Task 1:

Meeting notes are posted in the github repository:

https://github.com/Laith-i/Comp-5047--Software-Engineering

Personal Reflection:

As a leader, I leveraged systematic project management methodologies to guide our development. To begin with, I held initial meetings to explain the goals, boundaries, and limits of the project. Those discussions, which were well recorded in our GitHub repository, allowed us to achieve a common understanding and promoted openness. I introduced intermediate milestones and assessment points so the team could gauge our progress relative to established benchmarks. When dependencies showed up—most especially across subsystems—I actively managed those by way of task redistribution or timeline adjustments to ensure overall project stability. This proactive approach is going to mitigate risks and make sure that the project's trajectory stays true to our strategic vision and academic standards.

Individual Contributions to the Overall Project:

My technical contributions were inherently bound to the strategic guidance that I provided. In articulating subsystem requirements, I also contributed to the modeling and documentation activities, ensuring that our class diagrams, sequence diagrams, and architectural models were accurate and compliant with the set Unified Modeling Language (UML) standards. Such thorough documentation improved clarity, optimized later development activities, and made quality assurance practices really effective. Assuming leadership roles gave me substantial knowledge on how to lead complex software engineering projects in an academic setting. I honed my ability to balance technical requirements of project management with interpersonal dimensions, including effective communication.

Task 2:

(a) *CloudTables-Customer*: a mobile app for running on smart phones for restaurant customers to make table booking, food order, bill payment, and service review and ranking, etc.

Security

-Security And Privacy

- Data Encryption : CloudTables-Customer encrypts customer data at the application level—booking data, credit card info, ratings, etc.—and transmits it to the cloud so that if someone attempts to access info in transit—or on the server—it's unreadable. More specifically, the application uses AES-256 for data encryption and RSA for key exchange between the application and cloud. Therefore, the strength of encryption is strong, yet the user experience is seamless and safe.

-Decentralised Identity Management

- Decentralised Identifiers (DIDs): Implement Decentralised Identifiers (DIDs) to allow users to create and control their identities. This is a preferable option to having

identity characteristics stored in a central location, which is susceptible to being hacked in a data breach affecting millions. Self-Sovereign Identity

- (SSI): Implement self-sovereign identity capabilities for users. Users decide which information needs to be disclosed. For instance, a person can prove that they are over 18 without disclosing their date of birth; only the identifying characteristics will be displayed.

-Privacy-Preserving Analytics

- Homomorphic Encryption: Employ homomorphic encryption to be able to compute on ciphertext so that the system learns things (like when patrons book most often) without ever seeing decrypted patron information. This means analytics and feedback are produced without compromising privacy and security.
- Federated Learning: Implement federated learning when analysing patron usage patterns to enhance app functionality. That way, the computing happens without ever leaving the patrons' devices, and only the aggregate results come back to the main server for enhancement. This means no individual patron's information ever leaves their device—which is privacy and security beyond belief.

Performance

-Edge Computing For Latency Reduction

- Edge Computing: Instead of sending data all the way to the cloud to get processed, it gets processed where it needs to be processed, closer to the end user devices. Edge nodes in particular geographic locations allow for the processing of updating table availability or suggested menu options to occur at the edge so that it is faster in response time and loaded off the cloud.
- Edge Caching: Data that is needed frequently—menus for restaurants, busiest times for reservations—is cached at edge nodes so that latency is decreased for those users who need it frequently.

-Granular Data Partitioning

- Partition Data Based on Booking Types: Partition data in different reservation types, such as reservations for dine-in, take away, and private events; so that the querying isn't as load heavy and retrieval can happen faster without needing to go into a backed up single table for all bookings.
- Column Storage for Frequent Queries: Use column storage for data that's frequently queried—what do people order most, what time of day are bookings the most—so

that analytical queries are easier and faster. Column storage is best suited for operations with a high read rate and allows for insights to be had quicker.

-Optimised Mobile Experience

- Device Resource Adaptation: Utilise mobile hardware acceleration and change expectations based on the device. For instance, stronger and newer devices can use more powerful graphics and transitions while weaker devices are given a simplistic UI to prevent overexerting any one hardware capability while facilitating effective functioning across the application.
- Offline Functionality with Local Caching: Users can see past purchases and items in a shopping cart currently without opening the app when online. By caching this relevant information locally, access is almost instantaneous—even with Wi-Fi off. However, once the device powers back on, the app readjusts to sync with the cloud.

Reliability

-Blockchain For Distributed Reliability

- Immutable Customer Review: Store customer feedback in the form of service reviews using blockchain for authenticity and reliability of user information. It will avoid fraudulent modification and preserve the integrity of customer experiences recorded.
- Blockchain for Booking Ledger: A private blockchain could act as a distributed ledger for all bookings. Any transaction for booking takes place on-chain, forming it and rendering a decentralised means of accuracy and integrity across multiple nodes. This blockchain-based ledger can make it nearly impossible for booking information to be lost or tampered with, even if the central system experiences outages.

-Distributed Transaction Management

- Two-Phase Commit Protocol: Implement a two-phase commit protocol to ensure that distributed transactions (i.e., reserving a table, charging the customer, and sending the confirmation email) are properly completed and consistent across each microservice that needs to have such a transaction. This way, the customer payment goes through in every applicable microservice and the reservation is confirmed; or if something fails, neither goes through and the one table is available up for grabs with one in-progress potential option.
- SAGA Pattern for Long-Running Transactions: Implement the SAGA pattern for long-running transactions like the reservation itself. Reserving with an order will be broken down into microtransactions that can be compensated. If any of these microtransactions fail, compensating transactions will be executed to roll back any prior actions so that the system is still in a consistent state.

-Load Adaptive Redundancy

- Load Based Adaptive Replication: Implement a load-based adaptive replication approach which increases the number of replicas under increased load (i.e., when the system is busy and people are trying to book a reservation on a Saturday evening). This ensures read/write processes occur in a timely fashion.
- Hot Standby Replicas: Include hot standby replicas of the primary database which are instantly updated with the master instance and can take over failover instantly so that reservation or ordering activity is never offline.

Scalability

-Cloud Bursting For Dynamic-Scaling

- Hybrid Cloud Bursting: Augment capacity to a public cloud on a short-term basis as needed (weekends, busy holidays, etc.). While regular processing has resources/lives/calls in a private cloud because it's more cost-efficient, when traffic extends beyond what a private cloud can offer, everything in excess of the private cloud is "burst" into a public cloud like AWS or Azure to provide them with the Scalability without stressing resources
- Seamless Transition: Implement Proper data syncing between private and public clouds so that clients do not feel latency when bursting.

–Container Orchestration With Horizontal Pod AutoScaling

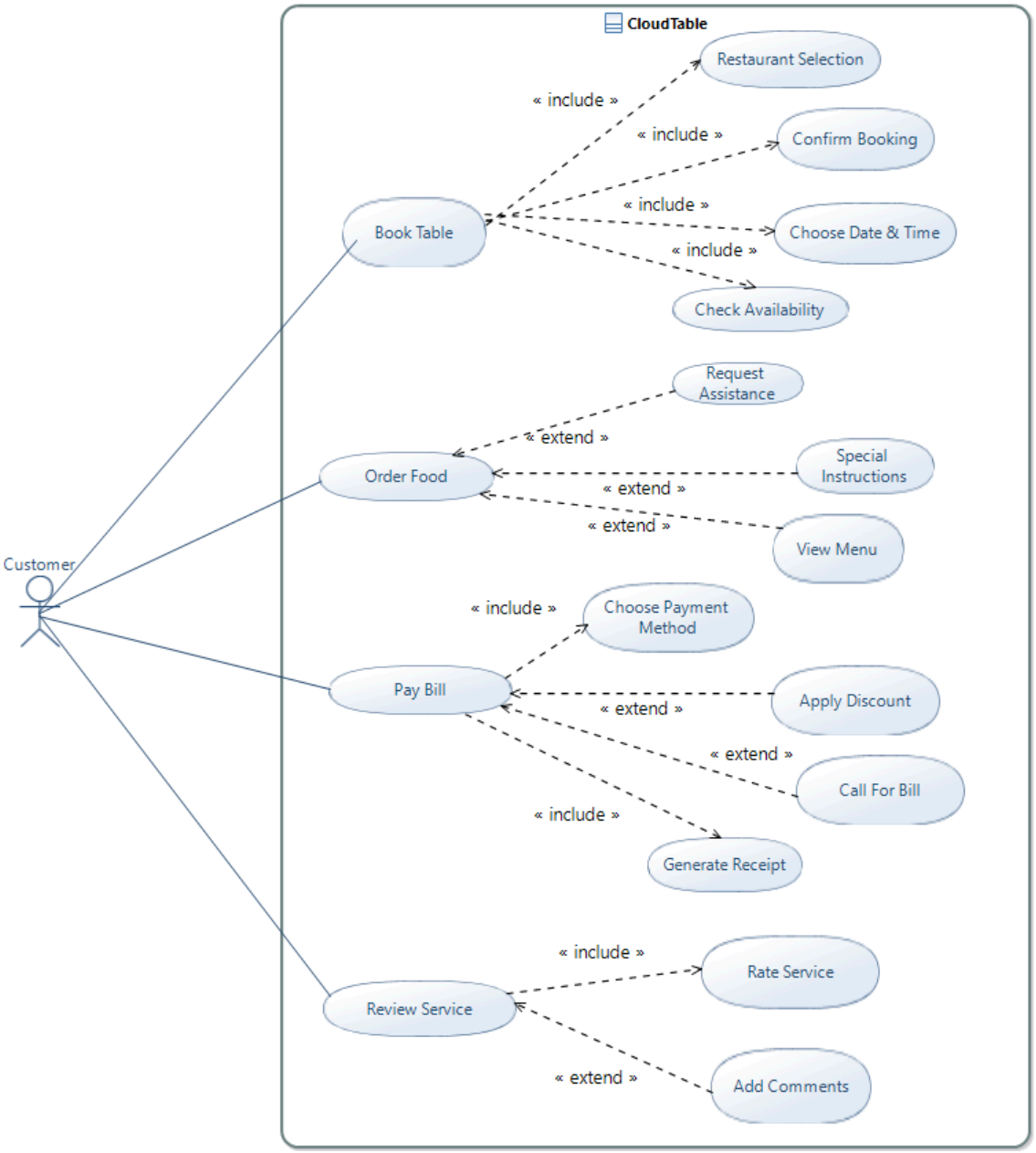
- Kubernetes for Orchestration: Deploying the application using Kubernetes will allow for horizontal pod autoscaling based on CPU and memory usage. If a traffic spike is detected, the system will automatically generate more pods to manage the load—with separate scaling for different subsystem components.
- Node Autoscaling: Kubernetes also provides autoscaling at the cluster level to join or leave cluster nodes so that during low traffic times, resources are not over-provisioned, but during high demand times, resources are not lacking.

-Containerized Microservices With Function Isolation

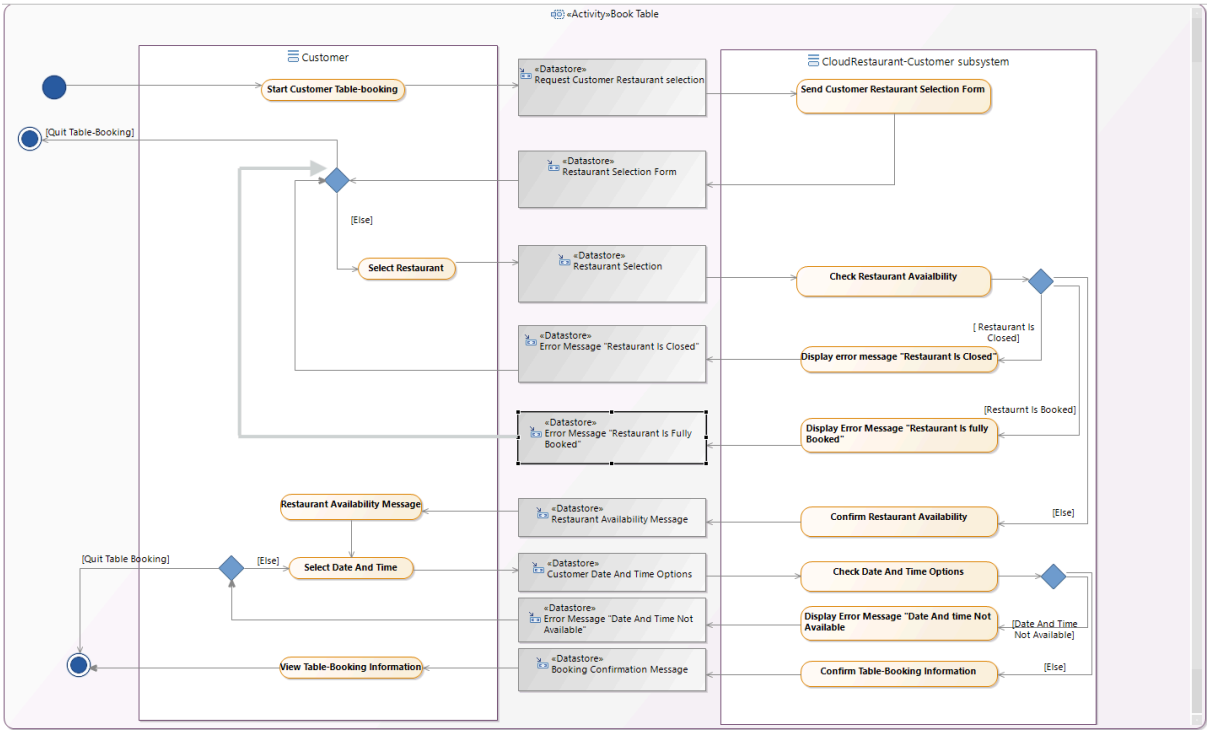
- Isolated Function Containers: Package each key function, for example table booking or ordering of food, in separated containerized microservices using light container platforms like Docker Swarm. This will ensure that the scaling of certain services is easily done when needed, without affecting other parts of the subsystem.
- Resource-Aware Scheduling: Make use of the resource-aware schedulers to route the workloads to the optimal node by utilising recent usages. It will scale the best at peak traffic, resource allocation, enhancing scalability and efficiency.

Task 3:

(a) *Use Case Model* (10 Marks, Individual effort): Each member of the team should develop one Use Case Diagram to define the use cases of the subsystem to specify the scope of the software engineering project.

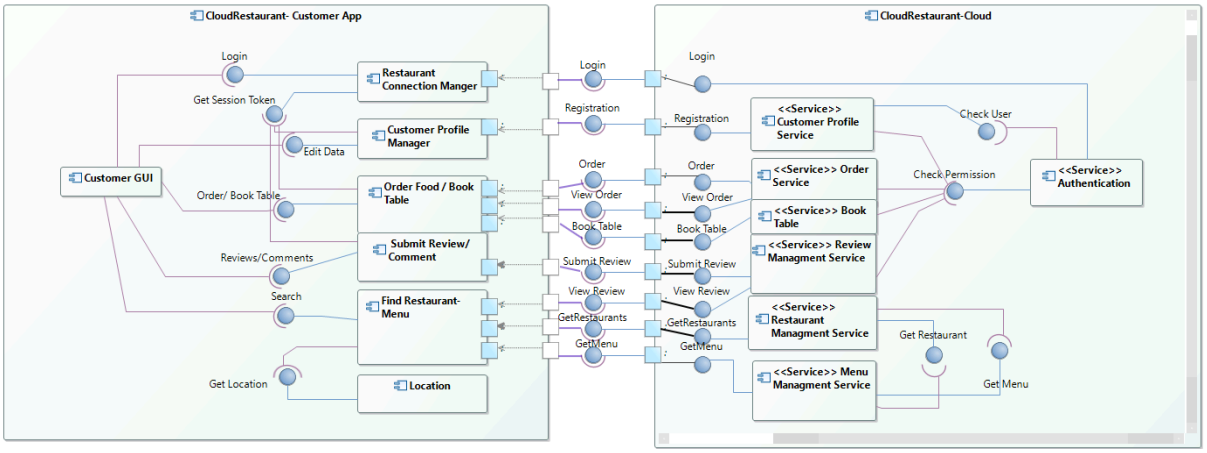


(b) *Activity Model* (10 Marks, Individual effort): Each team member should select one use case of your subsystem to produce one Activity Diagram for the selected use case to specify the interactions between a user and the subsystem.


«Activity»Book Table
Customer
CloudRestaurant-Customer subsystem
Start Customer Table-booking
«Datastore»
Request Customer Restaurant selection
Send Customer Restaurant Selection Form
[Quit Table-Booking]
«Datastore»
Restaurant Selection Form
[Else]
Select Restaurant
«Datastore»
Restaurant Selection
Check Restaurant Availalbility
[Restaurant Is Closed]
«Datastore»
Error Message "Restaurant Is Closed"
Display error message "Restaurant Is Closed"
[Restaurnt Is Booked]
«Datastore»
Error Message "Restaurant Is Fully Booked"
Display Error Message "Restaurant Is fully Booked"
Restaurant Availability Message
«Datastore»
Restaurant Availability Message
Confirm Restaurant Availability
[Else]
[Quit Table Booking]
[Else]
Select Date And Time
«Datastore»
Customer Date And Time Options
Check Date And Time Options
«Datastore»
Error Message "Date And Time Not Available"
Display Error Message "Date And time Not Available
[Date And Time Not Available]
View Table-Booking Information
«Datastore»
Booking Confirmation Message
Confirm Table-Booking Information
[Else]

Task 4:

(a) *Architecture of the subsystem* (10 Marks, Individual effort): Each member of the team should produce an architectural design of your subsystem with focus on the microservices your subsystem provides and the microservices that your subsystem requests and other subsystems provides.



Specification of Microservices:

Component	Authentication Service
Description	This microservices handles user authentication by validating login credentials and checking permissions for accessing various functionalities.
Stereotype	Microservice
Required Interfaces	CheckUser
Provided Interfaces	Login, Check Permission

Component	Customer Profile Service
Description	This microservice manages customer profile data, updates, and verification details.
Stereotype	Microservice
Required Interfaces	Check Permission
Provided Interfaces	Registration

Component	Order Service
Description	This microservice manages table bookings and order management, ensuring customers can book tables and order seamlessly.
Stereotype	Microservice
Required Interfaces	Check Permission
Provided Interfaces	Order, View Order

Component	Booking Table Service
Description	This microservice manages table bookings ensuring customers can book tables seamlessly.
Stereotype	Microservice
Required Interfaces	Check Permission
Provided Interfaces	Book Table

Component	Review Management Service
Description	This microservice manages customer reviews and comments for restaurants allowing to view, submit, and manage reviews
Stereotype	Microservice
Required Interfaces	Check Permission
Provided Interfaces	Submit Review, View Review

Component	Restaurant Management Service
Description	This microservice provides restaurant information such as, location, details of the restaurant, cuisine, and the menu
Stereotype	Microservice
Required Interfaces	Get Menu
Provided Interfaces	Get Restaurant

Component	Menu Management Service
Description	This microservice provides detailed menu information for each restaurant, including dishes, prices, and availability
Stereotype	Microservice
Required Interfaces	Get Restaurant
Provided Interfaces	Get Menu

Specification Of Interfaces:

Name	Login	
Provider	Authentication	
Operation	Signature	customerLogin(user_id: string, password: string) : sessionToken : string
	Function	Verifies the user's credentials against stored values. If successful, it returns a sessionToken, a string that represents the user's active

		session and can be used for subsequent authenticated requests. If the credentials are invalid, an error message is returned.

Name	Check Permission	
Provider	Authentication	
Operation	Signature	checkPermission(user_id: string, action: string): permissionStatus: boolean
	Function	Determines whether a user has the appropriate permissions to perform a specific action (e.g., editing a profile, booking a table). Returns true for valid permissions and false otherwise.

Name	Registration	
Provider	Customer Profile	
Operation	Signature	registerUser(name: string, email: string, password: string): confirmationStatus: Boolean
	Function	Registers a new user by creating a profile with their details. Returns a confirmation status (true for success, false for failure).

Name	Book Table	
Provider	Table Booking	
Operation	Signature	bookTable(user_id: string, restaurant_id: string, dateTime: DateTime): confirmationCode: String
	Function	Allows users to book a table

Name	order	
Provider	Ordering	
Operation	Signature	addOrder(in order_id: string): confirmationCode: String
	Function	Allows users to order food. Returns a unique confirmationCode upon successful order adding or an error message otherwise.

Name	View Order

Provider	Ordering	
Operation	Signature	getOrderDetails(in order_id: string):
	Function	Allows users to order food. Returns a unique confirmationCode upon successful order adding or an error message otherwise.

Name	Submit Review	
Provider	Review Management	
Operation	Signature	submitReview(user_id: string, restaurant_id :string, rating: integer, comment: string) : confirmation: Boolean
	Function	Processes user-submitted reviews and ratings for a restaurant. Ensures valid input and permissions before storing the review. Returns true on success, or false otherwise.

Name	View Review	
Provider	Review Management	
Operation	Signature	viewReview(restaurant_id :string) : reviewList: List<Review>
	Function	Fetches reviews and ratings for the specified restaurant_Id from the database. The reviewList is a list of review objects, each containing fields such as review_Id, user_Id, rating, and comment. The function queries the database and returns the reviews sorted by recency or rating.

Name	Get Restaurant	
Provider	Restaurant Management	
Operation	Signature	findRestaurant(location: string, cuisine: string) :List<Restaurant>
	Function	Queries the database to retrieve a list of restaurants that match the given location and cuisine parameters. Each Restaurant object in the returned list includes fields such as restaurant_Id, name, address, cuisine, rating, and contactInfo. Results are filtered and sorted based on relevance or proximity.

Name	Get Menu	
Provider	Menu Management	
Operation	Signature	fetchMenu(restaruant_id:string): List<MenuItem>
	Function	Queries the database to fetch the menu items for the specified restaurant_Id. The returned list contains MenuItem objects, each with fields such as menuItem_Id, name, description, price, availability, and category (e.g., Appetisers, Mains, Desserts).

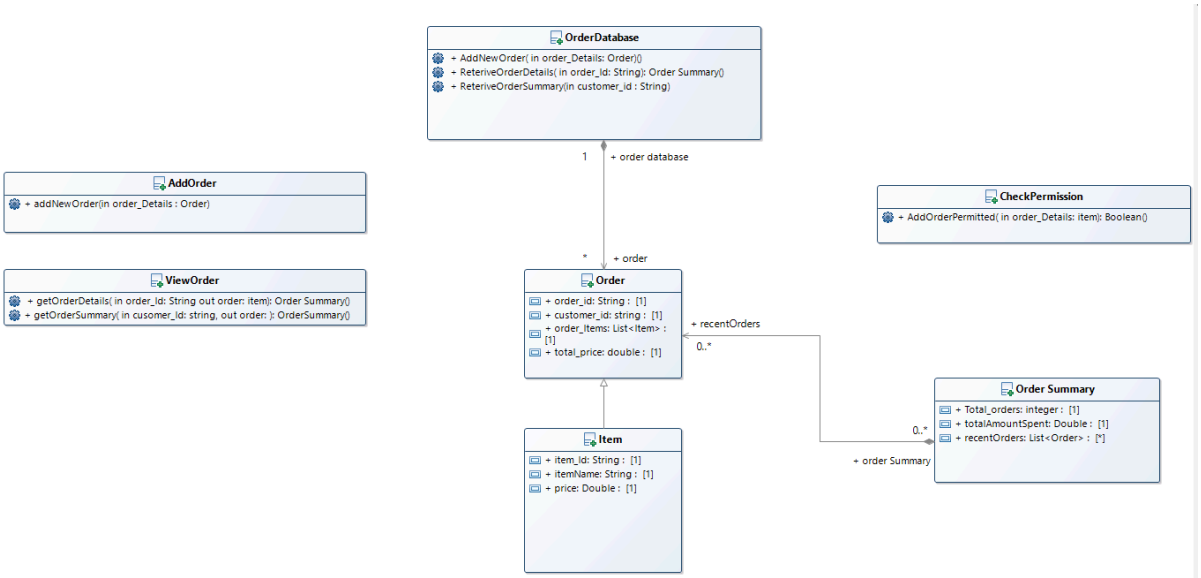
Database Name:	CloudRestaurant Database
Description:	This database stores information related to the CloudRestaurant system, including customer details, restaurant data, menus, reviews, and table bookings.
Type:	Relational Database
Design(Tables)	Customers, Restaurants, Menu, Reviews, Bookings
Table(fields) Customers	Customer_id, name, email, password, phoneNumber, address
Table(fields) Restaurants	Restaurant_id, name, location, rating, cuisine, contact info
Table(fields) Menu	Item_id, restaurant_id, name, description, price, availability
Table(fields) Reviews	Review_id, restaurant_id, customer_id, rating, comment, timestamp
Table(fields) Bookings	Booking_id, restaurant_id, customer_id, dateTime, numberOfGuests, confirmationCode

Task 5: (a) *Structural Model* (10 Marks, Individual effort): Each student should **select one component** in the architectural design of your subsystem and develop a Class Diagram to define the structure of the component.

Note:

a. *The selected component must be in the architectural model of your subsystem, but not in any other subsystems.*

b. *The UML class diagram must contain the classes and relationships between them. You must give the attributes and methods of the classes.*



Task5: (b) *Behaviour Model* (10 Marks, Individual effort): Each student should produce a Sequence Diagram for the same component selected in Task 3(a) to define the dynamic behaviour of the component.

Note:

a. *The sequence diagram should specify the internal process of the interactions between the objects inside the component. It must be consistent with the structural model that you produced in Task 3(a).*

b. *The behaviour model must cover all possible scenarios of the operations of the component using advanced modelling facilities such as frames.*