School of Computing: Assessment Brief

Module title	Blockchain Technologies
Module code	COMP5125M
Assignment title	Coursework
Assignment type and description	It is a programming assignment where students are required to develop smart contracts for carpooling system.
Rationale	The aim of this assignment is to evaluate the students' knowledge of blockchain based systems and smart contract development skills
Word limit and guidance	N/A
Weighting	30%
Submission deadline	3.5.2024
Submission method	Student will upload their assignment on Gradescope
Feedback provision	Feedback will be provided online through Gradescope
Learning outcomes assessed	Design decentralized applications and implement smart contracts, Understand the context to which distributed ledgers are applied
Module lead	Evangelos Pournaras
Other Staff contact	Zhuolun Li

1 Assignment Guidance

Context

Carpooling, also known as car sharing or ride sharing, involves individuals sharing a single vehicle for travel in the same direction or to a common destination, instead of driving separate cars. The primary aim of carpooling is to reduce the number of vehicles on the road, offering a sustainable transportation option. It addresses challenges such as traffic congestion, environmental impact, fuel costs, and energy conservation, while increasing social interaction among people.

A blockchain-based carpooling service offers distinct advantages over traditional, centralized carpooling apps. Its primary benefit is to enhance trust among users, thanks to the transparent and immutable nature of blockchain technology. Furthermore, blockchain enables smart contract-based, automatic payment settlements, ensuring fair compensation for drivers. This approach not only simplifies the payment process but also reduces intermediary fees. Overall, a blockchain-based carpooling system enhances transparency, security, and efficiency, providing a more trustworthy and cost-effective solution compared to conventional carpooling apps. In this coursework, you are required to develop a smart contract in Solidity for a carpooling system.

Scenario

This coursework focuses on a carpooling system with two types of participants: *drivers* and *passengers*, in a simplified scenario described below:

- **Rides**: Only drivers can create rides. It is assumed that drivers are honest in attending and completing rides. Passengers can view available rides and book seats. Passengers are assumed to always attend the rides they book, hence ride cancellations are not considered.

- **The map**: The system operates on a map with three independent locations: A, B, and C. Vehicles can travel between these points (e.g., A to B, B to C). Passengers can book a ride only if the starting point and destination match their travel requirements.

- **Time tracking**: For simplicity, we assume rides are created one day ahead of the travel date, eliminating the need to track days. Journey start times, precise to the hour, should be recorded. Time can be represented as an unsigned integer from 0 to 23 in the smart contract, where 0 represents 00:00, 1 represents 01:00, and so on.

- **Payment**: Ether is the currency used in the system. Passengers must pay for their seat in Ethers in advance when booking a ride. The seat price, set by the driver, is automatically transferred to the driver‘s account once the ride is marked as complete.

The Smart Contract

You are provided with a template for the smart contract in *contract_template.sol*. This template outlines the structure and essential functionalities that you need to implement. The details of these functionalities are as follows:

Basic Functionalities

- **Registration**: Before using any other functions, a blockchain address must register as either a driver or a passenger in the contract. For instance, the function to create a ride should only be used by a registered driver, and the function to join a ride should verify if the caller is a registered passenger.
- **Rides creation and recording**: Drivers can create rides by providing information including:
  - travel time (journey start time, represented as an unsigned integer from 0 to 23)
  - available seats in the ride
  - price of a seat
  - starting point of the ride
  - destination of the ride

  When rides are stored in the smart contract, additional information should be recorded, including:
  - unique ID of the ride (starts counting from 0)
  - blockchain address of the driver

  - status of the ride (the possible status are *BookingOpen, FullyBooked, Started, Completed*)
  - blockchain addresses of the passengers who have booked the ride

  The function should include necessary checks to ensure the ride is valid: the travel time should be a number between 0 and 23; the starting point and destination should be different; the price of a seat should be greater than zero; the available seats should be greater than zero.

- **Rides querying**: Passengers can find suitable rides by using the *findRides* function. This function returns all ride IDs that match the passenger's starting point and destination and are in *BookingOpen* status. Passengers can then query detailed information about a ride using the *getRideById* function provided in the template.

- **Rides booking**: Passengers book rides using the *joinRide* function, which requires a deposit of the seat price in Ethers. This function updates the available seats and passenger addresses accordingly. It also updates the ride status to *FullyBooked* when all seats are booked.

- **Starting and completing rides**: On departs, drivers call the *startRide* function to update the ride status to *Started*. On completing rides, they use the *completeRide* function to mark the ride as *Completed*, which triggers the transfer of the passengers' payments to the driver's account.

Coordination Mechanism

In the current setup, passengers individually book rides by manually selecting the most suitable option and sending a booking request to the contract. However, with access to information about all available rides and the preferences of all potential passengers, more efficient arrangements can be made.

For instance, consider a scenario where two available rides go from point A to point B, each with only one seat left. One departs at 8:00, the other at 14:00. Two passengers, Alice and Bob, wish to travel from A to B. Alice prefers to leave at 6:00, and Bob at 10:00. If they book separately and Bob books first, he would likely choose the 8:00 ride, closer to his preferred time. This leaves Alice with the 14:00 ride, which is far from her preferred time. However, through coordination, we could arrange for Bob to take the 14:00

ride and Alice the 8:00 ride. This adjustment would result in a smaller total deviation from their preferred travel times.

Let's denote the preferred travel time of a passenger as $t_{preferred}$ and the actual travel time as t_{actual}. The **travel time deviation** is calculated as $|t_{preferred} - t_{actual}|$. The **total travel time deviation** is the sum of travel time deviation of all passengers. In the example above, the original total travel time deviation is $|10 - 8| + |6 - 14| = 10$. If Alice and Bob agree to coordinate, the total travel time deviation is $|10 - 14| + |6 - 8| = 6$.

The aim of the coordination mechanism is to minimize the total travel time deviation in a gas-efficient way. It is crucial that this goal is achieved without excluding passengers from ride assignments. If an available ride matches a passenger's starting point and destination, they must be assigned to that ride, regardless of the travel time deviation. Failure to do so will result in a very low mark in this part.

To implement this, complete two functions in the template: *awaitAssignRide* and *assignPassengersToRides*, detailed below:

- **awaitAssignRide**: Passengers who opt for coordination use this function instead of *joinRide*. They provide their starting and destination points, preferred travel time, and the price they are willing to pay. The price they are willing to pay is transferred to the contract as the deposit. The function ensures the deposit is in place before recording the passenger's information. For simplicity, assume coordinating passengers always pay a good amount of deposits that are enough to join any rides (so you don't need to worry about this factor when assigning rides). Rather than immediately assigning a ride, this function records the passenger's information in the contract, awaiting the *assignPassengersToRides* function call.

- **assignPassengersToRides**: This function, executed by the system, assigns rides to passengers who have used *awaitAssignRide*. It aims to minimize the sum of differences between passengers' preferred and actual travel times. You are free to implement any assignment algorithm that meets this objective. Remember, when assigning a passenger to a ride, update the available seats, passenger addresses, and potentially the ride status, similar to the *joinRide* function. Moreover, if the allocated price is cheaper than the price a passenger prepaid, the difference should be refunded to the passenger; if a passenger is not assigned to any ride, the deposit should be fully refunded.

There are no restrictions on the data structures or auxiliary functions you can use to implement the coordination mechanism. Grading will be based on (1) the correctness of the implementation, e.g. one passenger should not be able to call **awaitAssignRide** twice, **assignPassengersToRides** should assign passengers to rides where it is possible; (2) the effectiveness of the coordination mechanism in reducing the total travel time deviation; and (3) the gas efficiency of the implementation.

2 Tasks and Requirements

Based on the information provided above, you are required to complete the following tasks:

1. Complete the smart contract template provided in *contract_template.sol*. You are required to complete the basic functionalities and the coordination mechanism.
    - You are allowed (but not required) to: add auxiliary functions, structures, contract variables, as per your requirements. For example, your *findRide* function can call another function you created to check if a ride matches the passenger's travel requirements.
    - You are **not** allowed to: change or delete anything declared in the template, including structs, function names, input parameters and return values in the template. External libraries are also not allowed. Failure to do so will make your contract fail tests, resulting in zero grading.
2. Complete a short documentation in the markdown file using the template provided in *readme_template.md.* The template provides instructions on what to include in the documentation. You are required to:
    - Provide a brief description of your proposed coordination mechanism.
    - If you use additional data structures, auxiliary functions other than the one provided in the template, please provide a short rationale.
    - If you implement any test cases to test your smart contract, please provide a short description of the test cases.

Recommended Development Environment

You are recommended to use Hardhat as your development environment. To set up the development environment, follow the steps below as a general guidelines (more resources for Hardhat can be found on Minerva):

1. Install Hardhat on your computer.
2. Create a new Hardhat JavaScript project with *npx hardhat init*. Use the default settings.
3. Copy the template file *contract_template.sol* to the */contracts* folder of the project.
4. Copy the test file *basic_contract_test.js* to the */test* folder of the project.
5. Implement the smart contract and test it using *npx hardhat test*.

Contract Size Limit

Ethereum has a limit of 24576 bytes for the code size of a smart contract. You must keep the contract size below this limit. If you see a warning about the contract size when compiling, or an error of deployment failure due to the contract size when running tests, it means your contract is too large.

If you reach the limit, it is likely because you implemented a sophisticated coordination mechanism that requires a large amount of code. In this case, you can move some coordination logic (with auxiliary functions) to a library while keeping **awaitAssignRide** and **assignPassengersToRides** functions. If it still exceeds the limit, here are other solutions to follow.

3 General guidance and study support

The reading list and study support material will be available on Minerva. Refer to the lab manuals and the smart contract development lecture slides for the course. You can also refer to the Solidity documentation and the Ethereum documentation for additional information.

4 Assessment criteria and marking process

The assessment criteria are provided at the end of this document in the form of rubrics. Please refer to Section 8 of the document.

Before submitting your work, you are **strongly recommended** using the test cases provided in *basic_contract_test.js* to make sure your implementation of the basic functionalities is correct. As the assignment will be marked by Autograder on Gradescope, your smart contract will be tested using a comprehensive set of test cases that are not limit to the provided basic test cases.

The grading test cases include: (1) Unit testing of each functions in both positive (e.g. registered driver should be able to create a ride) and negative (e.g. unregistered addresses try to create rides) cases. (2) Integration testing of the whole system that simulates scenarios with multiple drivers and passengers interacting with the system.

Therefore, it is also recommended writing your own test cases with different scenarios to test the behaviour for your smart contract. If you implement any additional test cases, briefly discuss what you tested in the readme file in your deliverables. The test cases you write are not used for grading, but could be used as additional reference in the case of having unexpected auto grading outputs of your smart contract.

Important: You are responsible for ensuring your contract compiles correctly. Failure to compile will result in a mark of zero. This includes making sure the contract code size below the limit of 24576 bytes. Moreover, ensure the basic functionalities (e.g. *passengerRegister, driverRegister, createRide, joinRide*) are working correctly because the advanced test cases are based on the basic functionalities. Failure to do so will result in very low marks.

5 Deliverables

You are required to upload on Gradescope: **(1) your code file(s) of smart contracts** along with **(2) a text (readme.md) file**. In the readme file, you are required to give an overview of the proposed coordination mechanism.

6 Submission requirements

This is an individual assignment (no teams). Submit your files on Gradescope and names of your files should be as follows:

Firstname_Surname_readme.md
car_pooling.sol

7 Academic misconduct and plagiarism

- Students at University of Leeds are part of an academic community that shares ideas and develops new ones.
- You need to learn how to work with others, how to interpret and present other people's ideas, and how to produce your own independent academic work. It is essential that you can distinguish between other people's work and your own, and correctly acknowledge other people's work.
- All students new to the University are expected to complete an online Academic Integrity tutorial and test, and all Leeds students should ensure that they are aware of the principles of Academic integrity.
- Generative artificial intelligence systems, such as ChatGPT, are not allowed in this assignment.
- When you submit work for assessment it is expected that it will meet the University's academic integrity standards.
- If you do not understand what these standards are, or how they apply to your work, then please ask the module teaching staff for further guidance.

8 Assessment/marking criteria grid

	Requirement and delivery
$> 90\%$	All functions are correctly implemented. The *assignPassengersToRides* function effectively reduces the total travel time deviation in all test scenarios compared to the non-coordinating approach under the same setting. The implementation of functions is gas efficient.
$< 90\%$	All functions are correctly implemented. The *assignPassengersToRides* function reduces the total travel time deviation in some test scenarios compared to the non-coordinating approach under the same setting. Gas efficiency can be improved.
$< 70\%$	All functions are correctly implemented. The *assignPassengersToRides* function is able to assign passengers to rides, the assignment does not improve on total travel time deviation compared to the non-coordinating approach under the same setting.
$< 60\%$	The basic functions including register drivers, register passengers, create ride, find rides, join ride are correctly implemented. Coordination mechanism is proposed but the functions *awaitAssignRide* and *assignPassengersToRides* are not fully functioning.
$< 50\%$	If smart contract is not compiling or smart contract fails to perform register drivers, register passengers, create ride, find rides, join ride, start ride or complete ride correctly. Coordination functions *awaitAssignRide* and *assignPassengersToRides* are not completed.