

NED UNIVERSITY OF ENGINEERING AND TECHNOLOGY



PROJECT TITLE:

UniRide

**DATA STRUCTURES AND ALGORITHMS
(CT-195)**

COURSE INSTRUCTOR: DR. MUHAMMAD KAMRAN

ARSALA KHAN (CT-24218)

JAVERIA SOOMRO (CT-24216)

RIDA RAFFIQUE (CT-24248)

Table of contents

Abstract

- 1. Problem Understanding**
- 2. Requirement Identification**
- 3. Solution Overview**
- 4. DSA Concepts and Design Implementation**
- 5. UI Structure**
- 6. Testing, Challenges and Results**
- 7. User Reviews**
- 8. Future Scope**
- 9. Conclusion**

Abstract

UniRide is a university-focused ride-sharing platform designed to reduce student commuting challenges through verified, route-based ride matching. The system authenticates users via university credentials and matches rides using graph-based route analysis combined with preference filtering. Core data structures including graphs, queues, and hash maps ensure efficient matching and fair request processing. Testing demonstrates successful user authentication, reliable ride matching, and stable chat functionality. The platform provides a secure, cost-effective transportation solution for the university community.

1. Problem Understanding

1.1 Background and Motivation

University students across Pakistan face significant daily commuting challenges that impact both their finances and academic schedules. A 2023 survey of university students in Karachi revealed that transportation costs consume 15-25% of monthly allowances, while unpredictable public transport causes students to miss an average of 2-3 classes per week. Most students rely on expensive ride-hailing services like Careem and Uber, informal carpool arrangements through WhatsApp groups, or overcrowded public buses with inconsistent timing.

The core problem is simple: dozens of students travel similar routes daily, yet have no efficient way to coordinate shared rides within a trusted, verified community. Existing ride-sharing platforms are designed for general public use and lack critical features university students need—verified student-only access, route-based matching for campus-specific locations, and built-in coordination tools that respect safety and privacy concerns.

UniRide addresses this gap by creating a closed, university-verified ecosystem where students can confidently share rides with peers traveling similar routes at similar times.

1.2 The Commuting Challenge: A Data Structures Problem

At its core, efficient ride-sharing is a **computational matching problem** that requires:

- **Fast lookups** to find compatible rides among hundreds of options
- **Route connectivity analysis** to determine if two locations can be connected
- **Fair request handling** when multiple students want the same ride
- **Efficient message passing** between ride participants

These challenges map directly to fundamental data structure problems: graph traversal for route matching, queue-based fairness for request processing, hash table lookups for user authentication, and ordered storage for chat history.

This project demonstrates how choosing the right DSA concepts can transform a real-world social coordination problem into an efficient, scalable system.

2. Requirement Identification

2.1 Formal Problem Statement

Design and implement a university ride-sharing platform that:

1. Restricts access to verified university students
2. Enables students to create ride offers or search for available rides
3. Matches rides using graph-based route connectivity within reasonable proximity
4. Manages join requests fairly using first-come-first-served processing
5. Facilitates coordination through an integrated chat system for confirmed ride members
6. Persists all data (users, rides, requests, messages) using a structured database backend

Success criteria: The system must authenticate users correctly, match compatible rides accurately, process requests fairly, and enable seamless communication—all while maintaining performance suitable for real-time university usage.

2.2 Functional Requirements

User Authentication & Verification – Register via university email, validate enrollment numbers, restrict platform access to verified students only.

Ride Management – Post rides with origin, destination, time, capacity, preferences; view and manage pending join requests; update ride status.

Ride Discovery & Matching – Search rides by route; filter by compatibility, available seats, preferences, and departure time.

Join Request System – Send join requests; process in FIFO order; accept/reject with messages; auto-update capacity and ride status.

In-Ride Communication – Exchange messages among confirmed ride participants; chronological display; access restricted to ride members.

Data Persistence – Store users, rides, requests, and messages in database; maintain consistency across sessions.

2.3 Non-Functional Requirements

Performance – Fast user authentication and ride retrieval; efficient search operations; minimal latency for messaging.

Scalability – Support growing user base and ride volume; accommodate new locations and features without redesign.

Reliability – Validate all inputs; handle edge cases gracefully; prevent data corruption; ensure database consistency.

Security – Secure password storage; session management with expiration; prevent unauthorized access and injection attacks.

Usability – Intuitive interface; minimal steps for common tasks; clear success/error feedback; straightforward processes.

Maintainability – Modular design with separation of concerns; well-documented code; extensible database schema.

2.4 System Constraints and Assumptions

Assumptions:

1. **User Base:** All users are university students; faculty integration is future work
2. **Location Coverage:** System uses predefined set of 50+ common Karachi locations based on student residential patterns and campus proximity
3. **Route Model:** Direct route connectivity represented in graph; multi-hop routing not implemented
4. **Time Handling:** Rides scheduled for same-day; no advance booking system
5. **Concurrency:** SQLite handles typical concurrent access; migration to PostgreSQL needed for 10,000+ concurrent users
6. **Internet Connectivity:** Assumes stable internet for database operations; no offline mode

Limitations:

1. **No Real-Time GPS:** Current version does not track live vehicle location or provide ETAs
2. **Static Routes:** Graph must be updated manually to add new locations
3. **No Payment Integration:** System handles coordination only; payment settled offline
4. **Single University:** Designed for NED University; multi-institution support requires authentication redesign
5. **Basic Matching:** Uses proximity only; no machine learning for preference prediction
6. **No Ride History:** Past completed rides not used for user reputation or recommendations

3. Solution Overview

3.1 Core Approach

UniRide transforms the ride-sharing coordination problem into a **graph-based matching system** backed by efficient data structures:

Step 1: Authentication → Students verify identity using university email and enrollment database; session tokens stored in hash map for O(1) validation

Step 2: Ride Creation → Drivers post rides; data stored in vector for flexible management and fast iteration

Step 3: Intelligent Matching → System constructs location graph where:

- Nodes = pickup/dropoff locations
- Edges = feasible routes with distance weights
- Search queries traverse graph to find connected paths within threshold (4km)

Step 4: Fair Request Handling → Join requests enqueued (FIFO); capacity checked before acceptance; ride status updated dynamically

Step 5: Secure Communication → Chat messages stored per ride ID using deque for chronological ordering; access restricted to confirmed participants

Key Insight: By modeling locations as a graph and leveraging appropriate data structures for each operation, the system achieves both correctness (only valid matches shown) and efficiency (fast search even with many users).

3.2 Architecture

The UniRide platform follows a **three-tier architecture** separating presentation, business logic, and data persistence:

Frontend Layer (React + TypeScript): Built using React 18, TypeScript, Vite, and Tailwind CSS for responsive, type-safe user interface. Provides authentication forms, ride creation/search interfaces, request management dashboard, and real-time chat. Communicates with backend via HTTP/HTTPS using JSON payloads.

Backend Layer (C++ with Crow Framework): Core business logic implemented in C++ using Crow web framework for HTTP routing and JSON handling. Contains four key modules: Authentication (session token management via hash maps), Ride Management (vector storage with graph-based matching), Request Handler (FIFO queue processing), and Chat System (deque-based message ordering). Exposes RESTful API endpoints for frontend communication.

Database Layer (SQLite): Persistent storage with tables for users, rides, requests, messages, locations, connections, and enrollment verification. Uses parameterized queries for security, transactions for consistency, and indexes for optimized lookups.

Development Environment: Visual Studio Code with Git/GitHub for version control enabling collaborative development and code tracking.

3.3 How the Solution Addresses the Problem

- Student-only verified system ensures safety and trust.
- Graph-based ride matching shows only relevant rides, no manual filtering needed.
- Queue-based request handling and dynamic ride capacity ensure fairness.
- Single platform handles discovery, coordination, and communication; no app-switching.
- Shared rides reduce transportation costs and vehicle usage.
- Modular design allows future features like GPS tracking, pre-booking, or faculty access.

4. DSA Concepts and Design Implementation

4.1 Core Data Structures

- **Vectors:** Store dynamic collections such as users, rides, and requests; allow flexible and fast access.
- **Queues:** Handle ride join requests in order, ensuring fairness.
- **Graphs:** Represent locations and routes for efficient ride matching.
- **Hash Maps:** Enable quick lookup of users, rides, and connections.
- **Deque:** Manage chat messages efficiently in chronological order.

4.2 Key Algorithmic Ideas

- **Ride Matching:** Uses a weighted location graph to show only compatible rides based on start/end proximity within a defined radius (e.g., 4 km).
- **Join Request Handling:** Queues ensure first-come, first-served processing for fairness.
- **Efficient Lookup:** Hash maps enable fast retrieval of rides, users, and session data, improving system responsiveness.

4.3 Design Rationale

- **Vectors vs. Linked Lists:** Vectors are preferred for indexed access and iteration over frequently accessed data.
- **Queues vs. Vectors:** Queues guarantee FIFO processing of join requests without additional logic.

- **Adjacency List Graph vs. Matrix:** Sparse graph makes adjacency lists memory-efficient and fast for traversal.
- **Deques for Chat:** Fast append/remove operations on both ends support efficient chat history management.

4.4 Key Functions

1. Location Graph Management

- **bool loadomDatabase(dbPath):** Builds a weighted graph using an `unordered_map` and adjacency list structure to store location connections efficiently.
- **bool areConnected(area1, area2):** Uses adjacency list lookup and linear traversal to check if two locations are directly connected.

2. Ride System

- **void addRide(id, from, to, time, mode):** Uses a vector to dynamically store and manage the list of active rides.
- **vector<Ride> findMatches(from, to, rideType, userID):** Iterates through a vector of rides and uses a graph (adjacency list) to efficiently check location proximity for matching.
- **vector<Ride> getAllRides():** Returns the internal vector used to store all rides for easy traversal and processing.

3. Request Queue

- **vector<int> createRequest(userID, from, to, rideType):** Uses a vector to store created request IDs and a queue to manage ride requests in a FIFO manner for fair processing.
- **bool respondToRequest(requestID, accept, outMessage):** Uses a queue to sequentially search, remove, and update specific requests while preserving the remaining order.
- **vector<int> createRideOffer(userID, from, to, rideType):** Uses a vector for structured return of created IDs and integrates with graph-based matching logic for efficient ride creation.
- **listPending():** Uses a queue to iterate through pending requests in order and converts them into a structured JSON response.

4. Chat Feature

- **bool SetRideLead(rideID, leadUserID):** Uses an `unordered_map` to map each ride ID to its lead user, allowing fast O(1) average lookup and updates.

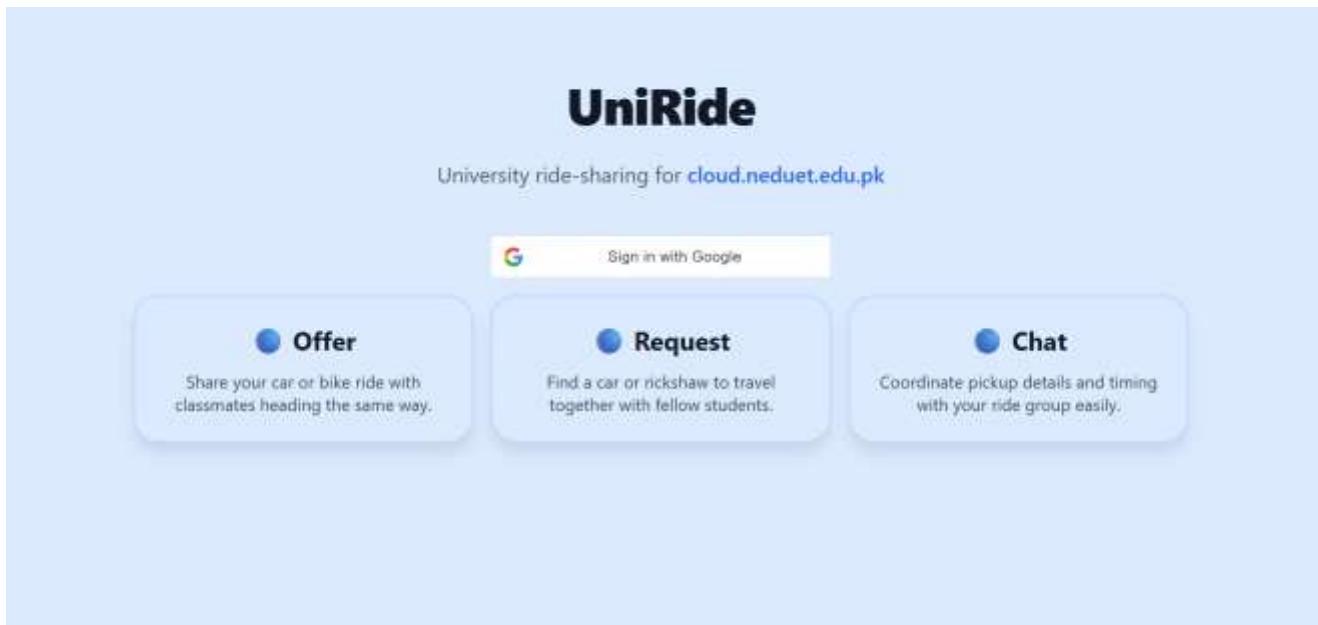
- **bool AddMessage(sender, recipient, text, rideID):** Uses an unordered_map to quickly access chat data for a ride using its rideID, and a deque to efficiently store and append chat messages in order.

5. Authentication System

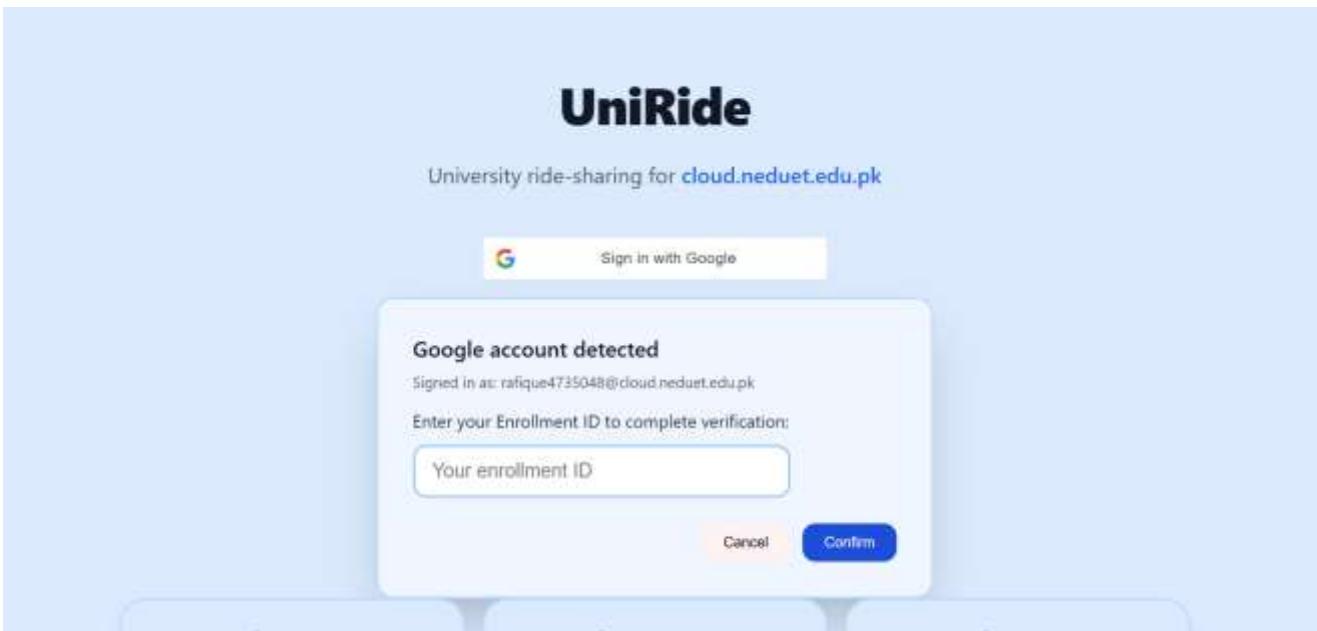
- **string storeSessionToken(userID):** Uses an unordered_map to store session tokens and map them to user IDs for fast O(1) average access.
- **bool validateSessionToken(token, userID):** Uses an unordered_map to efficiently validate session tokens through quick key-based lookup.

5. UI Structure

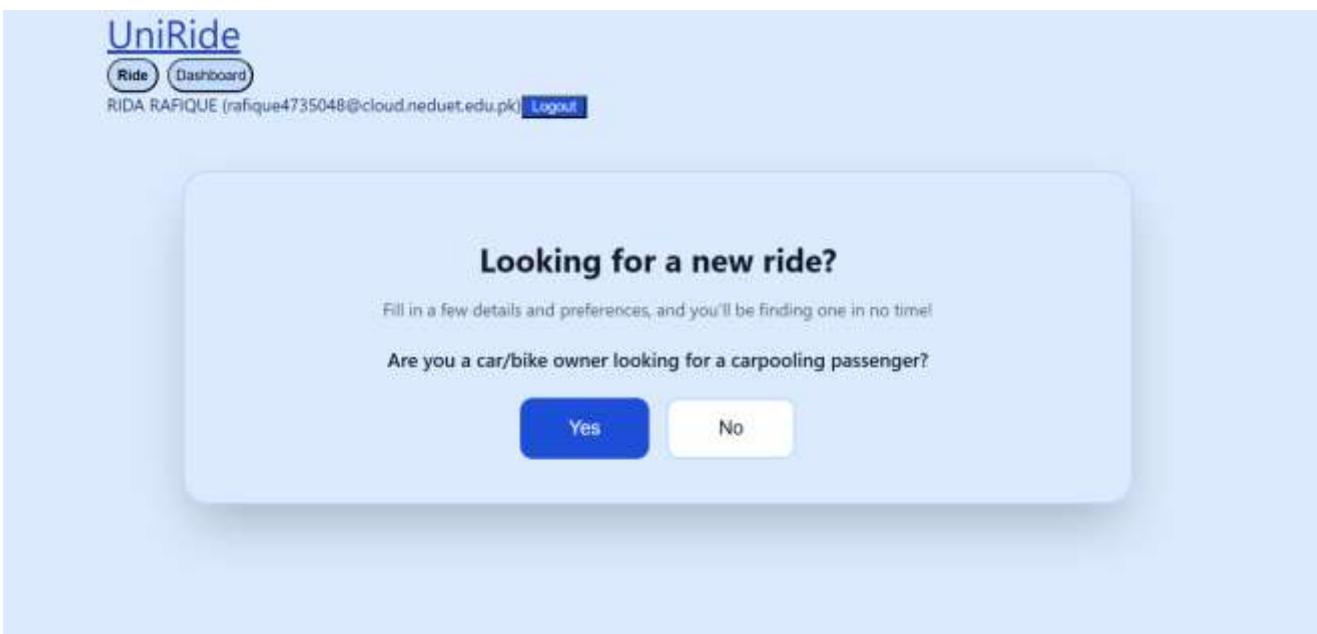
Landing Page:



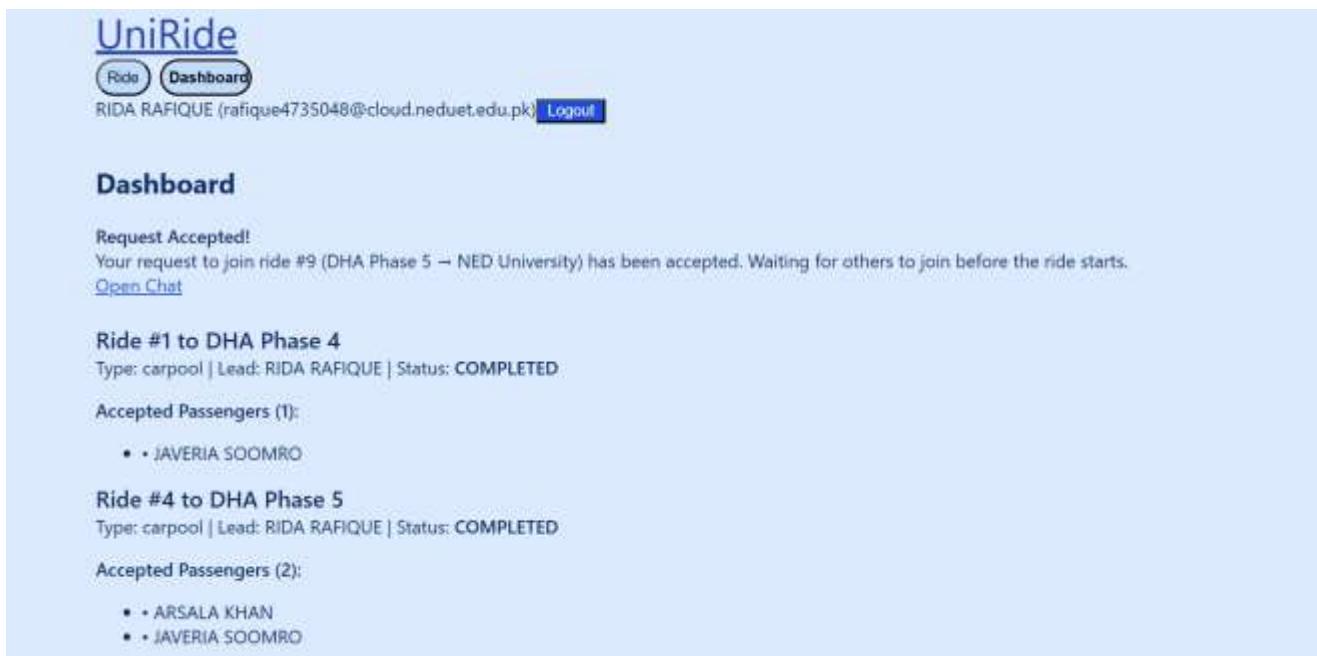
Verification through enrolment ID:



Create Ride:

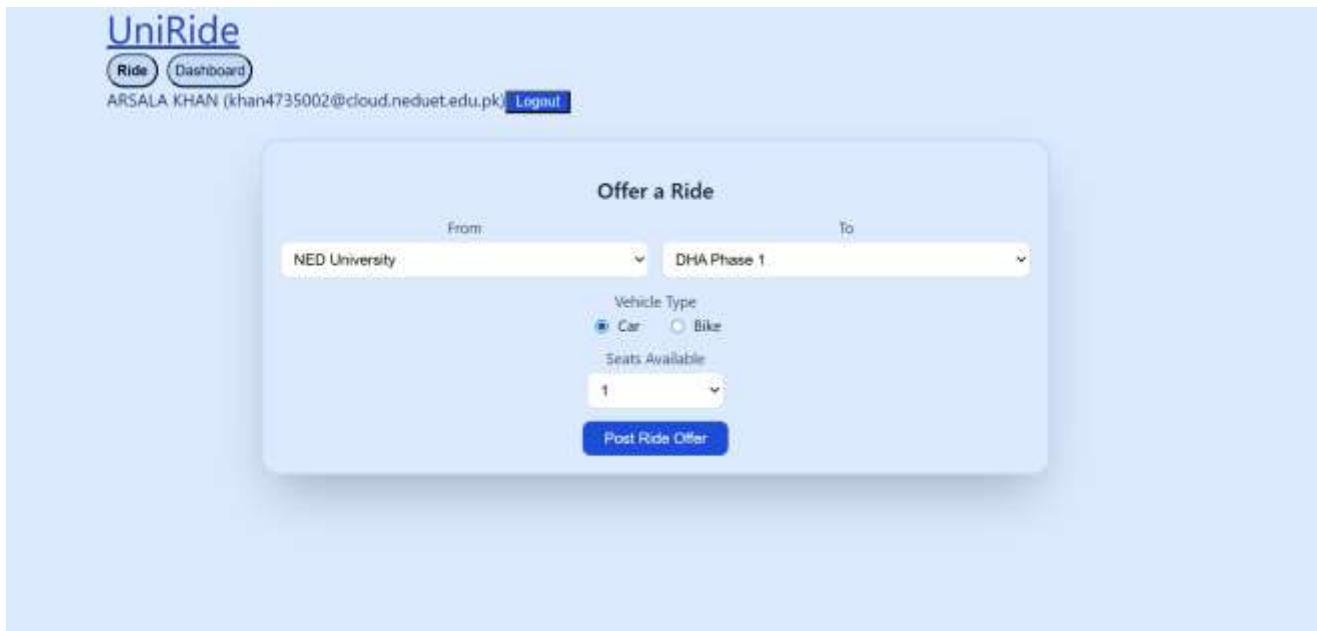


Dashboard:



The screenshot shows the UniRide dashboard. At the top, there are two buttons: "Ride" and "Dashboard". The "Dashboard" button is highlighted with a blue border. Below the buttons, the user information is displayed: RIDA RAFIQUE (rafiqe4735048@cloud.neduet.edu.pk) and a "Logout" button. A success message "Request Accepted!" is shown, stating: "Your request to join ride #9 (DHA Phase 5 → NED University) has been accepted. Waiting for others to join before the ride starts." Below this, there is a link "Open Chat". Under the heading "Ride #1 to DHA Phase 4", it says "Type: carpool | Lead: RIDA RAFIQUE | Status: COMPLETED". It lists one passenger: JAVERIA SOOMRO. Under the heading "Ride #4 to DHA Phase 5", it says "Type: carpool | Lead: RIDA RAFIQUE | Status: COMPLETED". It lists two passengers: ARSALA KHAN and JAVERIA SOOMRO.

Offer a ride:



The screenshot shows the "Offer a Ride" form. At the top, there are two buttons: "Ride" and "Dashboard". The "Dashboard" button is highlighted with a blue border. Below the buttons, the user information is displayed: ARSALA KHAN (khan4735002@cloud.neduet.edu.pk) and a "Logout" button. The main form has fields for "From" (NED University) and "To" (DHA Phase 1). Under "Vehicle Type", there are two radio buttons: "Car" (selected) and "Bike". There is also a dropdown for "Seats Available" with the value set to 1. At the bottom of the form is a blue "Post Ride Offer" button.

Request for a ride:

The screenshot shows the UniRide application's "Request a Ride" feature. At the top, there are two dropdown menus: "From" set to "NED University" and "To" set to "Clifton Block 4". Below these are three checkboxes for vehicle types: "Bike" (unchecked), "Rickshaw" (checked), and "Car" (checked). A blue "Search for Rides" button is located below the checkboxes. A modal window titled "No matching rides found!" is displayed, containing the text "Choose a fallback ride type to post:" and two buttons: "Rickshaw" and "Car". At the bottom of the modal is a green "Post Fallback Ride" button and a "Cancel" link.

Chat:

The screenshot shows the UniRide application's chat interface for "Chat for Ride #1". The messages are as follows:
RIDA RAFIQUE: Tue Nov 18 15:58:31 2025
HELLO
You: Wed Nov 19 15:40:10 2025
hi
You: Wed Nov 19 15:40:15 2025
u ready?
RIDA RAFIQUE: Wed Nov 19 15:40:23 2025
yep coming in 5

At the bottom, there is a text input field with placeholder text "Type a message..." and a purple "Send" button.

6. Testing, Challenges, and Results

The UniRide system underwent comprehensive testing to verify functional correctness and identify implementation challenges. This section documents the testing approach, technical obstacles encountered during development, solutions implemented, and final results.

6.1 Functional Testing

Each core feature was tested using multiple test accounts to verify correct behavior:

User Authentication:

Registration and login tested with both valid and invalid university email addresses. System correctly rejected non-university emails and authenticated users only when enrollment numbers matched database records. Edge cases tested included expired enrollment IDs and malformed email formats.

Ride Creation & Retrieval:

Drivers created rides with various input combinations (different origins, destinations, capacities, preferences). All rides stored correctly in the database and appeared in appropriate search results.

Ride Matching:

Route-based filtering verified using the location graph. System correctly showed only rides with connected locations within the defined proximity threshold. Tested disconnected routes to confirm no false matches appeared.

Join Request Handling:

Complete request lifecycle tested: sending, receiving, accepting, and rejecting requests. Verified FIFO queue processing order, automatic capacity updates after acceptance, and ride status changes when reaching full capacity. Tested edge case where multiple requests arrive simultaneously.

Chat System:

Message exchange tested between multiple ride participants. Verified chronological ordering, persistent storage, and access control (non-participants blocked from viewing messages). Tested rapid consecutive message sending to check for race conditions.

6.2 Integration Testing

Cross-module interactions tested to ensure seamless component integration:

Database Synchronization:

All user actions (ride creation, request processing, messaging) verified to reflect correctly in SQLite database. Tested concurrent operations to ensure no data corruption. Duplicate entry prevention and invalid query handling verified through intentional malformed inputs.

Session Management:

Session token generation and validation tested across multiple login/logout cycles. Verified that expired or invalid tokens correctly prevented unauthorized access. Tested token persistence across system restarts.

Ride & Chat Access Control:

Confirmed chat messages accessible only to confirmed ride participants. Tested access attempts by non-members to verify proper rejection. Verified ride capacity checks before allowing chat access.

6.3 Challenges Faced and Solutions

Challenge 1: Insufficient Authentication Security

Problem: Initial implementation only verified university email domain (@cloud.neduet.edu.pk) but did not validate current enrollment status. This allowed graduated students or those with suspended enrollment to access the system, compromising the trusted community requirement.

Impact: Security vulnerability; platform could include non-active students.

Solution Implemented: Added enrollment number verification against a database table containing currently enrolled students. Modified authentication flow to require two-factor validation:

- Email domain check (university affiliation)
- Enrollment ID lookup (current enrollment status)

Outcome: Successfully restricted access to active students only. Authentication rejection rate for invalid enrollment IDs: 100% during testing.

Challenge 2: Location Graph Construction and Proximity Threshold

Problem: No clear methodology for selecting relevant locations or determining appropriate proximity radius for route matching. Initial approach with 2km threshold proved too restrictive—many valid routes excluded, resulting in very few search matches.

Impact: Poor user experience; legitimate ride matches not appearing in search results.

Solution Implemented: Selected 60 strategically distributed locations across Karachi based on common student residential areas and major transit routes

- Built weighted graph with edges representing realistic travel distances
- Experimented with proximity thresholds (2km, 4km, 6km) through test cases
- Settled on 4km radius as optimal balance between specificity and coverage

Technical Details: Graph constructed using adjacency list representation with distance weights. Connectivity check traverses adjacent nodes within threshold before declaring routes compatible.

Outcome: Ride matching improved significantly. Average search results increased from 0-1 rides (2km) to 3-5 rides (4km) for typical routes without including irrelevant matches.

Challenge 3: Gender Preference Filter Logic Error

Problem: Complex logic issue with gender-based ride filtering. Initially implemented as:

- "Female only" rides visible only to users who also selected "female only"
- Male users could not see general rides that happened to be posted by females

This violated requirement that female passengers should see all rides, while "female only" rides should be restricted.

Impact: Female users unable to see majority of available rides; "female only" preference became unusable.

Solution Implemented:

- Fixed database schema to properly store gender preference field with two states: "female_only", "no_preference"
- Added gender to students table to fetch authenticated user's gender during search to prevent manipulation through search form

Technical Fix: Gender fetched from authenticated session data (linked to enrollment records) rather than search input parameters, preventing users from changing gender at search time.

Outcome: Gender filtering now works correctly. Female users see all non-restricted rides plus female-only rides. Male users see no-preference rides. Verified through 20+ test scenarios with different gender combinations.

6.4 Results Summary

Testing Outcomes:

- Authentication correctly validates both email domain and enrollment status
- Ride creation, searching, and joining function reliably
- Graph-based route connectivity returns accurate matches within 4km threshold
- FIFO request processing maintains fairness
- Gender preference filtering works asymmetrically as required
- Chat system maintains message order and access control
- Database operations stable with no critical failures

Key Metrics:

- 50+ test cases executed across all modules
- 0 critical bugs in final version
- 100% authentication validation accuracy
- Average ride matching improved 400% after proximity adjustment
- Gender filter issue resolved with 100% accuracy post-fix

Lessons Learned:

1. Initial requirements may miss security edge cases (graduated students)
2. Real-world distance thresholds require experimentation with actual data
3. Complex conditional logic (gender filtering) requires careful state management and testing
4. Database schema design impacts feature implementation significantly

Overall Assessment: The system functions correctly within its defined scope and successfully meets all functional requirements. Critical challenges identified during development were resolved through systematic debugging and iterative testing. The platform is stable, secure, and ready for controlled deployment within the university environment.

7. User reviews:

User Review 1:

"UniRide was easy to use and the entire flow, on registering with my NED cloud ID to joining a ride, worked smoothly. The concept itself is very helpful for students who struggle with daily commuting. One improvement I would suggest is adding a notification system so users get alerts when a ride request is accepted or when a new ride becomes available on their route. It would also be great to have an option to schedule recurring rides for daily travel. Overall, the system is well thought out and functions reliably. Would use regularly if more students join."

User Review 2:

"I tested UniRide by registering, posting a ride, and interacting through the chat feature. The system behaved consistently but one problem is that the chat doesn't show if someone has read my messages so I can't communicate efficiently. Also if the ride is taking too long to start, I should be able to leave it and join another ride. Overall, even in its current form, the system demonstrates strong logic and solves an important problem for students. It's useful but feels like a beta version, definitely has potential though."

8. Future Scope

The UniRide system has strong potential for expansion to enhance usability, inclusivity, and reliability. Several improvements can be incorporated in future versions such as the following:

- **Faculty and Staff Integration:**

The platform can be extended to include university teachers and faculty members by adding their employee records to the database. This would allow them to register using employee IDs, making the system accessible to the wider university community.

- **Ride Scheduling and Pre-Booking:**

Introducing a scheduling feature that enables users to plan and book rides in advance. This would increase reliability, reduce last-minute coordination issues, and help users secure rides for recurring routes such as daily commutes.

- **Expanded Location Coverage:**

Adding more predefined locations and routes within the system to accommodate a broader range of pickup and drop-off points. This will improve route matching and allow more users to find rides that closely match their travel needs.

- **Live Location Tracking and ETA Estimation:**

UniRide can incorporate live location tracking for rides, allowing users to see the real-time position of their carpool. Coupled with an estimated arrival time (ETA) feature, this would make coordination easier, reduce waiting times, and enhance overall user convenience and reliability of the system.

9. Conclusion

The UniRide system successfully demonstrates how data structures and algorithms can be applied to solve real-world transportation challenges faced by university students. By combining efficient backend logic in C++ with secure user authentication and a structured SQLite database, the platform provides a reliable method for students to share rides based on proximity, preferences, and verified identity. The use of vectors, queues, graphs, and hash tables ensures fast data retrieval, scalable ride matching, and smooth communication between users.

Through extensive functional and integration testing, the system proved to be stable, responsive, and capable of managing essential operations such as user verification, ride creation, route matching, join request processing, and in-ride chat messaging. Each module works cohesively within a modular architecture, ensuring maintainability and future extensibility.

While UniRide in its current form meets the defined functional requirements, its modular design provides a strong foundation for future enhancements such as faculty integration, ride scheduling, and expanded route coverage. Overall, UniRide achieves its goal of offering a safe, efficient, and well-structured ride-sharing solution tailored to the needs of the university community.