the ride's start - please mark it as completed".
  It gets triggered one hour after the ride's start time if the ride status of any of the members of the coalition is still not `CANCELED/COMPLETED`. The target screen depends on the type of user who forgot to complete their ride.

- **Title:** "Ride <offer_id> autocompleted".
  Body: "Your ride was automatically marked as completed. Enjoy the rest of the day <Firstname>!"
  It gets triggered one hour and 15 minutes after the ride's start time if the ride status of any of the members of the coalition is still not `CANCELED/COMPLETED`. The target screen depends on the type of user who forgot to complete their ride and carries a **small payload** that is used to reset the application UI.

Chapter 5

User Evaluation

In this section, we present the testing phase of the application and the results of its evaluation process. Testing is taking place in the City of Chania from the 21st of July 2025, and is expected to continue until at least the end of September 2025.

5.1 Testing Phase and Issues Reported

Testing started with the Android version and with a limited group of people involved. The application was prepared, submitted, and approved by Google within two days, then it was listed on the Google Play Store as an open beta. We chose open beta because we wanted to eliminate the tester hustle of giving their email and then wait for the invitation to join, which the closed beta procedure requires. Soon after, a closed group of people joined, downloaded, and started using the application. The entire group of early testers was coordinated by Professor Georgios Chalkiadakis. Then we continued with the joint testing of the Android and iOS versions. All Android beta builds were distributed via the Google Play Console, and all iOS builds via App Store Connect. Table 5.1 provides a brief summary of the reported issues; detailed descriptions and the corresponding fixes can be found in Appendix A.

Number	Description	Appendix
1	UI layout breaks due to large font size.	A.1
2	Missing passenger ride summary card.	A.2
3	Department field displayed in Greek.	A.3
4	Missing car brands (Suzuki and KIA).	A.4

Table 5.1: Issues reported by users.

5.2 Non-Requested Features

In addition to the feedback we received, we continued to improve the TUC Ridesharing application in the best possible way, by adding features that improve the user experience and overall usability. A brief overview of all the non-requested features we added can be found in Table 5.2, and a more detailed description in Appendix B. Complete videos of the added features can be found here.

Number	Description	Appendix
1	Server-side automatic ride completion.	B.1
2	Addition of a pulsing indicator for easier navigation.	B.2
3	Addition of an FAQ option on the Profile screen.	B.3
4	Addition of an Active requests and an Active rides screen.	B.4
5	Addition of a Community screen.	B.5
6	UI adaptation for iOS 26.	B.6

Table 5.2: Non-requested features.

5.3 Usability Evaluation

Throughout the testing period, user feedback was collected. The evaluation process of the application is based on the System Usability Scale (SUS), and an online questionnaire that can be found in the Appendix C. It includes four sections:

- The user demographics.
- The 10-item SUS questionnaire.
- A section to report a problem with the app.
- A section to propose new features.

As of the end of September 2025, the total number of responders was 21. The final results will be incorporated here once the evaluation is complete, as we intend to continue the evaluation for an extended period of time.

5.3.1 User demographics section

Based on current responses, the sample is student-heavy and driver-oriented.

- The age group bar chart shows the following. About 38,1% are 22-25; 14,3% are 18-21 and another 14,3% are 51-60; 9,5% are 26-30, 31-40, 41-50; and 4,8% are 61 and over.
- Students make up 66,7% of the responders, with staff at 19% and faculty, alumni, and affiliates each at 4,8%.
- The year of study data shows that 46,2% are in their fourth year, 23,1% in their third year, and the remaining groups (11th, 8th, 6th, and 1st,) are 7,7%.
- The primary usage breakdown shows a tilt towards drivers: most application users use it as drivers (52,4%), while 47,6% use it as passengers, which is healthy for the application and for everyone searching for a ride.

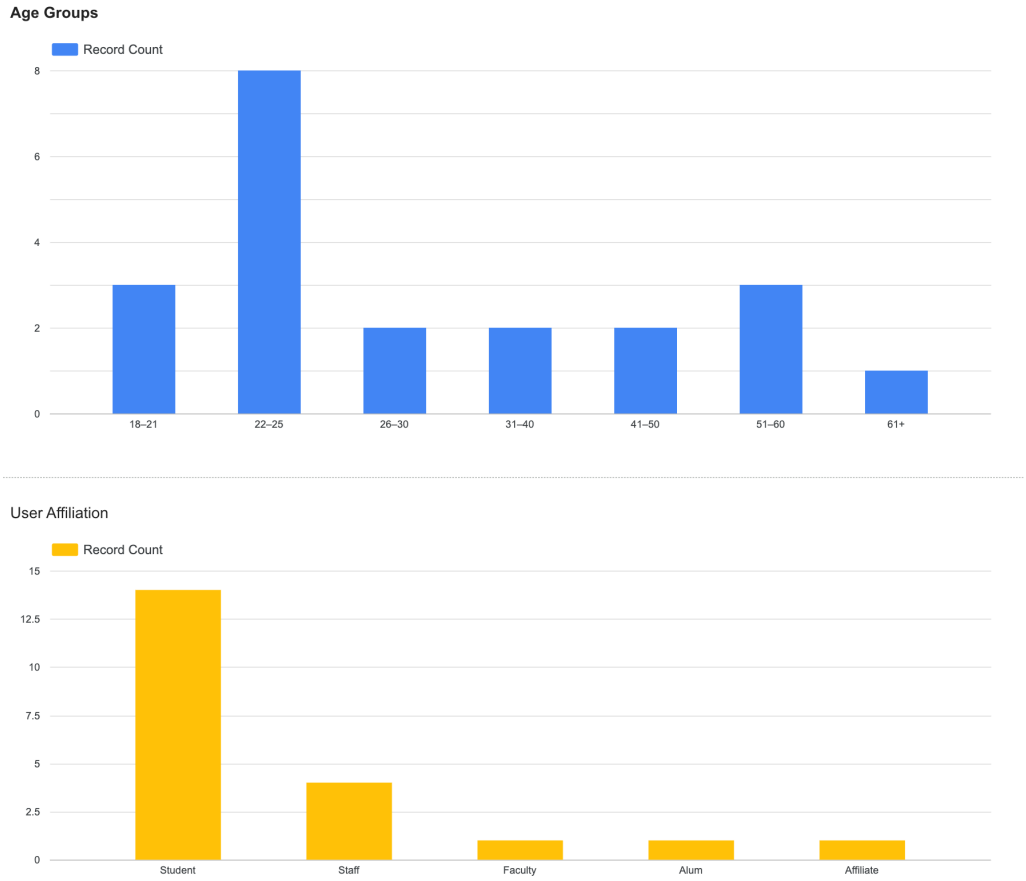


FIGURE 5.1: Age groups at the top & user affiliation at the bottom.

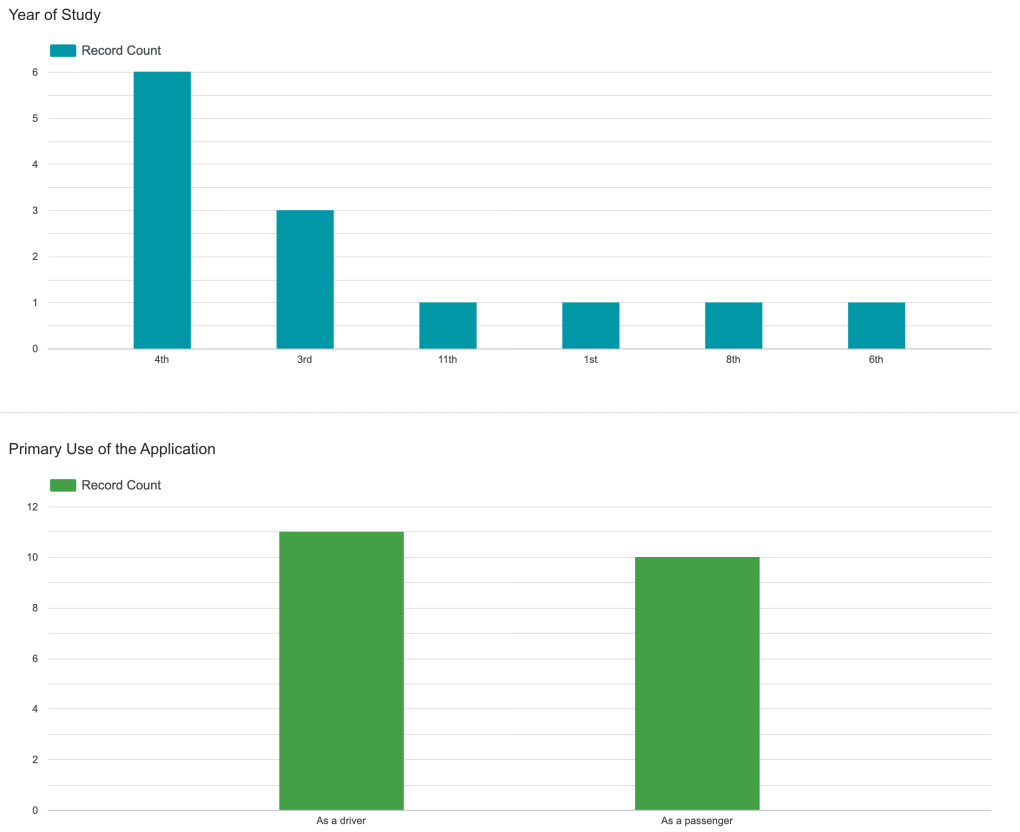


FIGURE 5.2: Year of study at the top & primary use of the application at the bottom.

5.3.2 10-item SUS questionnaire

The System Usability Scale (SUS) is an industry-standard questionnaire that is used to measure the usability of a product, application, website, or service. It consists of 10 statements rated on a five-point scale which yields a usability score of 0 to 100. The evaluation process of the application is based on the System Usability Scale (SUS) by Brooke [23]; the scores are interpreted using the adjective ratings of Bangor et al. [24]. Participants rate 10 statements on a five-point Likert scale ranging from Strongly disagree to Strongly agree.

- Q1: I think that i would like to use this application frequently.
- Q2: I found the application unnecessarily complex.
- Q3: I thought the application was easy to use.
- Q4: I think that i would need the support of a technical person to be able to use the app.

- Q5: I found the various functions in this application were well integrated.
- Q6: I thought there was too much inconsistency in this app.
- Q7: I would imagine that most people would learn to use this application very quickly.
- Q8: I found the application very cumbersome to use.
- Q9: I felt very confident using the app.
- 10: I needed to learn a lot of things before i could get going with this app.

We created the Google Form with the questionnaire and made it available to all the users, through a direct web link and via the feedback option inside the application. The results can be seen in Figure 5.3 below.

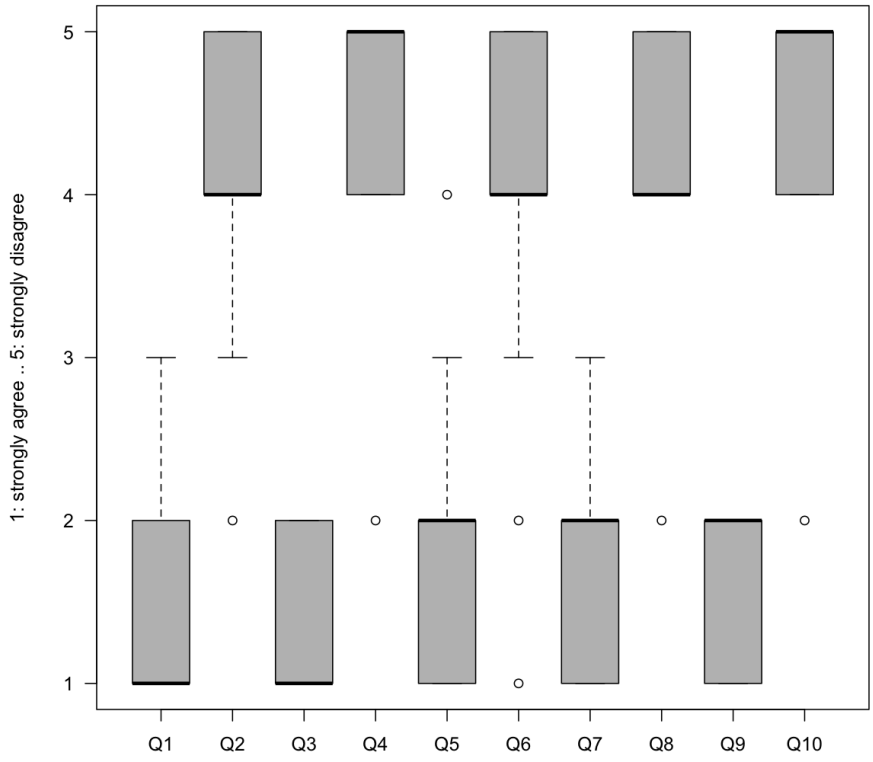


FIGURE 5.3: Responses boxplots for the 10 questions of our survey. The boxplots were created using R.

In general, users describe the application as easy to use, with an experience that feels straightforward and not overly complex. The user interface is considered easy to handle, with users accomplishing in-app tasks without much effort. The user experience is at a high level, which stems from the simplicity and predictable interactions with in-app elements. Finally, respondents suggest that onboarding is straightforward and that they feel confident using the TUC Ridesharing application. The SUS score of the application is *well above average* as the mean value of our sample is 85.0 (median: 85, standard deviation: 8.35) and can be seen in Figure 5.4.

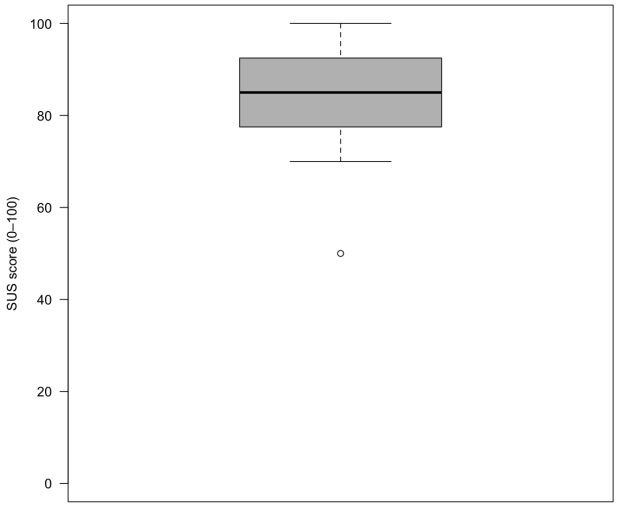


FIGURE 5.4: The boxplot of the SUS score of our evaluation.

5.3.3 Issue description section

In this section, users can report any issues they encountered while using the application. They would mention the operating system of their device, the screen on which they faced the issue, and give a brief description of it. One user suggested the creation of a document that explains the functionality of the app, but this is addressed with the FAQ page on the TUC Ridesharing website, which is mentioned in Section B.3. The same user stated that the UI needs extra work to become more user-friendly. Apart from that, zero issues were reported and the only result we can draw from this section is the OS distribution of the devices running the app: 61,9% is Android and 38,1% is iOS.

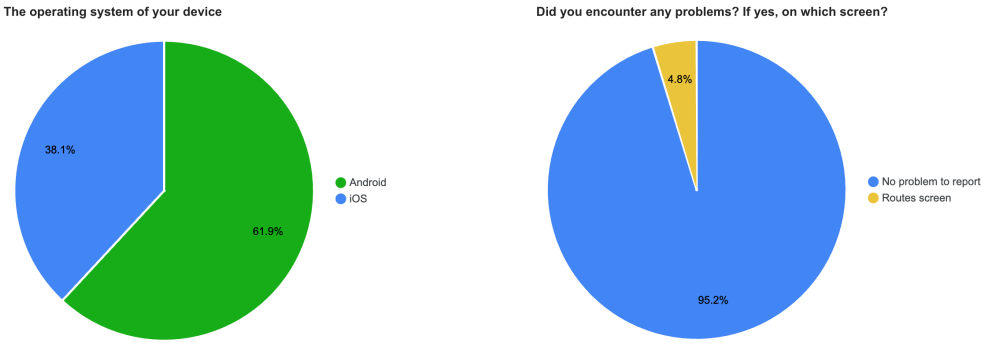


FIGURE 5.5: Issue description responses.

5.3.4 Suggestions section

The last section of the usability evaluation is reserved for user suggestions that can make the application even better. What was proposed from the responders is mostly implemented and can be found in Section 5.1 and in Appendix A. One frequent comment was that the application could use additional drivers. Suggestions that were not implemented and are considered future work, are listed below.

- Allow custom start and end of a route.
- Addition of a route to the airport.
- Addition of a route to Stavros.

Chapter 6

Conclusion and Future Work

In this thesis we showcased the social ridesharing problem of the Technical University of Crete community in the city of Chania and built a cross-platform application that attempts to address it. The main idea is to bring both passengers and drivers together in order to allow safe social ridesharing in an efficient manner. The application uses a modern and minimalist interface that is combined with a lightweight backend, that runs 24/7/365 on the TUC server infrastructure. The match-making algorithm that is used is a two-stage greedy one, which makes suggestions to the passengers and then, they join the ride that better suits their needs. This is the first time for a Greek thesis to create such an application, and to the best of our knowledge this is one of the first attempts ever for a thesis of any kind (Bachelor, Master, Doctorate) that not only prototypes, but delivers a beta tested, consumer-ready, cross-platform ridesharing application that eventually gets listed on both the App Store and the Google Play Store.

6.1 Future Work

Here we list several items for future work. Most were inspired by the day-to-day development of the TUC Ridesharing application, while others emerged through discussions with the Professors involved.

6.1.1 Implementation of cost-sharing methods

One of the key extensions proposed is the implementation of cost-sharing methods. Such methods have been extensively described in the literature, such as in the works of Bistaffa and Chalkiadakis [22, 25]. Integration into the system will provide a fair and transparent way of distributing travel costs among passengers, considering factors such as distance, fuel cost, number of passengers, and the initial cost of the trip. Furthermore, the implementation of these methods can improve the sustainability and acceptance of the system. On the one hand, drivers will feel that they are compensated for any extra costs for their rides (i.e., there will be no financial counter-incentives for them

offering the ride); and on the other hand, passengers will not feel "indebted" to the driver after being taken to their destination at no cost.

6.1.2 Multi-agent system backend integration

A second proposed extension is the integration of a multi-agent system like the one described by Pavlos Moraitis et al. in [26]. Such an action would enhance and update the backend. The server side could host different kinds of agents; like passenger, driver, and matching agents. For example, instead of having a centralized match-making algorithm that sends out data to the users, we could replace it with an agent-centric solution that will enable peer-to-peer matching, something that would reduce the load on the system. Each user can have their own personal agent that will take into account the preferences, ride history, and only attempt to match with users whose preferences and history align. This will greatly reduce the number of cancelations, and improve the overall application speed, usage, and user satisfaction. Appropriate coordination protocols between agents can further refine the details of the shared ride [27, 28].

6.1.3 Route expansion and optimization of stops

A third extension concerns further adaptation of the already existing stops, included in the implemented routes. Research should be conducted within the community, with the goal being to adapt the existing stops, and serve as many Technical University of Crete members as possible. At the same time, the creation of additional routes is proposed, if necessary, that will cover areas with high demand that are not adequately covered by the current system. This extension is expected to improve the user experience, increase user participation, and enhance its sustainability. The research can be based on usage data, as well as surveys and feedback from the community (polls on social media, discord servers, etc.).

6.1.4 Integration of alternative languages into the UI

An additional extension concerns the addition of alternative languages for the user interface (UI), starting with the Greek language. The integration of the Greek language is particularly important as the system serves the community of the Technical University of Crete, where Greek is the main language of communication. In subsequent stages, alternative languages can be added to enhance accessibility for potential foreign-speaking users, such as international students or visitors to the institution. This feature will not only improve the user experience but also promote inclusion, making the application more appealing to a wider audience.

6.1.5 Enrich the user experience on the two main screens

Another extension proposal concerns the actual ridesharing procedure. What would be desirable to be integrated would be to have a map showing the ride's complete route on both main screens. That way users can enjoy an even more modern UI and will be keener to use the application again in the future. Furthermore, when the driver presses the **Start** button on their ride summary card, in addition to the already implemented notification that informs all passengers that the vehicle has left the starting point, it could also show a pulsing indicator at the starting location. This indicator could move in real-time depending on the driver's location. The movement will be displayed on the driver's ride summary card and the ride summary card of the passengers who are part of that ride.

6.1.6 Social-score-driven Community screen

A sixth proposed extension is the addition of a Community screen that contains cards for all TUC members who use the application. Each card will include their name, surname, affiliation, department, and social score. The social score will be calculated based on each member's contributions to the community within the TUC Ridesharing application, and members will be sorted accordingly. Each card could also feature achievements based on the user's score, showcasing their credibility, trustworthiness, and dedication to other members. This feature is partially implemented with the creation of the Community screen, but it lacks the crucial social score functionality, which will provide valuable feedback to the community.

6.1.7 Option for the driver to reject passengers

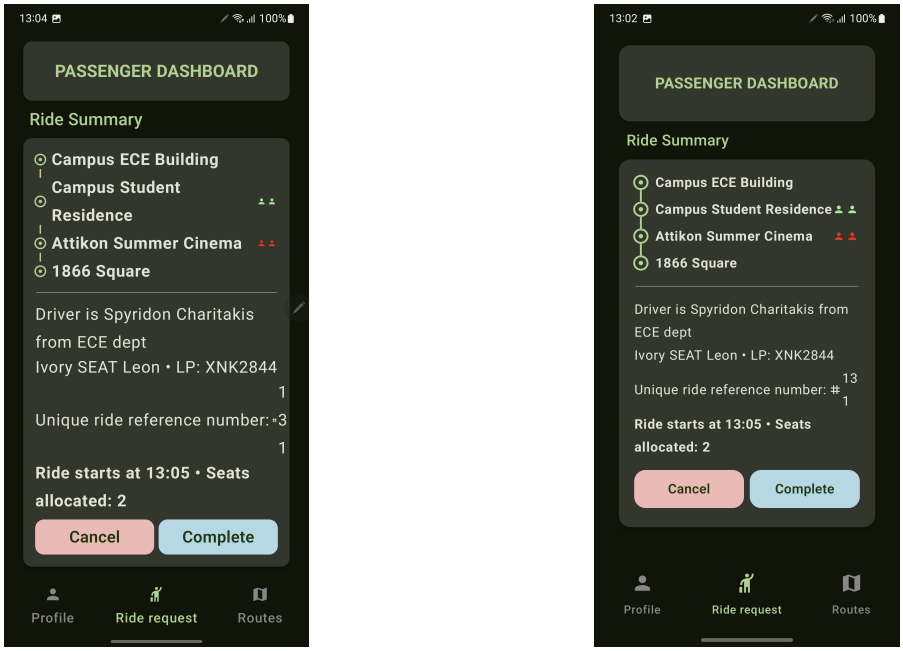
Lastly, another proposed extension concerns the addition of a feature that would allow drivers to reject a passenger. At the moment, passengers can join a ride and the driver has no control over who they will travel with. A **Reject** button can be added to the co-passenger card, which, when pressed will notify the passenger about the action and remove them from the ride. This feature will give drivers more control and encourage them to create more rides.

Appendix A

Fixes and Updates Based on User Evaluations and Feedback

A.1 UI layout breaks due to large font size

When the user was operating the application with increased font, the ridesharing cards appeared broken and the user experience was degraded. To resolve that, we capped the font size and allowed the UI to increase primarily via the 'Screen zoom' option in the device settings. By doing this, we achieved clean and correct scaling for all elements and fonts.

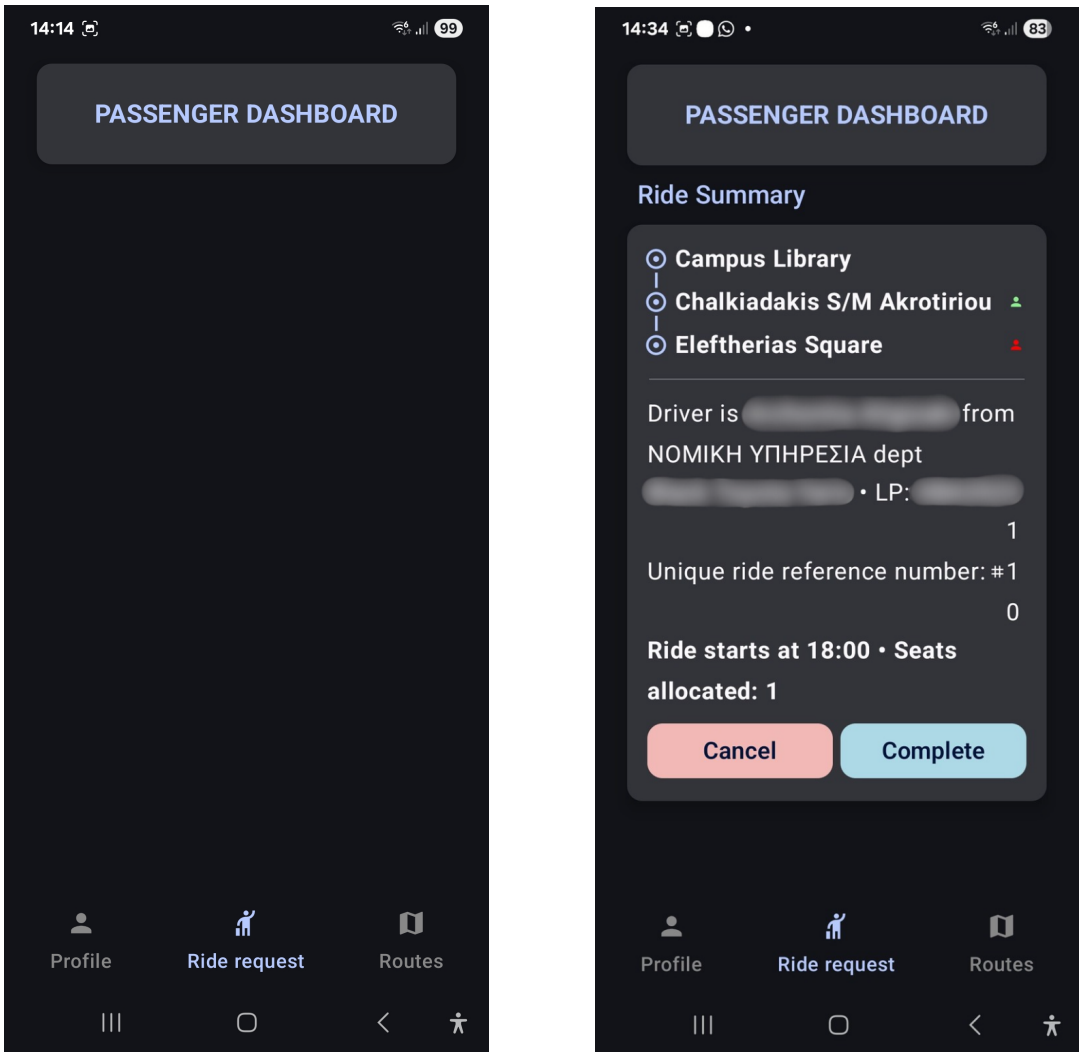


(A) (B)

FIGURE A.1: Passenger ride summary card before in (A), and after the fix in (B).

A.2 Missing passenger ride summary card

When the user was re-entering the application after the ride's start time, a certain line of code was causing the passenger ride summary card to disappear, leaving the passenger without data about the upcoming ride. This issue was found early in the testing phase when the number of testers was limited to two and was resolved the same day.



(A) (B)

FIGURE A.2: Missing ride summary card before in (A), and after the fix in (B).

A.3 Department field displayed in Greek

In various parts of the application, the user department appears. When the application was tested by a user outside the five main schools of the Technical University of Crete, their department appeared in Greek. To resolve that, we contacted TUC's IT department and received the complete department list. Then we translated it into English and incorporated it into the application.

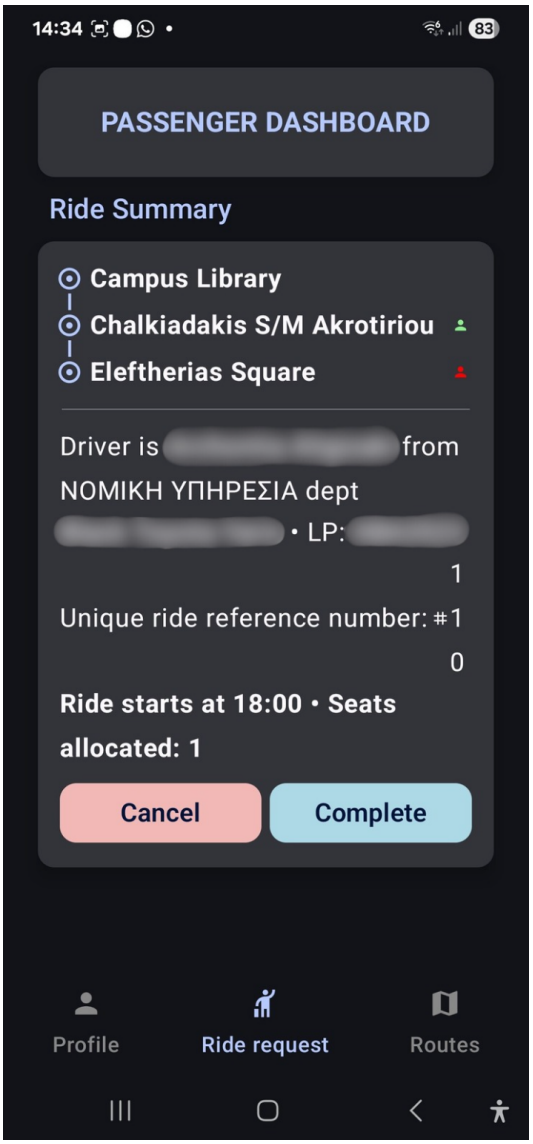
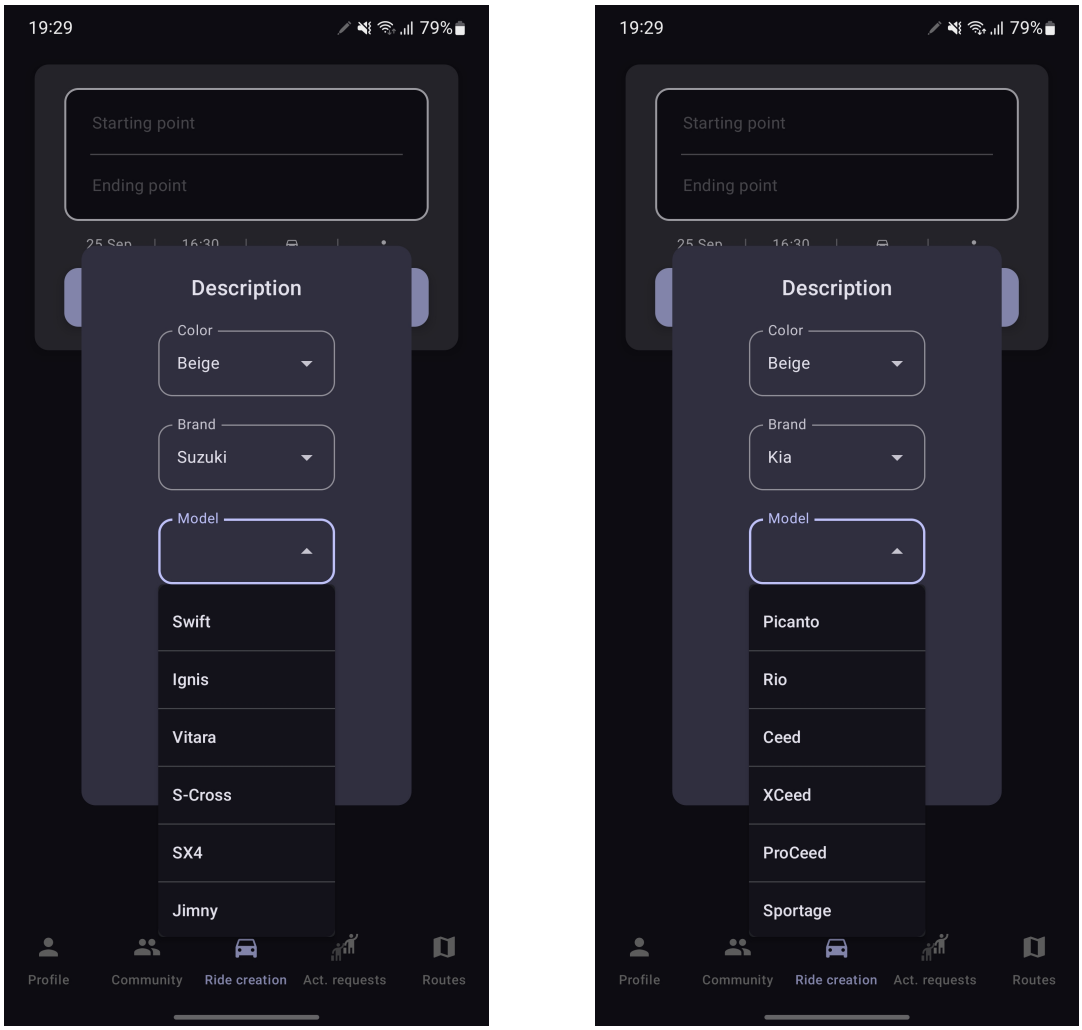


FIGURE A.3: User department in Greek before the fix.

A.4 Missing car brands (Suzuki and KIA)

Two of the users noticed that the Suzuki and KIA brands were missing from the vehicle description brand list, which could possibly prevent them from creating rides for the community. We added both brands, together with the 10 most well-sold models of each, over the past 20 years in Greece.



(A) (B)

FIGURE A.4: Addition of the Suzuki brand in (A), and the KIA brand in (B).

Appendix B

Non-Requested Features

B.1 Server-side automatic ride completion

Sometimes users forget to complete their rides, even though there are mechanisms in place to remind them. For this specific scenario, one hour and 15 minutes after the ride's start time, if the user has not yet canceled or completed their ride, a notification is sent to their device that clears their ride summary card and resets the application's UI.

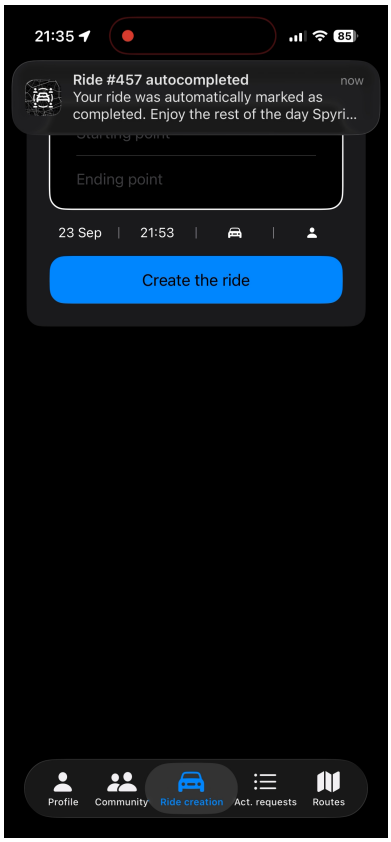


FIGURE B.1: The moment the autocompletion notification arrives, and resets the UI.

B.2 Addition of a pulsing indicator for easier navigation

The TUC Ridesharing application is capable of operating simultaneously in both driver and passenger mode. For the user to switch between the two modes, they have to exit and re-enter the application. After re-entering, they land on the Mode screen. There, next to the driver and passenger mode buttons, we added a green pulsing indicator that is active whenever a passenger request or (driver) ride is active. This makes the application easier to navigate.

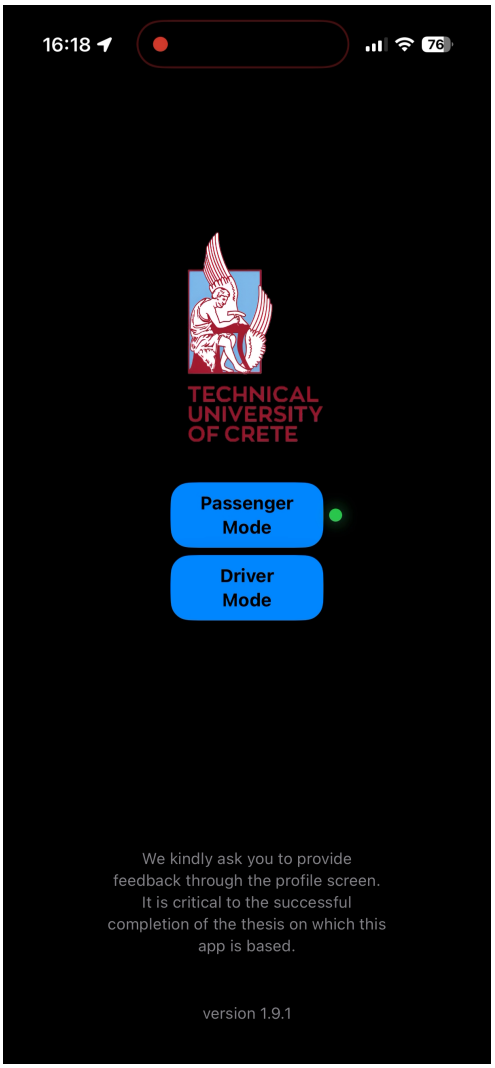
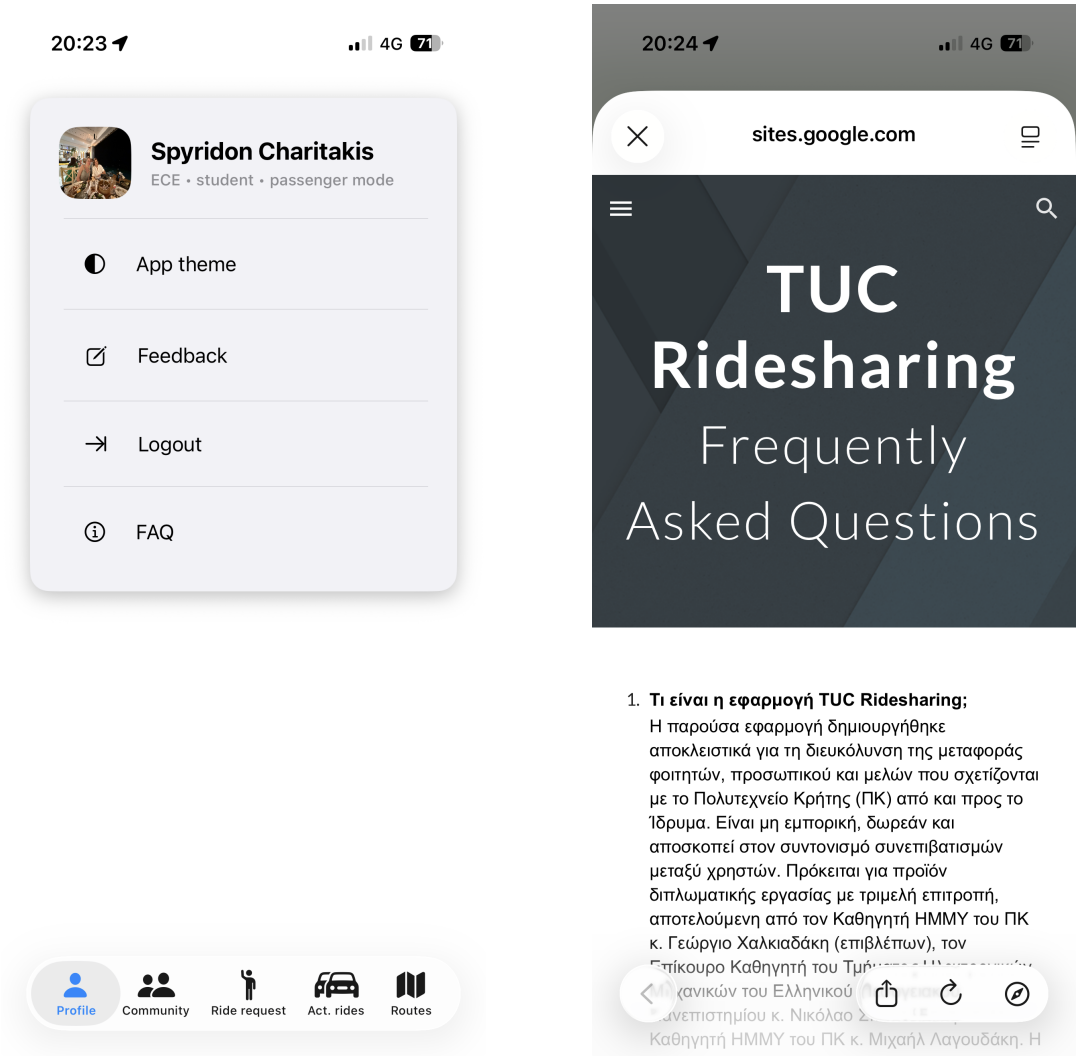


FIGURE B.2: Pulsing indicator next to the **Passenger Mode** button.

B.3 Addition of an FAQ option on the profile screen

Assistant Professor Nikolaos Spanoudakis proposed explaining to users how the application service works. This resulted in the creation of the FAQ page on the TUC Ridesharing website. This page is accessible through the Profile screen of the application.

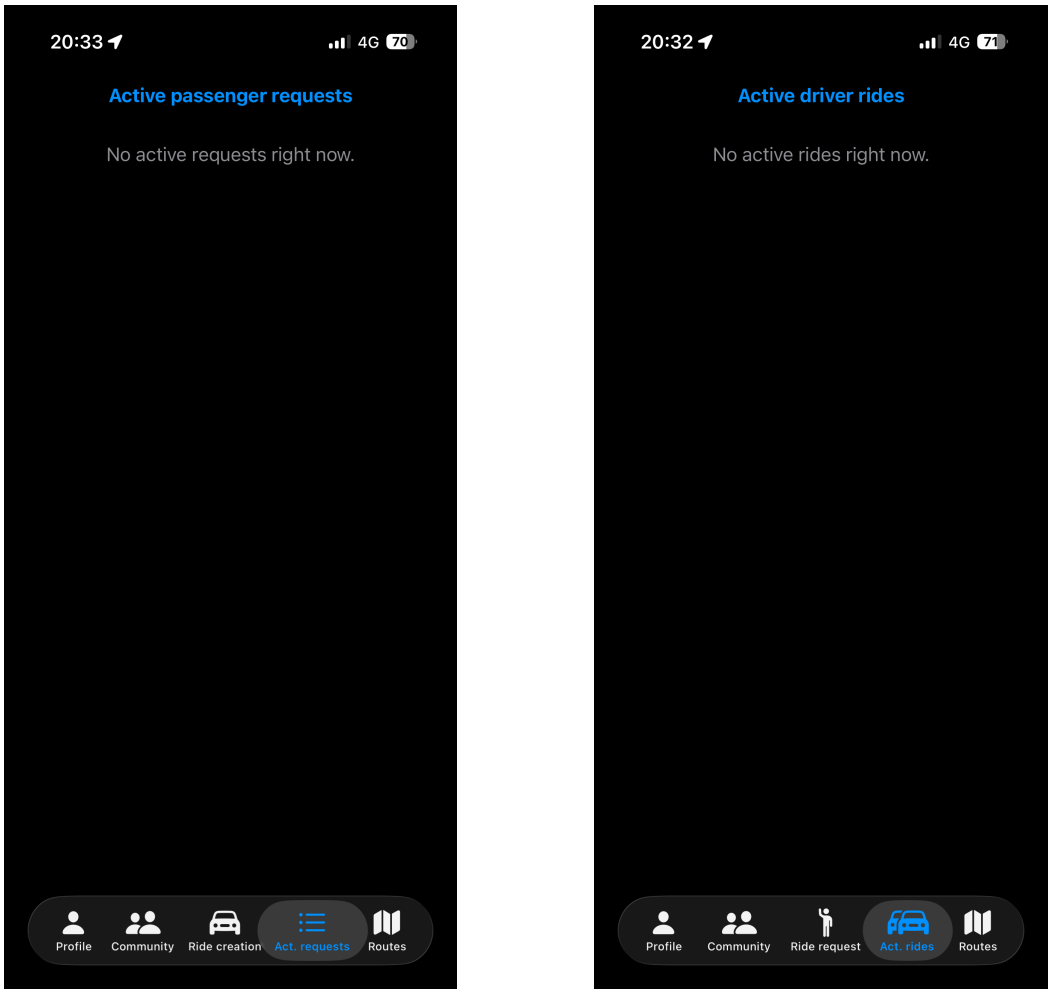


(A) (B)

FIGURE B.3: FAQ option on the Profile screen in (A), and the displayed content when pressed in (B).

B.4 Addition of an Active requests and an Active rides screen

The addition of an Active rides screen on the passenger side, together with the addition of an Active requests screen on the driver side, increases visibility on both sides, boosts application usage, and user satisfaction. The Active rides screen shows rides that have at least one available seat and start at least 15 minutes in the future. They remain visible until 12 minutes before their start time. The Active requests screen displays passenger requests that are at most 45 minutes in the past.



(A) (B)

FIGURE B.4: Active requests screen in (A), and Active rides screen in (B).

B.5 Addition of a Community screen

The integration of the Community screen, in addition to improving engagement, acts as a stepping stone on which future theses will build a very important feature, the TUC ridesharing social score. Based on that score, users will be sorted and passengers will be able to pick the highest rated driver to travel with.

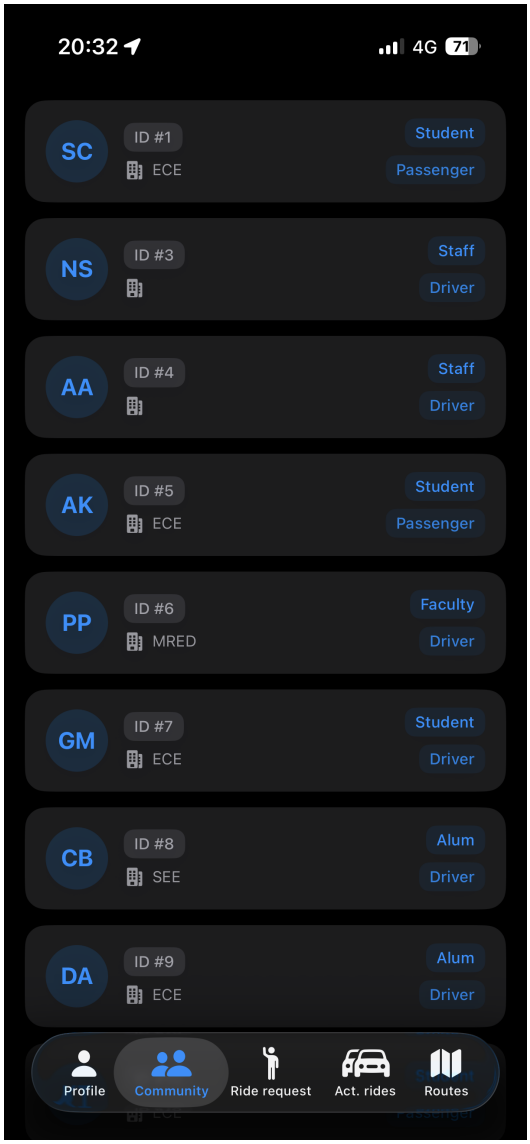
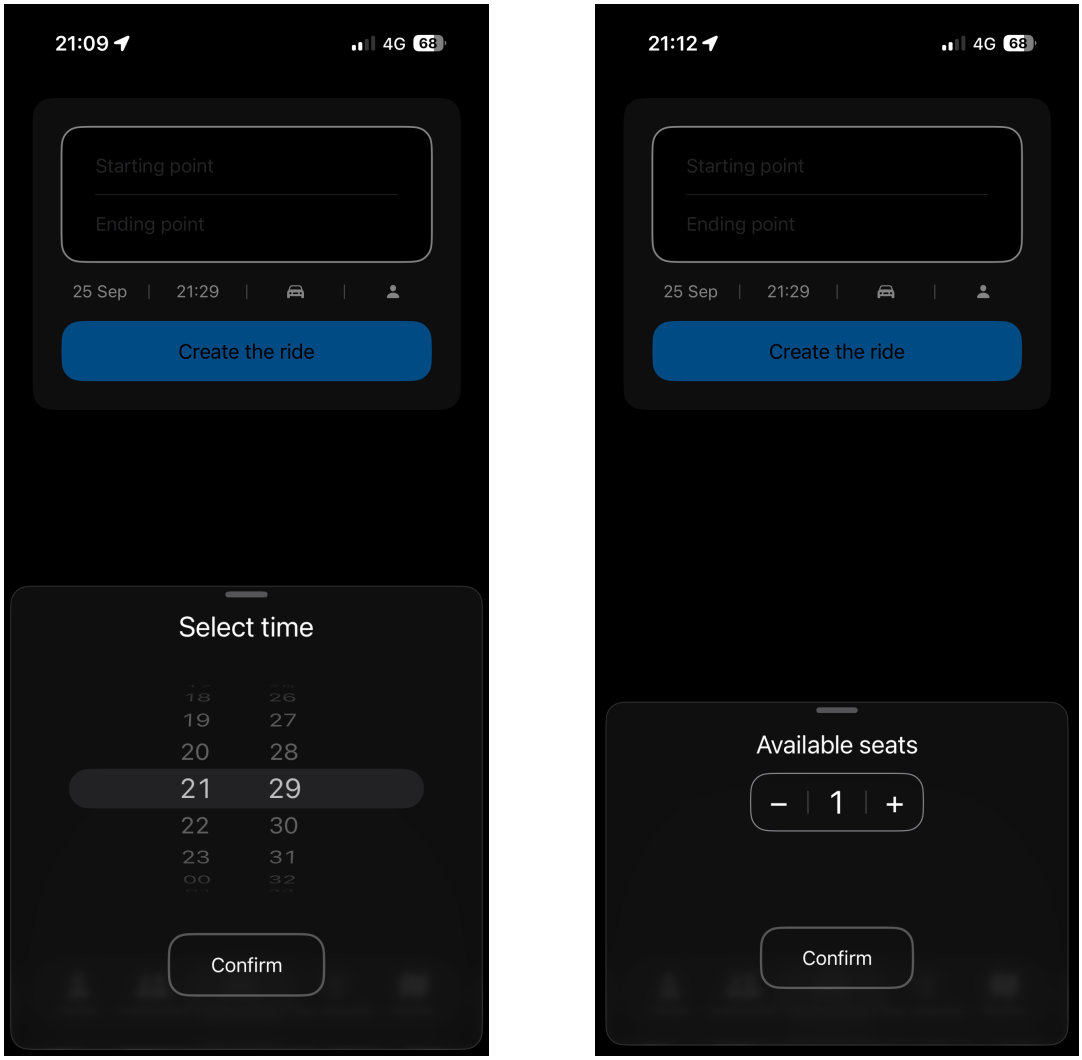


FIGURE B.5: Community screen where users are (currently) sorted by userID.

B.6 UI adaptation for iOS 26

The most significant iOS update in recent years, iOS 26, introduced many inconsistencies in the aesthetic department. Specifically, it inserted glass-like sheets below the existing ones, which degraded the user experience. We chose to embrace the new Liquid Glass design language and introduce glass-like sheets across the entire interface.



(A) (B)

FIGURE B.6: Translucent time picker sheet in (A), and translucent seat selector sheet in (B).

Appendix C

User Evaluation Form

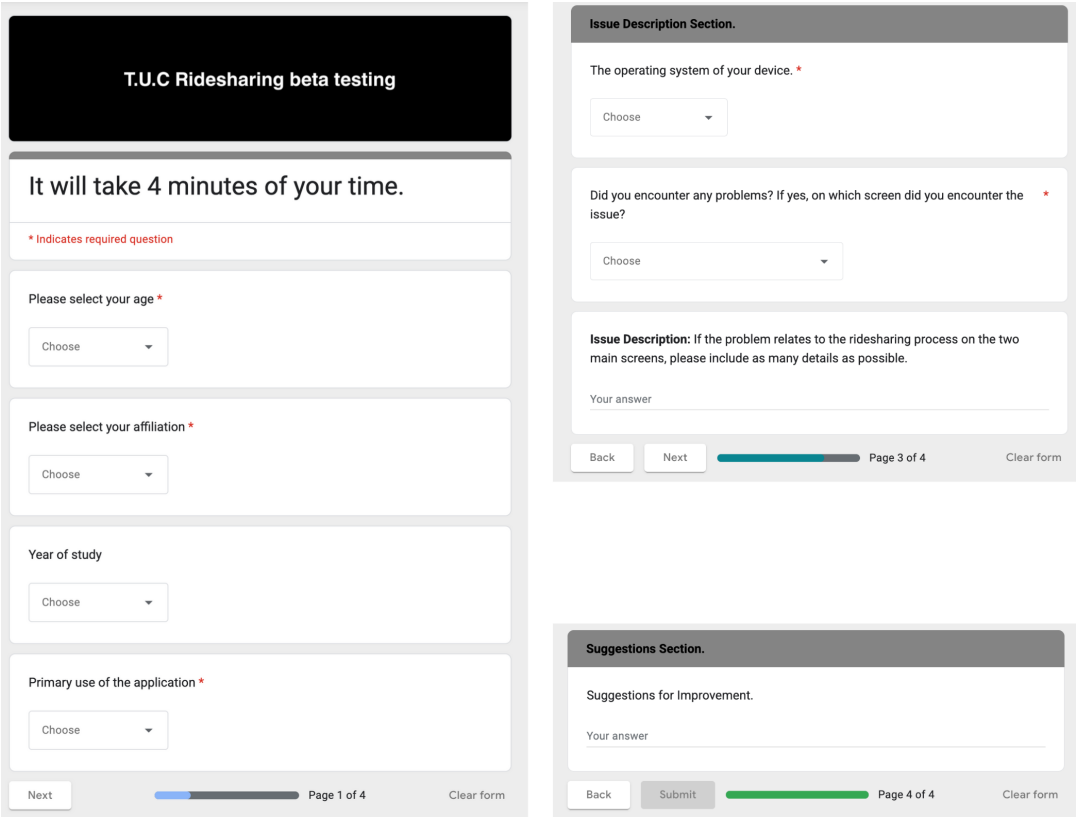
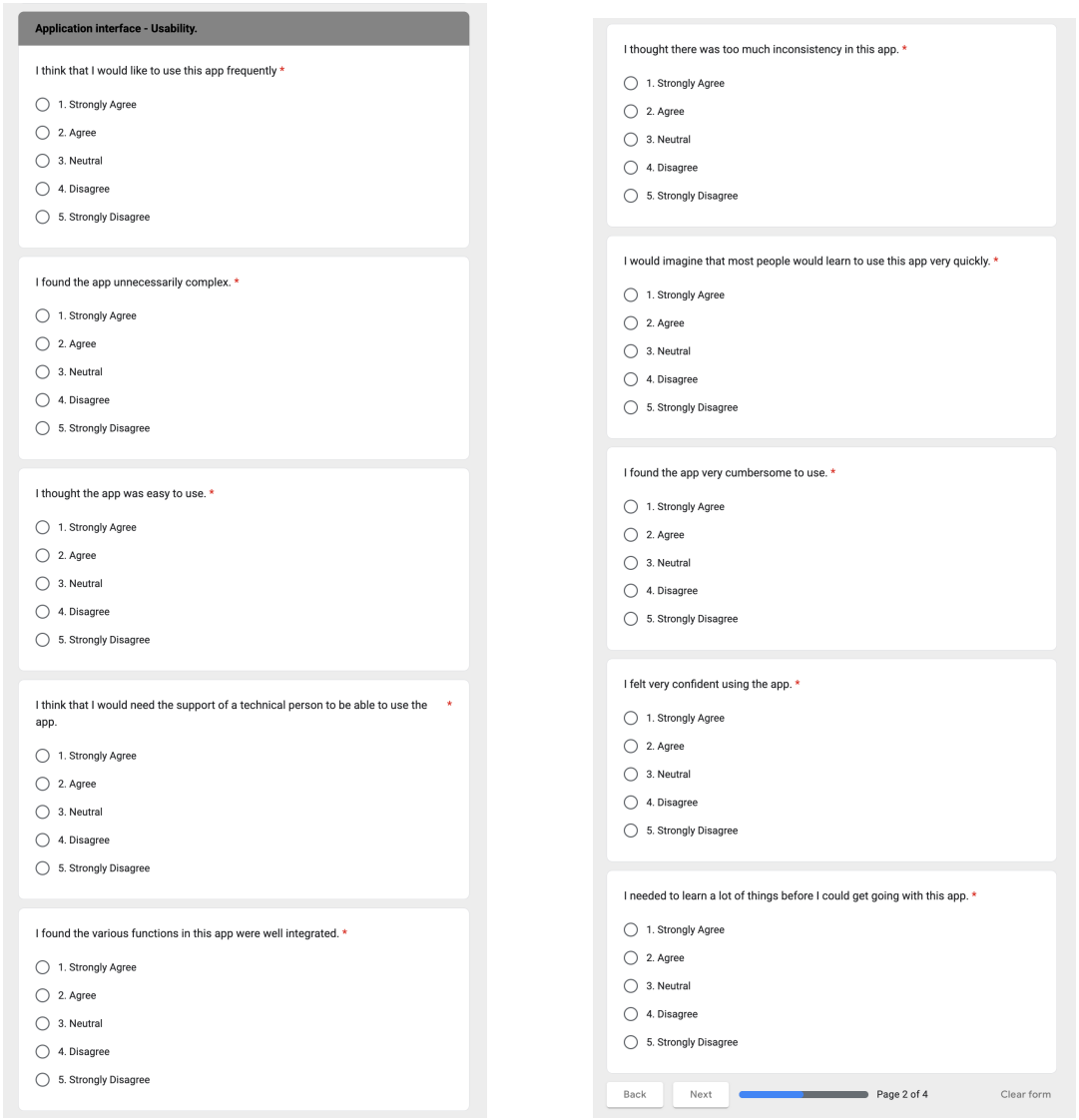


FIGURE C.1: User demographics section (left), issue description section (top right), and suggestions section (bottom right).



(A) (B)

FIGURE C.2: 10-item SUS questionnaire.

Bibliography

[1] S. Ganapati, C. G. Reddick. "Prospects and challenges of sharing economy for the public sector" (2018) (cit. on p. 1).

[2] K. Bradley, D. Pargman. "The sharing economy as the commons of the 21st century". *Cambridge Journal of Regions, Economy and Society* 10.2 (2017), pp. 231–247 (cit. on p. 1).

[3] T. Daglis. "Sharing Economy". *Encyclopedia* 2.3 (2022), pp. 1322–1332 (cit. on p. 1).

[4] E. Pagkalos. "Preferences-Aware Social Ridesharing". Senior Undergraduate Engineering Diploma Thesis. School of Electrical & Computer Engineering, Technical University of Crete, 2021 (cit. on pp. 1, 14, 15).

[5] A. Asproudi. "Preference Aggregation in the Ridesharing Domain". Senior Undergraduate Engineering Diploma Thesis. School of Electrical & Computer Engineering, Technical University of Crete, 2023 (cit. on pp. 1, 15).

[6] F. Bistaffa, C. Blum, J. Cerquides, A. Farinelli, J. Rodriguez-Aguilar. "A Computational Approach to Quantify the Benefits of Ridesharing for Policy Makers and Travellers" (2021) (cit. on pp. 1, 13, 17).

[7] J. Alonso-Mora, S. Samaranayake, A. Wallar, E. Frazzoli, D. Rus. "On-demand High-capacity Ride-sharing via Dynamic Trip-vehicle Assignment". *Proceedings of the National Academy of Sciences (PNAS)*. Washington, DC, USA: National Academy of Sciences, 2017, pp. 462–467 (cit. on p. 1).

[8] D. Bogiatzis. "Opinion research on Sustainable mobility in Chania municipality". Senior Undergraduate Engineering Diploma Thesis. School of Chemical & Environmental Engineering, Technical University of Crete, 2025 (cit. on p. 2).

[9] M. of the Interior, M. of the Digital Governance, A. M. of National Economy, Finance. *Public Services Evaluation Report - Wave 1 (May 2025)*. 2025 (cit. on p. 2).

[10] R. W. Schvaneveldt, F. T. Durso, D. W. Dearholt. "Graph Theoretic Foundations of Pathfinder Networks". *Computers & Mathematics with Applications*. Elmsford, NY, USA: Pergamon Press, 1988, pp. 437–442 (cit. on p. 6).

[11] S. S. Ray. *Graph Theory with Algorithms and its Applications*. 2012 (cit. on pp. 6, 7).

[12] J. A. Bondy, U. S. R. Murty. *Graph Theory with Applications*. 1976 (cit. on pp. 7, 8).

[13] N. Deo. *Graph Theory with Applications to Engineering & Computer Science*. 1974 (cit. on p. 9).

[14] G. Chalkiadakis, E. Elkind, M. Wooldridge. *Computational Aspects of Cooperative Game Theory*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, Oct. 2011 (cit. on p. 9).

[15] O. Shehory, S. Kraus. "Feasible formation of coalitions among autonomous agents in non-super-additive environments" (1998) (cit. on p. 10).

[16] T. Sandholm, K. Larson, M. Andersson, O. Shehory, F. Tohme. "Anytime Coalition Structure Generation with Worst Case Guarantees" (1999) (cit. on p. 10).

[17] S. Kraus, O. Shehory, G. Taase. "Coalition Formation with Uncertain Heterogeneous Information". *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. Melbourne, Australia: ACM, 2003, pp. 1–8 (cit. on p. 11).

[18] G. Chalkiadakis. "A Bayesian Approach to Multiagent Reinforcement Learning and Coalition Formation under Uncertainty". PhD Thesis. Graduate Department of Computer Science, University of Toronto, 2007 (cit. on pp. 11, 17).

[19] R. Alam, A. Jamal, A. Chowdhury. "Design and implementation of an environment friendly ride sharing platform for North South University Students" (2016) (cit. on p. 12).

[20] D. Sanghvi. "Ride Sharing Website" (2019) (cit. on p. 13).

[21] P. Santi, G. Resta, M. Szell, S. Sobolevsky, S. H. Strogatz. "Quantifying the benefits of vehicle pooling with shareability networks" (2014) (cit. on p. 13).

[22] F. Bistaffa, A. Farinelli, G. Chalkiadakis, S. D. Ramchurn. "A cooperative game-theoretic approach to the social ridesharing problem" (2017) (cit. on pp. 14, 17, 76).

[23] J. Brooke. "SUS: A 'Quick and Dirty' Usability Scale". *Usability Evaluation in Industry*. Ed. by P. W. Jordan, B. Thomas, I. L. McClelland, B. Weerdmeester. Taylor & Francis, 1996, pp. 189–194 (cit. on p. 71).

[24] A. Bangor, P. T. Kortum, J. T. Miller. "Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale". *Journal of Usability Studies* 4.3 (2009), pp. 114–123 (cit. on p. 71).

[25] F. Bistaffa, A. Farinelli, G. Chalkiadakis, S. D. Ramchurn. "Recommending fair payments for large-scale social ridesharing" (2015) (cit. on p. 76).

[26] P. Moraitis, E. Petraki, N. I. Spanoudakis. "Providing Advanced, Personalised Infomobility Services Using Agent Technology" (2003) (cit. on p. 77).

[27] N. Spanoudakis, P. Moraitis. "The ASEME Methodology". *International Journal of Agent-Oriented Software Engineering*. Geneva, Switzerland: Inderscience Publishers, 2022, pp. 79–107 (cit. on p. 77).

[28] N. Spanoudakis, P. Moraitis. "An Agent Modeling Language Implementing Protocols through Capabilities". *Proceedings of the 2008 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT-08)*. Sydney, Australia: IEEE/ACM, 2008, pp. 578–582 (cit. on p. 77).